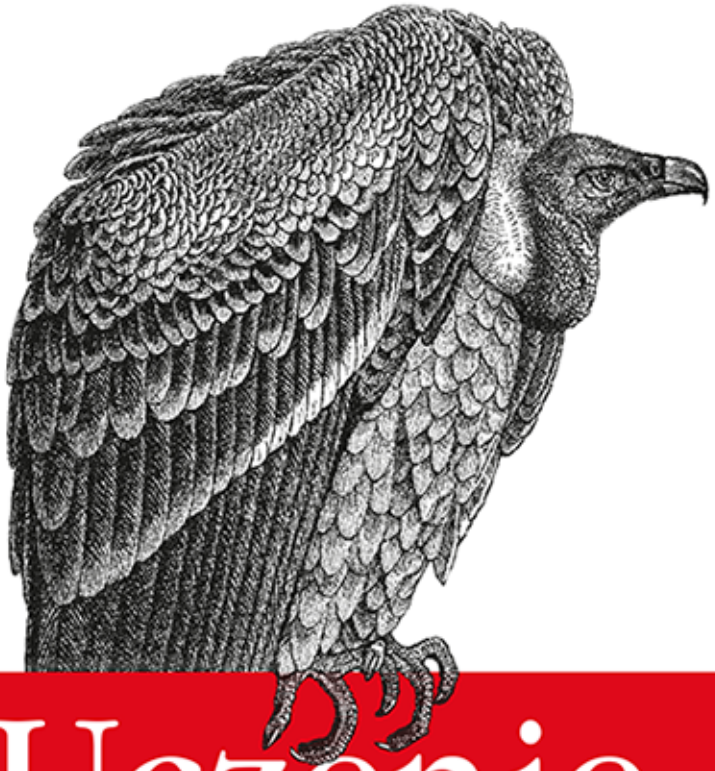
Wyciągnij najlepsze wnioski z dostępnych danych!



Uczenie maszynowe

dla programistów

HELION

O'REILLY®

Drew Conway, John Myles White

Tytuł oryginału: Machine Learning for Hackers

Tłumaczenie: Przemysław Szeremiota

ISBN: 978-83-246-9816-5

© 2015 Helion S.A.

Authorized Polish translation of the English edition of Machine Learning for Hackers 9781449303716 © 2012 Drew Conway and John Myles White.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności aniza ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłez wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/umapro
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 7.

Optymalizacja — łamanie szyfrów

Wprowadzenie do optymalizacji

Jak dotąd, większość algorytmów opisywanych w książce traktowaliśmy jak swego rodzaju „czarne skrzynki" — skupialiśmy się na zrozumieniu danych wejściowych i interpretacji danych wyjściowych. Zasadniczo więc traktowaliśmy algorytmy uczenia maszynowego jako funkcje biblioteczne, specjalizowane do wykonywania zadań predykcyjnych.

W tym rozdziale przyjrzymy się jednak technikom wykorzystywanym do implementowania najprostszych algorytmów uczenia maszynowego. Jako punkt wyjścia weźmiemy funkcję do dopasowywania prostych modeli regresji liniowej z jednym predyktorem. Przykład ten pozwoli na spojrzenie na problem dopasowania modelu do danych jako problem optymalizacji. Problem optymalizacji to taki problem, w którym dysponując pewnym urządzeniem albo mechanizmem, chcemy dostrajać jego działanie i mierzyć wpływ dostrajania na sprawność urządzenia bądź mechanizmu. Zadanie polega na znalezieniu najlepszych możliwych ustawień, to znaczy takich, które maksymalizują pewną prostą miarę sprawności urządzenia. Zbiór tych ustawień nazwiemy **punktem optimum**, a docieranie do tego punktu nazwiemy **optymalizacją**.

Po przybliżeniu podstaw działania optymalizacji przejdziemy do zadania zasadniczego, a mianowicie zbudowania prostego systemu łamania szyfrów (kodów), w ramach którego rozszyfrowywanie tekstu będziemy traktować jako problem optymalizacji.

Skoro zamierzamy zbudować własną funkcję regresji liniowej, wróćmy do naszego standardowego przykładowego zbioru danych — wagi i wzrostu. Załóżymy (sprawdziliśmy to w poprzednich ćwiczeniach), że będziemy w stanie przewidzieć wagę poprzez obliczenie funkcji wagi od wzrostu. A konkretnie — załóżymy, że taka funkcja będzie liniowa. W języku R wyglądałaby ona następująco:

```
height.to.weight <- function(height, a, b)
{
  return(a + b * height)
}
```

W rozdziale 5. zapoznaliśmy się ze szczegółami obliczania nachylenia i przesunięcia linii predykcji za pomocą funkcji `lm`. W tym przykładzie nachylenie reprezentuje parametr `b`, a przesunięcie reprezentuje parametr `a` (mówi on, ile ważyłaby osoba o zerowym wzroście).

Kup książkę Poleć książkę

Jak przy tak zadanej funkcji zdecydować o tym, jakie wartości `a` i `b` będą najodpowiedniejsze? Tu właśnie zaczyna się problem optymalizacji — zdefiniujemy miarę jakości naszej funkcji w zadaniu predykcji wagi na podstawie wzrostu, a potem będziemy zmieniać wartości `a` i `b` dopóty, dopóki nie osiągniemy możliwie najwyższej skuteczności predykcji.

Jak to zrobić? Cóż, robi to już za nas funkcja `lm` — optymalizuje ona prostą funkcję błędu, wyznaczając w ten sposób najlepsze wartości `a` i `b` za pomocą bardzo specyficznego algorytmu, działającego wyłącznie dla zwyczajnej regresji liniowej.

Spróbujmy więc wywołać funkcję `lm` i podejrzeć wyznaczone wartości `a` i `b`:

```
heights.weights <- read.csv(file.path('data', '01_heights_weights_genders.csv'))

coef(lm(Weight ~ Height, data = heights.weights))
#(Intercept)      Height
#-350.737192    7.717288
```

Dlaczego właśnie takie wartości zostały ustalone dla parametrów `a` i `b`? Aby odpowiedzieć na to pytanie, musimy znać funkcję błędu używaną w `lm`. W rozdziale 5. była mowa o tym, że `lm` wykorzystuje miarę błędu o nazwie „kwadrat błędu", obliczaną następująco:

1. Ustal wartości dla `a` i `b`.
2. Dla danej wartości wzrostu oblicz przewidywaną wagę.
3. Odejmij wagę przewidzianą od faktycznej — wystąpi błąd.
4. Podnieś błąd do kwadratu.
5. Zsumuj kwadraty błędu predykcji ze wszystkich danych wejściowych.

Do interpretacji w podsumowaniach zazwyczaj bierzemy średnią, a nie sumę, i pierwiastki, a nie kwadraty. Ale w przypadku optymalizacji nie jest to potrzebne, możemy więc oszczędzić odrobinę mocy obliczeniowej, ograniczając się do zliczania sumy kwadratów błędów.

Ostatnie dwa kroki są ze sobą ściśle powiązane. Gdybyśmy nie sumowali błędów, nie trzeba by ich było podnosić do kwadratu. Potęgowanie jest tu niezbędne właśnie z powodu sumowania — bez podnoszenia błędów do kwadratu ich suma dałaby zero.

Udowodnienie zerowania sumy błędów nie jest trudne, ale wymaga odwołania się do algebry, której w tej książce staramy się unikać.

Zobaczmy, jak można zaimplementować ten algorytm:

```
squared.error <- function(heights.weights, a, b)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(errors ^ 2))
}
```

Obliczmy teraz wartości funkcji `squared.error` dla pewnych konkretnych wartości `a` i `b` — zobaczymy, jak to działa w praktyce (wyniki eksperymentu podsumowuje tabela 7.1):

```
for (a in seq(-1, 1, by = 1))
{
```


Kup książkę Poleć książkę

```
  for (b in seq(-1, 1, by = 1))
  {
    print(squared.error(heights.weights, a, b))
  }
}
```

Tabela 7.1. Kwadraty błędu dla parametrów a i b w zakresie –1,1

a	b	Kwadrat błędu
–1	–1	536271759
–1	0	274177183
–1	1	100471706
0	–1	531705601
0	0	270938376
0	1	98560250
1	–1	527159442
1	0	267719569
1	1	96668794

Jak widać, dla niektórych kombinacji wartości `a` i `b` przyjęta miara błędu ma znacznie mniejsze wartości niż dla innych kombinacji. Oznacza to, że nasza funkcja błędu faktycznie niesie jakąś informację o jakości predykcji, i chcemy teraz znaleźć najlepsze wartości parametrów `a` i `b`. To pierwsza część zadania optymalizacyjnego — ustalenie miary, którą będziemy minimalizować albo maksymalizować. Miara ta jest zazwyczaj nazywana **funkcją celu** (ang. *objective function*). Problem optymalizacji staje się więc problemem znalezienia takich wartości `a` i `b`, które dają największą albo najmniejszą wartość funkcji celu.

Oczywistym sposobem poszukiwania optymalnych wartości jest metoda przeszukiwania tzw. „kraty" (ang. *grid search*). Należy wygenerować kratę wartości `a` i `b` podobną do tej z tabeli 7.1 dla odpowiednio dużego zakresu wartości `a` i `b`, a potem wybrać z niej wiersz o najmniejszej wartości kwadratu błędu. Metoda ta pozwala każdorazowo znaleźć najlepsze możliwe wartości parametrów, wydaje się więc odpowiednia. Niestety, ma istotne wady:

- Jak dobrać zakresy i odległości pomiędzy wartościami poszczególnych parametrów kraty? Czy `a` powinno mieć wartości 0, 1, 2 i 3? A może odpowiedniejsze będą wartości 0, 0,001, 0,002 i 0,003? Innymi słowy, jaki zakres i z jaką rozdzielczością chcemy przeszukać? Odpowiedź na to pytanie wymaga przeliczenia obu wariantów kraty i sprawdzenia, która z nich daje więcej informacji. Ale takie podejście jest kosztowne obliczeniowo, więc zasadniczo wprowadza kolejny problem optymalizacji, sprowadzający się do optymalizowania rozmiaru kraty. A tu zaczyna się szaleństwo nieskończonej rekurencji.
- Jeśli chcemy przeliczyć kratę dla 10 wartości dla każdego z dwóch parametrów, musimy zbudować tabelę o 100 elementach. Ale dla 100 parametrów będzie to już tabela o rozmiarze 10^{100} elementów. Problem wykładniczego wzrostu złożoności obliczeniowej jest tak powszechny w uczeniu maszynowym, że zyskał tu niesławne miano przekleństwa wielowymiarowości.

Ponieważ chcemy mieć możliwość używania regresji liniowej z setkami, a może i tysiącami wejść, przeszukiwanie kraty rozumiane jako algorytm optymalizacji uznamy za skreślone. Jak więc postąpić? Na szczęście informatycy i matematycy badali problem optymalizacji od

Kup książkę Poleć książkę

dawna i dopracowali się sporego zbioru algorytmów gotowych do łatwego użycia. W języku R pierwsze podejście do problemu optymalizacji sprowadza się do użycia funkcji `optim`, będącej czarną skrzynką implementującą wiele popularnych algorytmów optymalizacji.

Aby zademonstrować działanie funkcji `optim`, spróbujemy wykorzystać ją do dopasowania naszego modelu regresji liniowej — z nieśmiałą nadzieją, że otrzymane wyniki będą podobne do wyników uzyskanych z funkcji `lm`:

```
optim(c(0, 0),
      function (x)
      {
        squared.error(heights.weights, x[1], x[2])
      })
#$par
#[1] -350.786736    7.718158
#
#$value
#[1] 1492936
#
#$counts
#function gradient
#     111       NA
#
#$convergence
#[1] 0
#
#$message
#NULL
```

Zgodnie z przytoczonym przykładem funkcja `optim` przyjmuje kilka różnych parametrów. Jako pierwszy należy przekazać wektor liczbowy punktów startowych dla optymalizowanych parametrów. W tym przypadku mówimy, że początkowe wartości parametrów `a` i `b` odpowiadają wektorowi `c(0,0)`. Kolejnym parametrem funkcji `optim` powinna być funkcja przyjmująca wektor (u nas nazywany `x`) zawierający komplet parametrów do zoptymalizowania. Ponieważ nasza funkcja celu przyjmuje dwa parametry, ujmujemy jej wywołanie w funkcji anonimowej przyjmującej wektor `x` i wyłuskującej z niego parametry dla wywołania właściwej funkcji celu — `squared.error`.

Wykonanie powyższego wywołania funkcji `optim` zwraca wartości dla parametrów `a` `b` jako wartości kolumny `par`. Jak widać, wartości te są bardzo bliskie wartościom obliczanym przez `lm`, co sugeruje, że funkcja `optim` faktycznie działa[1]. W praktyce `lm` używa algorytmu optymalizacji mocno specjalizowanego dla regresji liniowej, dzięki czemu otrzymuje wyniki precyzyjniejsze niż wyniki z optymalizacji funkcją `optim`. Za to funkcja `optim` dobrze sprawdza się we własnych implementacjach, bo nadaje się do stosowania również w modelach innych niż regresja liniowa.

Pozostałe wartości wyliczane przez `optim` są najczęściej mniej interesujące. Pierwsza z nich to `value`, reprezentująca wartość funkcji celu (kwadratu błędu) dla zoptymalizowanych wartości parametrów. Dalej mamy `counts`, czyli licznik wykonań głównej funkcji (`function`), oraz gradient (`gradient`), będący opcjonalnym argumentem optymalizacji do stosowania, kiedy zna się matematykę wystarczająco dobrze, aby obliczyć gradient funkcji głównej.

[1] A przynajmniej, że działa tak samo dobrze (albo tak samo źle) jak `lm`.

Kup książkę Poleć książkę

Nie przejmujmy się, jeśli pojęcie gradientu jest nam zupełnie obce. Ręczne obliczanie gradientów jest żmudne, więc zazwyczaj zdajemy się na funkcję `optim` bez dookreślania opcjonalnej wartości gradientu. Jak dotąd, ta strusia strategia sprawdzała się nie najgorzej.

Następna wartość to konwergencja (`convergence`), mówiąca o tym, czy funkcja `optim` ma pewność, że znalazła faktycznie najlepszy możliwy zestaw parametrów. Jeśli wszystko poszło dobrze, wartość ta będzie równa 0. Jeśli konwergencja jest różna od zera, należy sprawdzić przyczynę niepełnej optymalizacji w dokumentacji funkcji `optim` — `convergence` będzie wtedy kodem błędu. Wreszcie wartość `message` zawiera komunikat z ewentualnymi dodatkowymi istotnymi informacjami o przebiegu optymalizacji.

Zasadniczo funkcja `optim` implementuje wiele sprytnych pomysłów obliczeniowych, pozwalających na przeprowadzenie optymalizacji. Ponieważ matematycznie jest ona dość zaawansowana, nie będziemy zagłębiać się w tajniki jej wewnętrznego działania. Natomiast ogólna zasada działania funkcji `optim` daje się bardzo łatwo zilustrować graficznie. Wyobraźmy sobie, że chcemy znaleźć najlepszą wartość `a` przy `b` ustalonym na 0. Kwadrat błędu dla wyizolowanej zmiennej `a` można wtedy obliczyć następująco:

```
a.error <- function(a)
{
  return(squared.error(heights.weights, a, 0))
}
```

Wykres kwadratu błędu w zależności od wartości `a` pozwala łatwo znaleźć miejsce najlepszej wartości `a`; wystarczy użyć funkcji `curve` języka R, która obliczy funkcję albo wyrażenie dla wielu wartości zmiennej, a potem wyrysuje wyniki. W poniższym przykładzie szacujemy `a.error` dla wielu wartości `x`. Z powodu zawiłości obliczania wartości wyrażeń w języku R musimy wykorzystać do tego funkcję `sapply`:

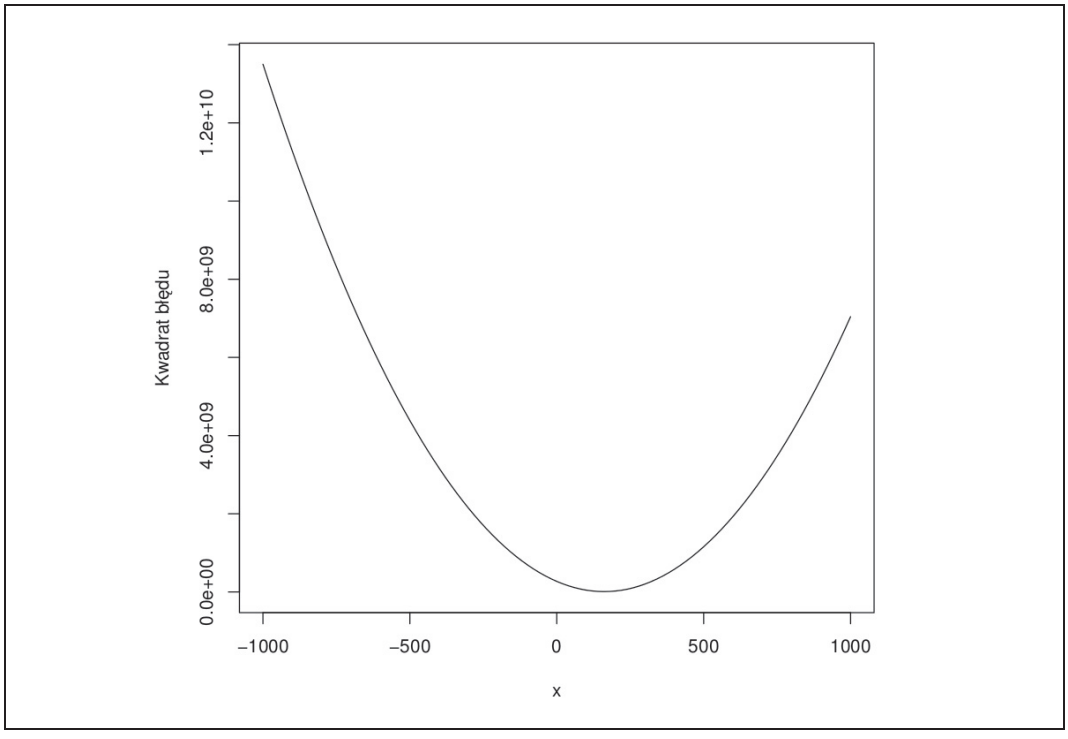
```
curve(sapply(x, function (a) {a.error(a)}), from = -1000, to = 1000)
```

Na rysunku 7.1 można wskazać miejsce, w którym wartość `a` jest optymalna z punktu widzenia funkcji celu. W miarę oddalania się od tego miejsca błąd rośnie. W takim przypadku mówimy o istnieniu **globalnego optimum**. Funkcja `optim` na podstawie kształtu funkcji celu może wówczas ustalić, w którym kierunku przesuwać sprawdzian po obliczeniu funkcji celu dla jednej wartości `a`. Użycie lokalnej informacji do pozyskania wiedzy o globalnej strukturze problemu pozwala funkcji `optim` bardzo szybko znaleźć punkt optimum.

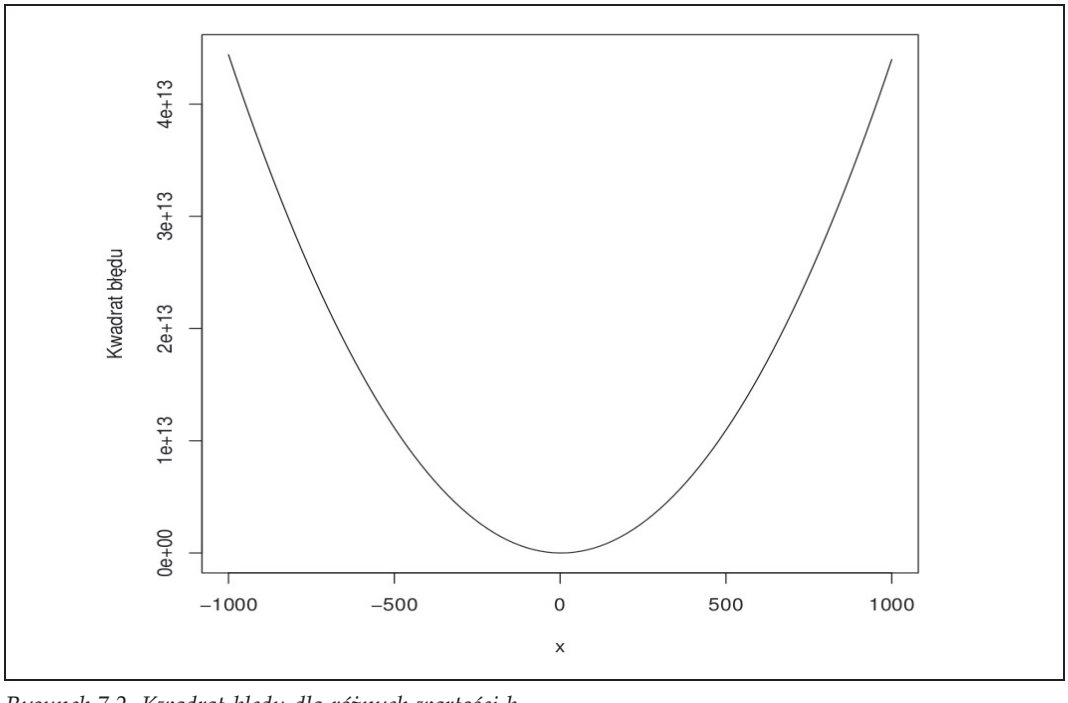
Aby odnieść to do pełnego problemu regresji, powinniśmy również zbadać zmienność funkcji kwadratu błędu w zależności od zmian parametru `b` (patrz rysunek 7.2):

```
b.error <- function(b)
{
  return(squared.error(heights.weights, 0, b))
}
curve(sapply(x, function (b) {b.error(b)}), from = -1000, to = 1000)
```


Kup książkę Poleć książkę



Rysunek 7.1. kwadrat błędu dla różnych wartości a



Rysunek 7.2. Kwadrat błędu dla różnych wartości b

Kup książkę Poleć książkę

Z rysunku 7.2 wiemy, że dla parametru `b` funkcja błędu również posiada globalne optimum. Skoro istnieje lokalne optimum dla parametrów `a` i `b` rozpatrywanych osobno, wydaje się, że funkcja `optim` powinna poradzić sobie ze znalezieniem jednego zestawu wartości `a` i `b`, dla których funkcja błędu ma najmniejszą wartość.

Ogólniej można powiedzieć, że funkcja `optim` działa, ponieważ może przeszukiwać przestrzeń funkcji błędu dla wszystkich parametrów naraz. Działa szybciej niż przeszukiwanie kraty, bo wykorzystuje informacje o obecnie rozważanym punkcie do wnioskowania o punktach sąsiednich, co pozwala na podjęcie decyzji o kierunku dalszych poszukiwań. Ta jej zdolność do adaptacji sprawia, że jest znacznie wydajniejsza od łopatologicznego przeszukiwania kraty kombinacji parametrów.

Regresja grzbietowa

Skoro wiemy już, jak używać funkcji `optim`, możemy przystąpić do stosowania algorytmów optymalizacji do implementowania własnej wersji regresji grzbietowej. **Regresja grzbietowa** (ang. *ridge regression*) to specyficzna forma regresji ujmująca regularyzację omawianą w rozdziale 6. Różnica pomiędzy klasyczną regresją najmniejszych kwadratów a regresją grzbietową sprowadza się do użytej funkcji błędu — regresja grzbietowa dąży do zmniejszania współczynników, uwzględniając ich wartości przy określaniu błędu. W tym przykładzie spowoduje to spychanie wartości przesunięcia i nachylenia w kierunku zera.

Poza zmianą funkcji błędu regresja grzbietowa wprowadza też narzut złożoności obliczeniowej, wprowadzając dodatkowy parametr — `lambda` — balansujący wagę przykładaną do minimalizowania kwadratu błędu i wagę przykładaną do minimalizowania wartości `a` i `b` (co pozwala na uniknięcie nadmiernego dopasowania). Dodatkowy parametr tego regularyzowanego algorytmu nosi nazwę hiperparametru, omawianego już w pewnym zakresie w rozdziale 6. Dla ustalonej wartości lambda funkcję błędu regresji grzbietowej możemy zapisać następująco:

```
ridge.error <- function(heights.weights, a, b, lambda)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(errors ^ 2) + lambda * (a ^ 2 + b ^ 2))
}
```

Zgodnie z omówieniem z rozdziału 6., wartość lambdy dobiera się za pomocą sprawdzianu krzyżowego. Na potrzeby tego rozdziału założymy po prostu, że sprawdzian ten został już przeprowadzony i odpowiednią wartością parametru `lambda` będzie `1`.

Po zdefiniowaniu funkcji błędu regresji grzbietowej jako funkcji języka R rozwiązanie zadania regresji grzbietowej sprowadza się znów do wywołania `optim` — równie łatwo, jak w przypadku regresji z kwadratem błędu:

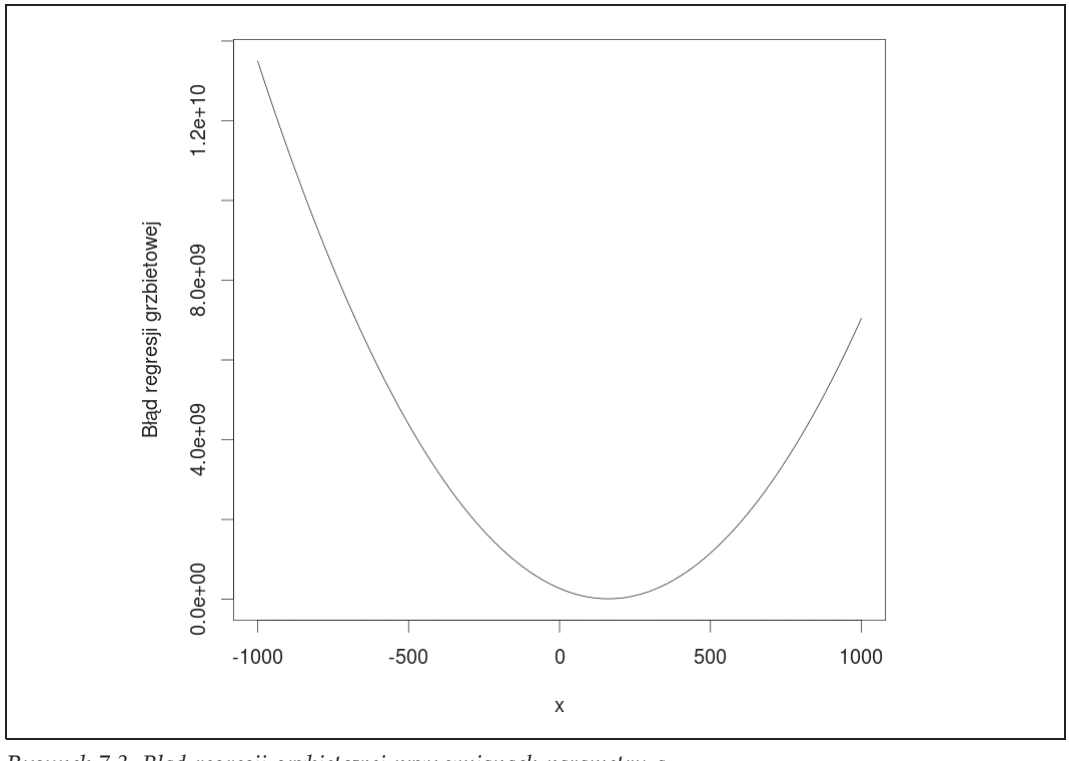
```
lambda <- 1

optim(c(0, 0),
      function (x)
      {
        ridge.error(heights.weights, x[1], x[2], lambda)
      })
#$par
#[1] -340.434108    7.562524
#
```


Kup książkę Poleć książkę

```
#$value
#[1] 1612443
#
#$counts
#function gradient
#     115       NA
#
#$convergence
#[1] 0
#
#$message
#NULL
```

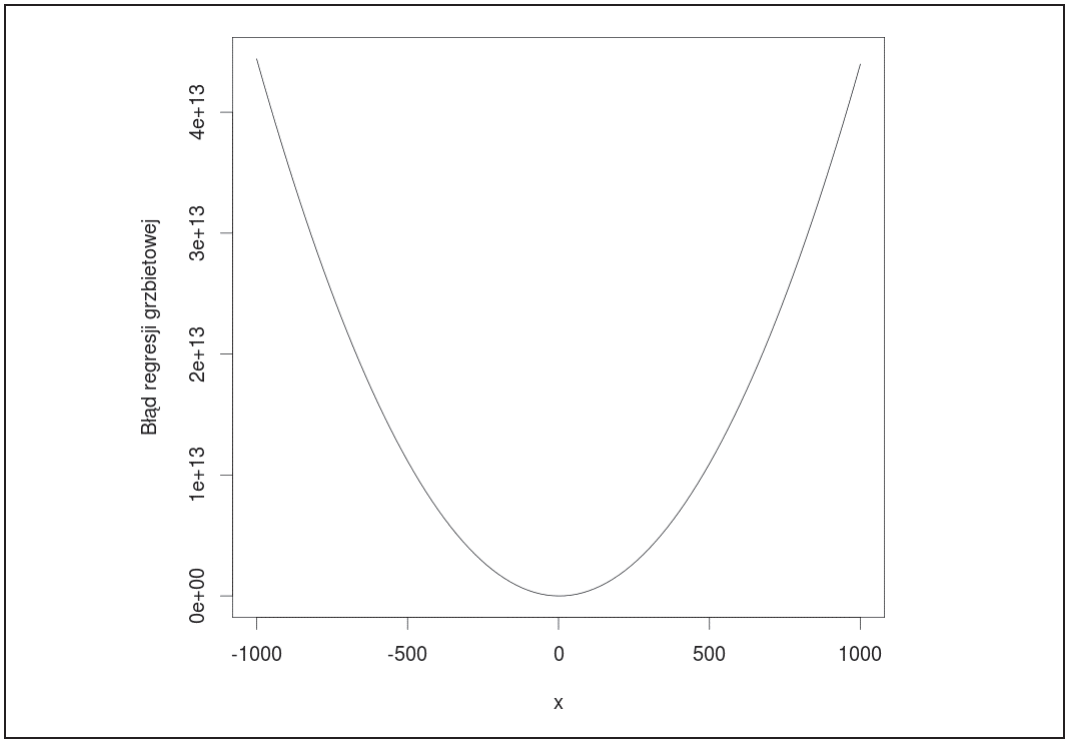
Wynik funkcji `optim` pokazuje, że możemy w ten sposób wygenerować nieco mniejsze przesunięcie i nachylenie niż w przypadku regresji liniowej funkcją `lm`, gdzie przesunięcie wynosiło –350, a nachylenie 7,7. W tym uproszczonym przykładzie nie jest to szczególnie przydatne, ale w regresjach o większej skali, takich jak regresje z rozdziału 6., penalizowanie dużych wartości współczynników okazało się zbawienne dla jakości algorytmu.

Oprócz oglądania dopasowanych współczynników regresji możemy też powtórzyć wywołania funkcji `curve` w celu sprawdzenia, dlaczego funkcja `optim` radzi sobie z regresją grzbietową równie dobrze, jak ze zwyczajną regresją liniową. Odpowiednie wykresy prezentowane są na rysunkach 7.3 i 7.4.



Rysunek 7.3. Błąd regresji grzbietowej przy zmianach parametru a

Kup książkę Poleć książkę



Rysunek 7.4. Błąd regresji grzbietowej przy zmianach parametru b

```
a.ridge.error <- function(a, lambda)
{
  return(ridge.error(heights.weights, a, 0, lambda))
}

curve(sapply(x, function (a) {a.ridge.error(a, lambda)}), from = -1000, to = 1000)

b.ridge.error <- function(b, lambda)
{
  return(ridge.error(heights.weights, 0, b, lambda))
}

curve(sapply(x, function (b) {b.ridge.error(b, lambda)}), from = -1000, to = 1000)
```

Przykład ten pokazuje, że w uczeniu maszynowym możemy wiele zdziałać samą tylko umiejętnością stosowania funkcji takich jak `optim` w celu minimalizowania pewnej miary błędu predykcji. Zalecamy samodzielne przepracowanie kilku przykładów i eksperymentowanie z różnymi własnymi funkcjami błędu. Umiejętność posługiwania się funkcją `optim` i interpretowania jej wyników jest przydatna zwłaszcza przy testowaniu funkcji wartości bezwzględnej błędu, jak tutaj:

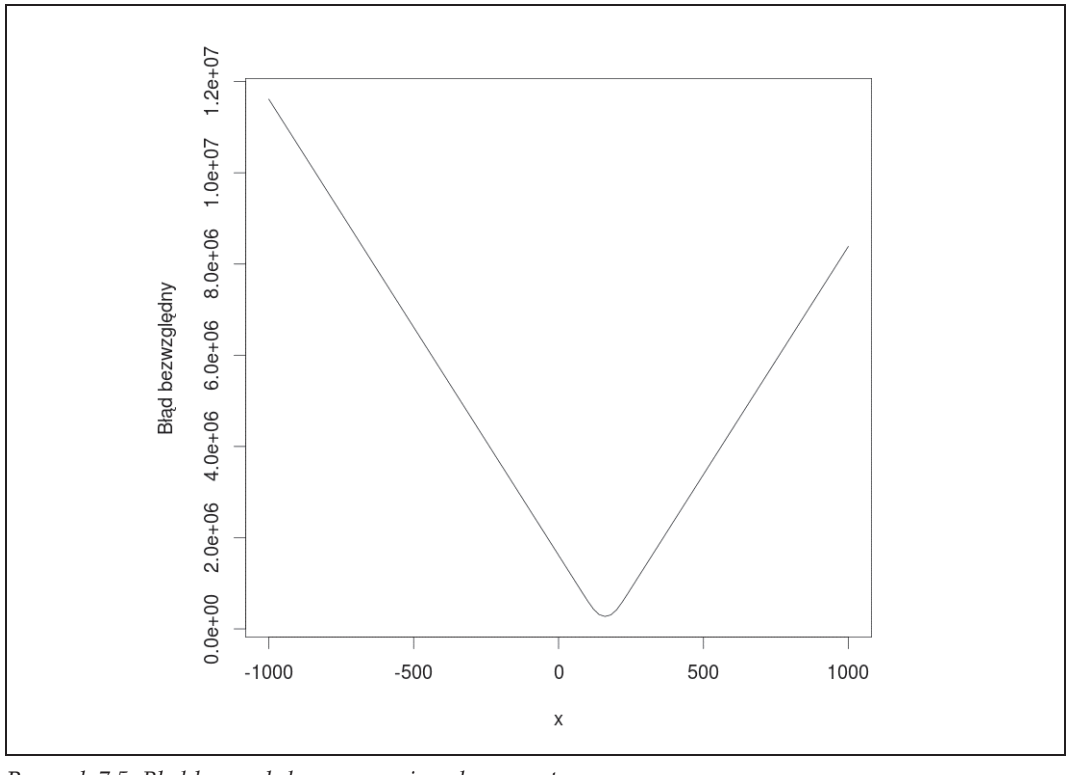
```
absolute.error <- function (heights.weights, a, b)
{
  predictions <- with(heights.weights, height.to.weight(Height, a, b))
  errors <- with(heights.weights, Weight - predictions)
  return(sum(abs(errors)))
}
```


Kup książkę Poleć książkę

Z uwagi na techniczne aspekty rachunku różniczkowego taka miara błędu nie da dobrych wyników z funkcją `optim`. Wyjaśnienie tego bez wykonywania zaawansowanych obliczeń jest nieco utrudnione, ale da się to wytłumaczyć na podstawie wizualizacji. Powtórzmy więc wywołanie funkcji rysującej wykres:

```
a.absolute.error <- function(a)
{
  return(absolute.error(heights.weights, a, 0))
}

curve(sapply(x, function (a) {a.absolute.error(a)}), from = -1000, to = 1000)
```

Na rysunku 7.5 widzimy znacznie ostrzejszy przebieg krzywej błędu bezwzględnego w porównaniu z kwadratem błędu i błędem regresji grzbietowej. Ponieważ kształt krzywej jest tak ostry, funkcja `optim` nie uzyskuje wystarczającej ilości informacji o pożądanym kierunku dalszych poszukiwań i niekoniecznie dotrze do globalnego optimum, mimo że na wykresie punkt optimum widać jak na dłoni.



Rysunek 7.5. Błąd bezwzględny przy zmianach parametru a

Skoro niektóre typy miar błędu załamują skuteczność nawet zaawansowanych algorytmów, elementem sztuki uczenia maszynowego jest poznanie warunków skutecznego działania narzędzi takich jak funkcja `optim` i rozpoznawanie sytuacji, w których te prostsze narzędzia przestają wystarczać. Istnieją przecież algorytmy działające również w przypadku optymalizacji z błędem bezwzględnym, choć ich omawianie wykracza poza zakres tej książki — zainteresowanych Czytelników pozostaje odesłać do lokalnego guru matematyki, aby pogawędzili o optymalizacji wypukłej (ang. *convex optimization*).

Kup książkę Poleć książkę

Łamanie szyfrów jako problem optymalizacji

Wychodząc poza modele regresji, praktycznie każdy algorytm w uczeniu maszynowym można postrzegać jako problem optymalizacji, w ramach którego chcemy zminimalizować pewną miarę błędu predykcji. Ale niekiedy nasze parametry nie są prostymi liczbami, przez co obliczenie funkcji błędu w jednym punkcie danych nie daje wystarczającej informacji o punktach sąsiednich, aby skutecznie użyć funkcji `optim`. W przypadku takich problemów możemy uciec się do przeszukiwania kraty, ale są też inne, efektywniejsze techniki. Skupimy się tu na jednej z nich, szczególnie intuicyjnej i dość efektywnej. Koncepcja stojąca za tym nowym podejściem, które będziemy nazywać **optymalizacją stochastyczną**, polega na przejściu przez zakres możliwych parametrów w pewien sposób losowo, ale z zapewnieniem utrzymywania kierunku, w którym funkcja błędu raczej maleje, niż wzrasta. Podejście to jest związane z wieloma popularnymi algorytmami optymalizacji, z symulowanym wyżarzaniem (ang. *simulated annealing*), algorytmami genetycznymi i markowowskim Monte Carlo (MCMC, od ang. *Markov Chain Monte Carlo*) na czele. Konkretny algorytm, którego użyjemy, nosi miano metody Metropolis. Rozmaite wersje tej metody stanowią podstawy wielu współczesnych algorytmów uczenia maszynowego.

Aby zilustrować metodę Metropolis, jako studium przypadku dla tego rozdziału wybraliśmy problem łamania tajnych szyfrów. Algorytm, który będziemy tu definiować, nie jest bynajmniej szczególnie efektywnym systemem deszyfrującym i nie mógłby być brany na poważnie w rozwiązaniach produkcyjnych, ale stanowi dobitny przykład zastosowania metody Metropolis. Co ważne, jest to też przykład problemu, z którym gotowe algorytmy optymalizacji w rodzaju funkcji `optim` w ogóle sobie nie radzą.

Określmy więc zadanie — dla danego ciągu liter, o którym wiemy, że jest szyfrogramem wytworzonym przez szyfr podstawieniowy, należy zdecydować o przyjęciu reguły deszyfrującej, pozwalającej na odtworzenie jawnego tekstu. Dla osób nieznających szyfrów podstawieniowych wypada przypomnieć, że są to najprostsze możliwe systemy szyfrowania, w których każda litera jawnego tekstu jest zamieniana na odpowiadającą jej literę szyfrogramu (zawsze taką samą). Chyba najbardziej znanym przykładem takiego szyfru jest ROT13[2]; innym słynnym szyfrem z tej rodziny jest szyfr Cezara. W szyfrze Cezara każda litera jest zamieniana na następną literę alfabetu: „a" zamienia się na „b", „b" na „c", a „c" na „d" (w przypadku brzegowym: „z" zamienia się na „a").

Aby przećwiczyć stosowanie prostych szyfrów w R, napiszemy w tym języku implementację szyfru Cezara:

```
english.letters <- c('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                     'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v',
                     'w', 'x', 'y', 'z')

caesar.cipher <- list()
inverse.caesar.cipher <- list()

for (index in 1:length(english.letters))
{
  caesar.cipher[[english.letters[index]]] <- english.letters[index %% 26 + 1]
  inverse.caesar.cipher[[english.letters[index %% 26 + 1]]] <- english.letters[index]
}

print(caesar.cipher)
```

[2] Szyfr ROT13 zamienia każdą literę na literę odległą o 13 pozycji w alfabecie: „a" zamienia się na „n", „b" na „o" itd.

Kup książkę Poleć książkę

Po zaimplementowaniu szyfru możemy przystąpić do programowania funkcji pomocniczych, które pozwolą na zaszyfrowanie ciągu znaków za pomocą przekazanego szyfru:

```
apply.cipher.to.string <- function(string, cipher)
{
  output <- ''
  for (i in 1:nchar(string))
  {
    output <- paste(output, cipher[[substr(string, i, i)]], sep = '')
  }
  return(output)
}

apply.cipher.to.text <- function(text, cipher)
{
  output <- c()
  for (string in text)
  {
    output <- c(output, apply.cipher.to.string(string, cipher))
  }
  return(output)
}

apply.cipher.to.text(c('sample', 'text'), caesar.cipher)
```

Wypracowaliśmy sobie podstawowe narzędzia do pracy z szyframi, możemy więc zacząć zastanawiać się nad problemem łamania szyfrów. Podobnie jak w regresji liniowej, problem złamania podstawienia szyfrującego będziemy rozwiązywać etapami:

1. Zdefiniuj miarę jakości proponowanej reguły deszyfrującej.
2. Zdefiniuj algorytm proponowania nowych potencjalnych reguł deszyfrujących, który losowo modyfikuje warianty obecnie najlepszej reguły.
3. Zdefiniuj algorytm przesuwania się w kierunku coraz lepszych reguł deszyfrujących.

W ramach przymiarki do mierzenia jakości reguły deszyfrującej załóżmy, że mamy dany fragment tekstu, o którym wiemy, że został zaszyfrowany za pomocą szyfru podstawieniowego. Na przykład sam Juliusz Cezar przysłał nam wiadomość o treści „wfoj wjej wjdj". Po odwróceniu szyfru Cezara tekst ten zamieni się w słynne zdanie „veni vidi vici".

Wyobraźmy sobie, że mamy do dyspozycji trochę zaszyfrowanego tekstu i zapewnienie, że tekst jawny szyfrogramu był tekstem w języku angielskim. Jak podeszlibyśmy do złamania szyfrogramu? Otóż powiedzielibyśmy, że reguła deszyfrująca jest dobra, jeśli zamienia szyfrogram na tekst w języku angielskim. Dla proponowanej reguły deszyfrującej należałoby zastosować ją do szyfrogramu i sprawdzić, czy otrzymany tekst jest faktycznie tekstem w języku angielskim. Na przykład dane są dwie proponowane reguły deszyfrujące A i B, które dają następujące wyniki:

- `decrypt(T, A)` = xgpk xkfk xkek
- decrypt(T, B) = veni vidi vici

Po obejrzeniu wyników zastosowania obu proponowanych reguł wydaje się oczywiste, że reguła B jest lepsza niż reguła A. Jak doszliśmy do tego intuicyjnego wniosku? Widzimy, że reguła B jest lepsza od A, bo tekst wynikowy z zastosowania reguły B przypomina prawdziwy język, podczas gdy wynik zastosowania reguły A wygląda jak kompletny bełkot. Aby przełożyć ludzką intuicję na coś automatycznego, co da się zaprogramować na komputerze,

Kup książkę Poleć książkę

będziemy musieli skorzystać z leksykalnej bazy danych, określającej prawdopodobieństwo występowania poszczególnych słów. Jako prawdziwy język wytypujemy wtedy taki tekst, który będzie się składał z wyrazów mających wysokie prawdopodobieństwo występowania. Bełkot byłby tu odpowiednikiem tekstu, który składa się ze słów o niskim prawdopodobieństwie występowania. Jedyna trudność w tym modelu polega na radzeniu sobie ze słowami, które w ogóle nie istnieją. Ponieważ prawdopodobieństwo ich wystąpienia wynosi zero, a prawdopodobieństwo przynależności całego tekstu do prawdziwego języka będziemy oceniać jako iloczyn prawdopodobieństwa wystąpienia poszczególnych słów, powinniśmy zastąpić zero jakąś bardzo małą wartością — na przykład minimalną wartością zmiennoprzecinkową, którą nazwiemy epsilon. Gdy załatwimy w ten sposób przypadki brzegowe, możemy użyć leksykalnej bazy danych do ułożenia rankingu jakości dwóch rozszyfrowanych tekstów poprzez wyznaczenie prawdopodobieństwa wystąpienia każdego ze słów i przez wyznaczenie iloczynu tych prawdopodobieństw jako łącznego prawdopodobieństwa przynależności tekstu do języka.

Zastosowanie leksykalnej bazy danych do obliczania prawdopodobieństwa wystąpienia odszyfrowanego tekstu będzie naszą miarą błędu przy ocenianiu proponowanej reguły deszyfrującej. A skoro mamy już pomysł na funkcję błędu, nasz problem faktycznie przekształca się całkowicie w problem optymalizacji. Teraz wystarczy więc znaleźć reguły deszyfrujące, generujące teksty o wysokim prawdopodobieństwie wystąpienia w założonym języku.

Niestety, zadanie wyszukiwania reguł o najwyższym prawdopodobieństwie wystąpienia tekstu nie jest nawet w przybliżeniu zadaniem, z którym poradzi sobie funkcja `optim`. Z reguł deszyfrujących nie da się zrobić wykresu; nie mają też gładkości potrzebnej funkcji `optim` do określania kierunku poszukiwań coraz lepszych reguł. Potrzebujemy więc zupełnie innego algorytmu optymalizacji. Będzie nim zapowiadana już metoda Metropolis. Algorytm ten powinien się dość dobrze sprawiać w naszym problemie, chociaż jest algorytmem powolnym (przynajmniej dla jakichkolwiek dłuższych tekstów)[3].

Podstawowa koncepcja metody Metropolis polega na tym, że zaczynamy od arbitralnie przyjętej reguły deszyfrującej i potem iteracyjnie próbujemy ją polepszać — do momentu, w którym uznamy, że to może być właściwa reguła. Wydaje się, że brzmi to nieco magicznie, ale w praktyce często działa zupełnie dobrze, o czym przekonamy się doświadczalnie. A po uzyskaniu stosunkowo dobrej reguły deszyfrującej możemy odwołać się do własnej intuicji w oparciu o ocenę spójności znaczeniowej i gramatycznej i w ten sposób ostatecznie zdecydować, czy tekst został poprawnie odszyfrowany.

Aby wygenerować dobrą regułę deszyfrującą, zaczniemy od zupełnie dowolnej reguły, a potem będziemy wielokrotnie — powiedzmy, 50 000 razy — powtarzać pojedynczą operację polepszającą tę regułę. Ponieważ każdy krok iteracji będzie wykonywany w kierunku coraz lepszych reguł, uparte powtarzanie tej operacji powinno wreszcie doprowadzić do sensownych wyników. Nie ma tylko gwarancji, że całość zamknie się w 50 000 iteracji, a nie na przykład w 50 000 000. Dlatego właśnie nie zastosowalibyśmy tego algorytmu do łamania szyfrów na poważnie — nie mamy gwarancji, że algorytm dostarczy rozwiązanie w skończonym i rozsądnym czasie, a oczekując na zakończenie, trudno często stwierdzić, czy całość posuwa się aby ku lepszemu. To studium przypadku można więc uznać za ćwiczenie czysto dydaktyczne. Jedynym jego założeniem jest pokazanie zastosowania algorytmów optymalizacji do skomplikowanych zadań, które z pozoru są problemami zupełnie innej klasy.

[3] Jego powolność jest silnie uwypuklona przez nieefektywność dostępnych w R narzędzi do przetwarzania tekstu, nieporównanie wolniejszych od podobnych narzędzi na przykład w języku Perl.

Kup książkę Poleć książkę

Skonkretyzujmy sposób proponowania nowej reguły deszyfrującej. Otóż będziemy losowo zmieniać bieżącą regułę w jednym punkcie. W każdym kroku będziemy więc modyfikować wpływ reguły deszyfrującej na pojedynczą literę alfabetu: jeśli „a" w obecnej regule zamienia się na „b", zaproponujemy regułę zmodyfikowaną, w której „a" zamienia się na „q". Z powodu zasady działania szyfru podstawieniowego zmiana ta będzie wymagała równoczesnej modyfikacji tej części reguły, która dotychczas zamieniała na „q" inną literę alfabetu, dajmy na to „c". Aby szyfr wciąż działał, „c" będzie w nowej regule zamieniane na „b".

Tak więc nasz algorytm proponowania nowych reguł deszyfrujących sprowadza się do wykonania dwóch zamian w istniejącej regule — jednej dobranej losowo i drugiej korygującej, wymuszonej przez zasadę działania szyfru podstawieniowego.

W naiwnym podejściu proponowaną nową regułę zaakceptowalibyśmy tylko wtedy, kiedy zwiększałaby prawdopodobieństwo odszyfrowania wiadomości. Taka optymalizacja nazywa się **optymalizacją zachłanną** albo **agresywną** (ang. *greedy optimization*). Niestety, zachłanna optymalizacja wprowadza ryzyko utknięcia na złych regułach, więc przy akceptowaniu nowej propozycji reguły deszyfrującej będziemy używali oceny niezachłannej:

1. Jeśli prawdopodobieństwo wystąpienia tekstu odszyfrowanego regułą B jest większe niż prawdopodobieństwo wystąpienia tekstu odszyfrowanego regułą A, reguła B zastępuje regułę A.
2. Jeśli prawdopodobieństwo wystąpienia tekstu odszyfrowanego regułą B jest mniejsze niż prawdopodobieństwo wystąpienia tekstu odszyfrowanego regułą A, to reguła B zastąpi regułę A, ale nie za każdym razem. Konkretnie mówiąc, jeśli prawdopodobieństwo skuteczności reguły B to `probability(T,B)`, a analogiczne prawdopodobieństwo dla reguły A to `probability(T,A)`, wybierzemy regułę B w `probability(T,B)/probability(T,A)` procencie przypadków.

Jeśli powyższy współczynnik dopuszczenia zastąpienia lepszej reguły przez słabszą regułę wydaje się arbitralny, nie szkodzi. Tak naprawdę nie liczy się konkretny współczynnik, ale sam fakt, że słabsza reguła ma jakąkolwiek szansę zastąpić regułę silniejszą. W ten sposób unikamy pułapki zatrzymania zachłannej optymalizacji na lokalnym optimum funkcji błędu.

Zanim będziemy mogli użyć metody Metropolis do łamania najróżniejszych szyfrów, potrzebujemy jeszcze kilku narzędzi do generowania i modyfikowania szyfrów. Oto one:

```
generate.random.cipher <- function()
{
  cipher <- list()

  inputs <- english.letters
  outputs <- english.letters[sample(1:length(english.letters),
    length(english.letters))]

  for (index in 1:length(english.letters))
  {
    cipher[[inputs[index]]] <- outputs[index]
  }

  return(cipher)
}

modify.cipher <- function(cipher, input, output)
```


Kup książkę Poleć książkę

```
{
  new.cipher <- cipher
  new.cipher[[input]] <- output
  old.output <- cipher[[input]]
  collateral.input <- names(which(sapply(names(cipher),
                            function (key) {cipher[[key]]}) == output))
  new.cipher[[collateral.input]] <- old.output
  return(new.cipher)
}

propose.modified.cipher <- function(cipher)
{
  input <- sample(names(cipher), 1)
  output <- sample(english.letters, 1)
  return(modify.cipher(cipher, input, output))
}
```

Połączenie narzędzia do proponowania nowych reguł i opisanej procedury zamiany łagodzi zachłanność metody optymalizacji bez narażania nas na marnowanie zbyt dużej ilości czasu na reguły wyraźnie słabe, które dają znacznie mniejsze (niż reguła bieżąca) prawdopodobieństwo odszyfrowania. Aby to złagodzenie zapisać algorytmicznie, obliczamy iloraz `probability(T,B)/probability(T,A)` i porównujemy wynik z liczbą losową z zakresu od 0 do 1. Jeśli otrzymana liczba losowa jest większa niż iloraz `probability(T,B)/probability(T,A)`, reguła proponowana zastępuje regułę bieżącą. Jeśli nie, pozostajemy przy regule bieżącej.

Na potrzeby wyznaczania prawdopodobieństw, na które się stale powołujemy, tworzymy leksykalną bazę danych mówiącą o częstości występowania słów zgromadzonych w pliku */usr/share/dic/words* w tekście z Wikipedii. Aby załadować taką bazę do programu w języku R, korzystamy z następującej instrukcji:

```
load('data/lexical_database.Rdata')
```

Bazę danych można przejrzeć, podglądając pojedyncze słowa i ich częstości wystąpień w przykładowym tekście (patrz tabela 7.2):

```
lexical.database[['a']]
lexical.database[['the']]
lexical.database[['he']]
lexical.database[['she']]
lexical.database[['data']]
```

Tabela 7.2. Leksykalna baza danych

Słowo	Prawdopodobieństwo wystąpienia
a	0,01617576
the	0,05278924
he	0,003205034
she	0,0007412179
data	0,0002168354

Po załadowaniu bazy danych potrzebujemy jeszcze metod obliczania prawdopodobieństwa wystąpienia tekstu. Najpierw więc piszemy funkcję ujmującą zadanie wyciągnięcia prawdopodobieństwa z bazy danych. Rolą tej funkcji będzie też ułatwienie obsługi fikcyjnych słów, którym funkcja będzie przypisywać najmniejsze możliwe prawdopodobieństwo — równe najmniejszemu epsilonowi reprezentacji zmiennoprzecinkowej dla danego typu komputera. Wartość ta w języku R jest dostępna jako zmienna `.Machine$double.eps`:

Kup książkę Poleć książkę

```
one.gram.probability <- function(one.gram, lexical.database = list())
{
  lexical.probability <- lexical.database[[one.gram]]

  if (is.null(lexical.probability) || is.na(lexical.probability))
  {
  return(.Machine$double.eps)
  }
  else
  {
  return(lexical.probability)
  }
}
```

Zaimplementowaliśmy metodę wyszukiwania prawdopodobieństwa pojedynczych słów — potrzebujemy jeszcze metody obliczania łącznego prawdopodobieństwa występowania tekstu w języku angielskim. Metoda ta dzieli tekst na słowa, pozyskuje prawdopodobieństwo pojedynczych słów, a potem oblicza prawdopodobieństwo wynikowe jako iloczyn prawdopodobieństw jednostkowych. Niestety, okazuje się, że stosowanie surowych wartości prawdopodobieństwa jest niestabilne obliczeniowo (z powodu ograniczonej precyzji obliczeń na operandach zmiennoprzecinkowych przy mnożeniu). Z tego powodu postanowiliśmy obliczać zlogarytmowane prawdopodobieństwo występowania tekstu w języku angielskim, będące sumą zlogarytmowanych prawdopodobieństw występowania poszczególnych słów. Tak obliczona miara okazała się stabilna obliczeniowo.

```
log.probability.of.text <- function(text, cipher, lexical.database = list())
{
  log.probability <- 0.0

  for (string in text)
  {
    decrypted.string <- apply.cipher.to.string(string, cipher)
    log.probability <- log.probability +
      log(one.gram.probability(decrypted.string, lexical.database))
  }

  return(log.probability)
}
```

Skoro mamy już wszystkie komponenty wykonawcze, możemy zdefiniować krok metody Metropolis:

```
metropolis.step <- function(text, cipher, lexical.database = list())
{
  proposed.cipher <- propose.modified.cipher(cipher)

  lp1 <- log.probability.of.text(text, cipher, lexical.database)
  lp2 <- log.probability.of.text(text, proposed.cipher, lexical.database)

  if (lp2 > lp1)
  {
    return(proposed.cipher)
  }
  else
  {
    a <- exp(lp2 - lp1)
    x <- runif(1)
    if (x < a)
    {
      return(proposed.cipher)
```


Kup książkę Poleć książkę

```
    }
    else
    {
      return(cipher)
    }
  }
}
```

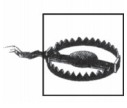
Możemy już przystąpić do połączenia poszczególnych etapów algorytmu optymalizacji — napiszemy prosty program i sprawdzimy jego działanie. Na początek potrzebujemy jakiegoś tekstu wzorcowego, który zapiszemy w R jako wektor znakowy:

```
decrypted.text <- c('here', 'is', 'some', 'sample', 'text')
```

Tekst ten zaszyfrujemy szyfrem Cezara:

```
encrypted.text <- apply.cipher.to.text(decrypted.text, caesar.cipher)
```

Teraz wygenerujemy losowy szyfr dekodujący, rozpoczniemy od niego 50 000 iteracji metody Metropolis i zapiszemy wyniki poszczególnych iteracji w ramce danych `results`. W każdym przebiegu będziemy rejestrować logarytm prawdopodobieństwa dla odszyfrowanego tekstu, odszyfrowany w danej iteracji tekst oraz sztuczną zmienną, sygnalizującą, czy tekst został odszyfrowany poprawnie.



Gdybyśmy naprawdę chcieli złamać tajny szyfr, nie bylibyśmy w stanie oznaczyć odszyfrowanych wiadomości jako prawdziwe. Tu robimy to jedynie w ramach przykładu.

```
set.seed(1)
cipher <- generate.random.cipher()

results <- data.frame()

number.of.iterations <- 50000

for (iteration in 1:number.of.iterations)
{
  log.probability <- log.probability.of.text(encrypted.text, cipher, lexical.database)
  current.decrypted.text <- paste(apply.cipher.to.text(encrypted.text, cipher),
                                  collapse = ' ')

  correct.text <- as.numeric(current.decrypted.text == paste(decrypted.text,
                                                             collapse = ' '))
  results <- rbind(results,
                   data.frame(Iteration = iteration,
                              LogProbability = log.probability,
                              CurrentDecryptedText = current.decrypted.text,
                              CorrectText = correct.text))
  cipher <- metropolis.step(encrypted.text, cipher, lexical.database)
}

write.table(results, file = file.path('data/results.tsv'),
            row.names = FALSE, sep = '\t')
```

Wykonanie programu zajmie dobrą chwilę, więc podczas oczekiwania na wyniki zapraszamy do spojrzenia na próbkę gotowych wyników, zebranych w tabeli 7.3.

Kup książkę Poleć książkę

Tabela 7.3. Postępy metody Metropolis

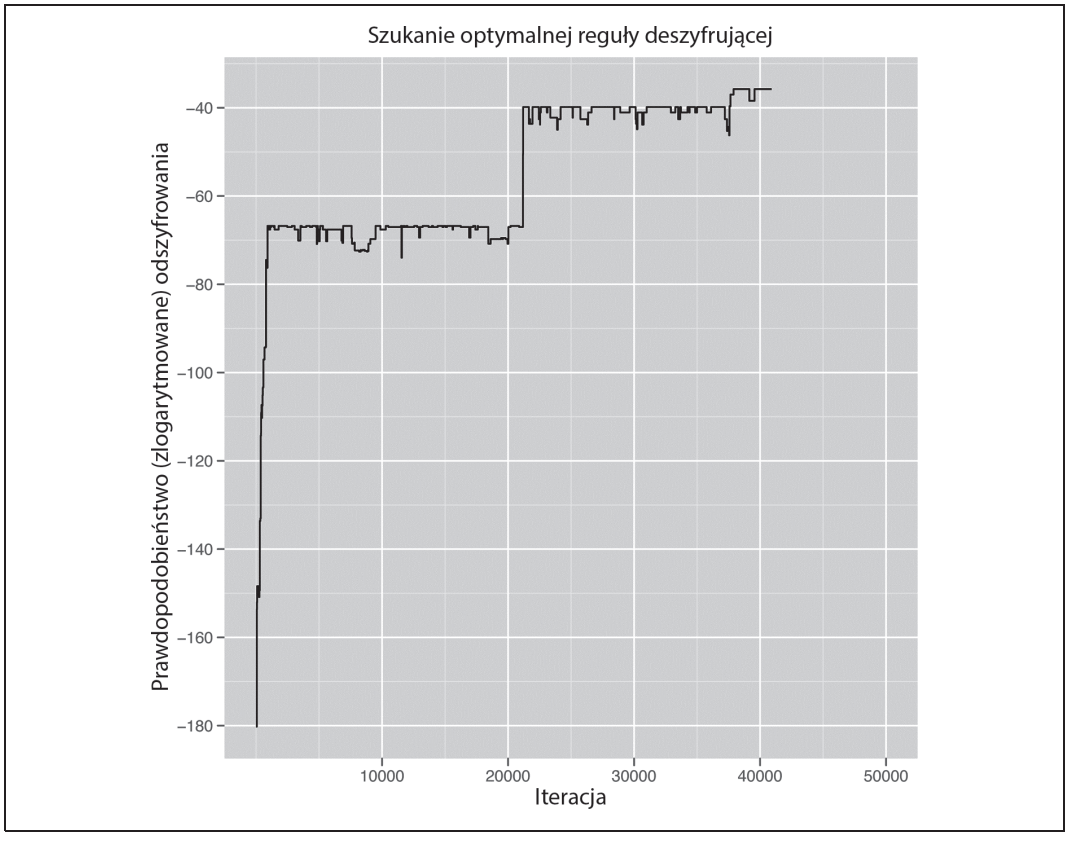
Iteracja	Logarytm prawdopodobieństwa	Odszyfrowany tekst
1	-180.218266945586	lsps bk kfvs kjvhys zsrz
5000	-67.6077693543898	gene is same sfmpwe text
10000	-67.6077693543898	gene is same spmzoe text
15000	-67.6077693543898	gene is some scmhbe text
20000	-70.8114316132189	dene as some scmire text
25000	-39.8590155606438	gene as some simple text
30000	-39.8590155606438	gene as some simple text
35000	-39.8590155606438	gene as some simple text
40000	-35.784429416419	were as some simple text
45000	-37.0128944882928	were is some sample text
50000	-35.784429416419	were as some simple text

Jak widać, po 45 000 krokach metody Metropolis byliśmy już bardzo blisko poprawnego odszyfrowania tekstu. Szczegółowa analiza wszystkich wyników pokazuje, że poprawny tekst został uzyskany w iteracji numer 45 609, ale algorytm poszedł dalej, zamieniając regułę w pełni poprawną na inną zaproponowaną. Jest to skutek ograniczenia naszej funkcji celu, która nie ocenia, czy odszyfrowany tekst jest poprawnym tekstem języka angielskiego, lecz jedynie sprawdza, czy poszczególne wyrazy są poprawnymi wyrazami. Jeśli zmiana reguły pozwala na odszyfrowanie bardziej prawdopodobnego słowa, algorytm pójdzie w tym kierunku, nawet jeśli wynikowy tekst będzie koszmarnie niegramatyczny albo nielogiczny. Moglibyśmy użyć do oceny dodatkowych miar, na przykład prawdopodobieństwa występowania par słów. Na razie jednak zostawimy algorytm bez zmian, podkreślając na tym przykładzie skomplikowanie metod optymalizacyjnych opartych na zmontowanych naprędce funkcjach oceny — niekiedy oczekiwane rozwiązanie wcale nie jest rozwiązaniem najlepszym z punktu widzenia oceny celu. Jak widać, przydałaby się tu interwencja operatora, znającego oczekiwania lepiej niż funkcja celu.

Problematyczność optymalizacji nie ogranicza się zresztą do kłopotów z nadmiernym prymitywizmem funkcji celu. Metoda Metropolis jest przede wszystkim zawsze metodą optymalizacji losowej. Na szczęście zaczęliśmy od dobrego zarodka generatora losowego, ale w przypadku niektórych innych stanów początkowych generatora losowego może się okazać, że na poprawną regułę deszyfrującą poczekamy nawet tryliony iteracji. Można to łatwo samemu sprawdzić, zmieniając wartość parametru funkcji `set.seed` i próbkując metodę Metropolis porcjami po 1000 iteracji dla każdego badanego stanu generatora losowego.

Ponadto metoda Metropolis zawsze będzie skłonna do porzucania dobrych odszyfrowań. Dzięki temu algorytm jest niezachłanny. Gdy będziemy analizować szczegółowy przebieg wykonania optymalizacji, często zdarzy się nam zobaczyć, jak metoda Metropolis porzuca właściwe, oczekiwane rozwiązanie i idzie dalej, w kierunku rozwiązania nieco słabszego. Na rysunku 7.6 pokazujemy wykres logarytmu prawdopodobieństwa dla odszyfrowanego tekstu z poszczególnych kroków metody Metropolis. Widać, że ruchy algorytmu w przestrzeni rozwiązań są dość nerwowe.

Kup książkę Poleć książkę



Rysunek 7.6. Postęp metody Metropolis — szukanie optymalnej reguły deszyfrującej

Te losowe zwroty metody można stłumić na kilka często stosowanych sposobów. Jednym z nich jest powiększanie zachłanności metody z upływem czasu poprzez coraz rzadsze akceptowanie niepostępowych propozycji. Technika ta, znana jako **symulowane wyżarzanie** (ang. *simulated annealing*), jest bardzo efektywnym mechanizmem optymalizacji. Możemy ją wypróbować w swoim własnym programie, zmieniając sposób akceptowania nowej reguły deszyfrującej[4].

Można też przyjąć losowy charakter metody z dobrodziejstwem inwentarza i zamiast zawężać wynik do jednej optymalnej reguły, dawać w odpowiedzi rozkład możliwych odpowiedzi w ujęciu prawdopodobieństwa. W zadaniu deszyfrowania nie byłoby to szczególnie użyteczne, ale tam, gdzie szukamy odpowiedzi liczbowej, możliwość wygenerowania zestawu odpowiedzi bliskich odpowiedzi optymalnej do wyboru bywa bardzo pomocne.

Mamy nadzieję, że udało się nam przedstawić zasadę i sedno działania metod optymalizacji w ogóle, a w szczególności techniki angażowania optymalizacji do rozwiązywania skomplikowanych problemów uczenia maszynowego. Do niektórych przywołanych tu pomysłów będziemy wracać w dalszych rozdziałach, zwłaszcza przy omawianiu systemów rekomendacyjnych.

[4] Funkcja `optim` przyjmuje nawet opcjonalny parametr zmuszający ją do stosowania symulowanego wyżarzania zamiast standardowego algorytmu optymalizacji. Ale w naszym przykładzie i tak nie możemy skorzystać z „gotowca" w postaci funkcji `optim`, bo optymalizacja realizowana przez tę funkcję działa na wartościach liczbowych i nie obsługuje struktur danych, które wykorzystujemy do reprezentowania szyfrów.

Kup książkę Poleć książkę

Kup książkę Poleć książkę

Skorowidz

Kup książkę Poleć książkę

G

H

I

J

K

Kup książkę Poleć książkę

L

M

N

O

P

R

Kup książkę Poleć książkę

S, Ś

T

U

W

Z

Kup książkę Poleć książkę

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Programowanie

O'REILLY®

Uczenie maszynowe dla programistów

Maszyna myśląca jak człowiek to marzenie ludzkości. Dzięki dzisiejszej wiedzy i dostępnym narzędziom wciąż przybliżamy się do jego spełnienia. Zastanawiasz się, jak nauczyć maszynę myślenia? Jak sprawić, żeby podejmowała trafne decyzje oraz przewidywała najbliższą przyszłość na podstawie przygotowanych modeli? Na to i wiele innych pytań odpowiada ta wspaniała książka.

Dzięki niej poznasz język R, nauczysz się eksplorować dostępne dane, określać wartość mediany i odchylenia standardowego oraz wizualizować powiązania między kolumnami. Gdy opanujesz już solidne podstawy teoretyczne, możesz śmiało przejść do kolejnych rozdziałów i zapoznać się z klasyfikacją binarną, tworzeniem rankingów oraz modelowaniem przyszłości przy użyciu regresji. Ponadto zrozumiesz, jak tworzyć systemy rekomendacyjne, analizować sieci społeczne oraz łamać szyfry. Książka ta jest doskonałą lekturą dla pasjonatów analizy danych i wyciągania z nich wniosków.

Dzięki tej książce:

- poznasz język R
- wyciągniesz najlepsze wnioski z dostępnych danych
- znajdziesz powiązania między danymi
- przeanalizujesz sieć społeczną Twittera
- podejmiesz trafne decyzje

Naucz się czytać i analizować dane!

helion.pl
księgarnia internetowa

Nr katalogowy: 27830

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-246-9816-5

Cena 49,00 zł

9 788324 698165