Systemy uczące się w R
Zadania A

Miron Kursa

13 marca 2019

1 Niemal jak wielbłąd

Zreplikujemy *Łasicę Dawkinsa,* czyli prostą demonstrację działania algorytmów genetycznych. Program ten ma odtworzyć frazę `"METHINKS IT IS LIKE A WEASEL"` produkując losowe mutacji 28-literowego łańcucha i wybierając te zbliżające się do celu. W tym celu:

1. Napisać funkcję `score(a,b)` która zwróci procent pozycji na których w łańcuchu `a` i `b` jest ta sama litera, t.j. dla `"ABCD"` i `"XKCD"` zwróci 0.5 a dla `"ABCD"` i `"DCBA"` 0. Zapoznać się z `?strsplit`. Zrobić tak żeby `a` mogło być wektorem łańcuchów i wtedy zwracało wektor score'ów. Przyjąć że `all(nchar(a)==nchar(b))` i `length(b)==1`.

* Dodać wartowników (`?stopifnot`) którzy tych warunków dopilnują.

2. Napisać funkcję `mutate(a)` która z prawdopodobieństwem 5% zmienia każdą kolejną literę na losową z zakresu A-Z+' ', t.j. `ABCD` może zamienić na `AZCD`, `AJCG`, `AB D` itp. Zapozać się z `?paste`. Zrobić tak żeby `mutate` działał dla wektora łańcuchów.

3. Napisać funkcję `generation(curBest,mutator,scorer,population)` która zrobi `population` kopii `curBest`, zawoła na nich najpierw `mutator` a potem `scorer`, na końcu zwróci listę dwóch elementów: `newBest` z najlepszy nowo uzyskanym łańcuchem i `scores` z wektorem posortowanych score'ów.

4. Napisać funkcję `weasel(nGen=100,target="METHINKS IT IS LIKE A WEASEL", scorer=score,mutator=mutate)` która inicjalizacje `curBest` losowym łańcuchem o długości takiej jak `target`, potem `nGen` razy (w pętli `?for`) będzie wołać `generation` na `curBest`, odświeżać `curBest` zastępując go elementem `newBest` wyniku i zarazem zbierać wyniki dodając element `scores` jako nową kolumnę do macierzy `scoreHistory`.

5. Zbadać `?message` i `?sprintf` i użyć do raportowania zmian score'u w locie.

6. Puścić `weasel` dla `"METHINKS IT IS LIKE A WEASEL"`.

* Puścić `weasel` dla `"METHINKS IT IS LIKE A WEASEL"` i dla `"KOT MA ALA NA IMIE"`. Przeczytać `?matplot` i narysować jak to się nasz rozkład fitnessu propaguje przez pokolenia w obu przypadkach. Zobaczyć ile średnio iteracji potrzeba na odtworzenie celu. Napisać `scoreR` który dodaje do score'ów liczy losowe z $N(0, \sigma)$ i zbadać jak σ wpływa na przebieg fitnessu i ogólnie zdolność algorytmu genetycznego do zbiegnięcia.

2 Permutowane seriale

Przeanalizujemy oceny widzów dla poszczególnych odcinków różnych popularnych seriali, pochodzące z serwisu IMDB; dane można ściągnąć z `https://mbq.me/stuff/seriale.RData`. Po wczytaniu, proszę sprawdzić:

- Jaki jest najdłuższy serial w zbiorze (wg. liczby odcinków)?
- Jaki jest najlepiej i najgorzej oceniony według mediany?
- Jaki ma największy rozrzut ocen (wg. odchylenia ćwiartkowego)?
- Czy jest zależność między długością serialu a śrenią oceną, skrajnymi ocenami (Q_1/Q_3) i ich rozrzutem? Użyć korelacji Spearmana.

Sprawdźmy istotność statystyczną znalezionych korelacji; w tym celu napisać funkcję implementującą następujący test permutacyjny. Najpierw bierzemy nasze dwie serie danych, po czym liczymy między nimi oryginalną korelację; następnie mieszmay kolejność ocen w jednej z serii i znów liczmy korelację. Drugi krok powtarzamy wielokrotnie (powiedzmy 10 000 razy) i sprawdzamy w jakiej części przypadków nasza oryginalna korelacja jest mniejsza co do wartości od tej na przemieszanych danych; to będzie nasze nieparametryczne p-value.

Wybierzmy teraz jeden serial i sprawdźmy które seriale są od niego istotnie lepsze a które istotnie gorsze. Podobnie jak wyżej, napiszmy permutacyjny test oceniajacy istotność różnicy mediany; naszym odniesienie będziemy konstruować następująco: oceny obu seriali trzeba połączyć, przemieszać i ponownie rozdzielić między seriale by uzyskać te same liczności.

Ponieważ test zostanie powtórzony dla wielu seriali, uzyskane p-value trzeba będzie poprawić na wielokrotne testowanie (na przykład funkcją `p.adjust`).

3 Polowanie na miny

Napiszemy implementację prostego algorytmu uczenia maszynowego, kNN-a, i zastosujemy do danych Sonar. kNN to algorytm który wybiera `k` najbliższych obiektów do danego obiektu, zbiera ich klasy i tą najczęstszą zwraca jako predykcję. W tym celu:

1. Załadować pakiet `mlbench`, potem dane Sonar. Przeczytać `?Sonar`. Do dalszej zabawy oddzielić decyzję do zmiennej `Y` a resztę danych Sonar (predyktory) przerobić na macierz i zapisać w zmiennej `B`.

2. Liczenie odległości Euklidesa między wektorami liczbowymi `a` i `b` może wyglądać tak: `sqrt(sum((a-b)^2))`. Zastanowić się jak policzyć to hurtem między a a kilkoma wektorami b_i które, jak się składa, są wierszami macierzy `B` (można to robić bez żadnych pętli, tylko korzystając z wektoryzacji i funkcji `rowSum` albo `colSum`).

3. Mając funkcję odległości, złożyć funkcję pomocniczą `kNNsingle(a,B,Y,k)` która policzy numery `k` wierszy w `B` najbliższych `a`, wyciągnie te indeksy z `Y` i zobaczy jaka klasa jest tam większościowa (i tą zwróci).

4. Poskładać to w funkcję `kNN(trainX,trainY,testX,k=5)` która zawoła `kNNsingle` dla każdego wiersza `testX`

5. Przetestować na treningu z nieparzystych wierszy `B` i teście z parzystych.

* Ocenić jakość klasyfikatora cross-validacją typu 632 bootstrap; w tym celu losujemy obiekty ze zwracaniem aż dostaniemy zbiór treningowy tej samej wielkości co oryginalny set; nieużyte do tego obiekty traktujemy jako test. Z tego uzyskujemy dokładność (zdefiniowaną jako procent dobrych predykcji). Powtarzamy 30 razy i liczymy średnią.

* Powtórzyć 632CV dla wartości `k` od 1 do 10; zbierać wszystkie odczyty dokładności. Narysować boxplot-y (`?boxplot`) dokładności dla każdego `k`; użyć `notch=TRUE` do oceny czy wybór optymalnego `k` jest istotny statystycznie.

* Napisać metaklasyfikator który produkuje kNNy dla $k \in [1, \ldots, 10]$, ocenia przez 632CV i wybiera `k` dla minimalnego średniego błędu. Taką konstrukcję ocenić przez 632CV j.w. (tzw. zagnieżdżona cross-validacja). Dodać ten wynik do jako kolejny boxplot do poprzednich wyników.