WDUM MAD – praca domowa 3

Jan Pogłód

21 grudnia 2023

1 Wstęp

Celem pracy domowej było zaimplementowanie algorytmu dla modelu k najbliższych sąsiadów oraz przetestowanie go na różnych zbiorach danych. Funcja knn() przyjmuje na wejściu następujące dane:

- Macierz rzeczywistą X typu n x p, reprezentującą n punktów w R^p. Macierz ta reprezentuje zbiór treningowy.
- Macierz rzeczywistą Z typu m x p, reprezentującą m punktów w R^p. Macierz ta reprezentuje zbiór testowy.
- n - elementowy obiekt y, który odpowiada etykietom w zbiorze X
- Liczbę całkowitą $1 \leq k \leq n.$, oznaczającą liczbę najbliższych sąsiadów biorących udział w poszukiwaniu etykiety odpowiadającej punktom ze zbioru testowego
- Wartość rzeczywistą p, określającą wybór metryki minkowskiego L_p

Jako wyjście należało zwrócić m – elementowy obiekt w, gdzie w_i reprezentuje etykietę odpowiadającą obserwacji Z_i.

2 Implementacja algorytmu

2.1 Definicja 2 - Algorytm k najbliższych sąsiadów

Algorytm k-NN jest metodą klasyfikacji i regresji, która opiera się na koncepcji bliskości punktów w przestrzeni cech. Dla zbioru treningowego zawierającego pary (x_i, y_i), gdzie x_i to wektory cech, a y_i to etykiety, algorytm k-NN klasyfikuje etykietę nowej obserwacji na podstawie k najbliższych sąsiadów w przestrzeni cech. Przykład mojej własnej implementacji algorytmu przedstawiłem poniżej.

2.2 Pseudokod

Algorithm 1 Pseudokod dla algorytmu knn()

```
1:  procedure KNN(X,y,Z,k,p = 2)
2:      w ← []
    ▷ 1) Obliczanie najblizszych sasiadow
3:      for i ← 0 to Z.shape[0] do
4:          distances ← []
5:          for j ← 0 to len(y) do
6:              dj ← np.linalg.norm(Z[i,] − X[j,], ord = p)
7:              distances.append([dj, j])
8:          end for
9:          distances.sort(key = lambda x : x[0])
10:         indices = [x[1] for x in distances][: k]
11:         y = np.array(y)
12:         labels = list(y[indices])
    ▷ 2.1) Wybieranie elementu mody – lista checking
13:         checking = [0] ∗ (Z.shape[1] + 1)
14:         for el ∈ labels do
15:             checking[el] + = 1
16:         end for
17:         checking = sorted(checking, reverse = True)
    ▷ 2.2) Przypadek gdy jest wiecej niz jeden najczestszy indeks
18:         modeResults ← []
19:         if checking[0] > checking[1] then
20:             modeResult = max(set(labels), key = labels.count)
21:         else
22:             for el ∈ set(labels) do
23:                 if labels.count(el) == checking[0] then
24:                     modeResults.append(el)
25:                 end if
26:             end for
27:             modeResult = np.random.choice(modeResults)
28:         end if
29:         w.append(modeResult)
30:     end for
31:     return w
32: end procedure
```

W powyższym pseudokodzie w części pierwszej zaimplementowałem sposób w jaki algorytm mierzy odległość pomiędzy wektorem Z_i oraz X_j dla każdego z indeksów *j=0,...,m-1*. W funkcji knn() można wybrać dowolną metrykę Minkowskiego L_p manipulując wartością p, gdyż funkcja *linalg.norm* z pakietu numpy może obliczyć dowolną z metryk dla p - niezerowej wartości całkowitej. W części testowej będziemy korzystali z metryk L_1, L_2 oraz L_{inf}. Dalej wyliczone odległości dodajemy do listy *distances*, którą następnie sortujemy rosnąco zaczynając od punktu w przestrzeni, który okazał się być najbliżej szukanej wartości ze zbioru testowego i następnie indeksy tych punktów przechowujemy w liście indices, która jest obcięta do k punktów - najbliższych punktów w zależności od punktu w przestrzeni wielowymiarowej Z_i. W ten sposób wybieramy etykiety odpowiadające odpowiednim indeksom w zbiorze treningowym X_j.

Część druga skupia się na jak najodpowiedniejszym wybraniu mody z elementów, które są znalezionymi etykietami z części 1). W przypadku gdy najczęściej pojawiający się element jest tylko jeden, to ustawiamy go jako modę i zwracamy jako *i-ty* element wektora w. Jeżeli jest więcej niż 1 taki element, to z tych najczęściej pojawiających się wybieramy jeden z jednakowym prawdopodobieństwem z rozkładu jednostajnego za pomocą funkcji *random.choice* z pakietu numpy. Dodatkowa lista checking pozwala nam sprawdzić, ile dokładnie jest wybranych elementów z konkretnych klas.

Zważając, że funkcja z pakietu numpy, która liczy odległość punktów w przestrzeni ma złożoność p, bo wektory są z przestrzeni R^p, to całkowita złożoność powyższego algorytmu to *O(mnp)*

3 Testy

Jako zbiory do testowania przyjąłem dwa zbiory z przestrzeni R^2. Pierwszy z nich dobrany został w taki sposób, aby punkty pierwszej klasy i punkty drugiej klasy tworzyły osobne obszary w przestrzeni, drugi zaś został wygenerowany zupełnie losowo. Przeanalizowałem każdą z metryk L_1, L_2 oraz L_{inf} i przeprowadziłem badanie polegające na najlepszym dobrze liczby najbliższych sąsiadów. Na początku otrzymałem następujące wyniki dla losowo wygenerowanych 40 punktów w R^6 testujących skuteczność poszczególnych metryk i liczby k. Jako miarę przyjąłem dokładność.

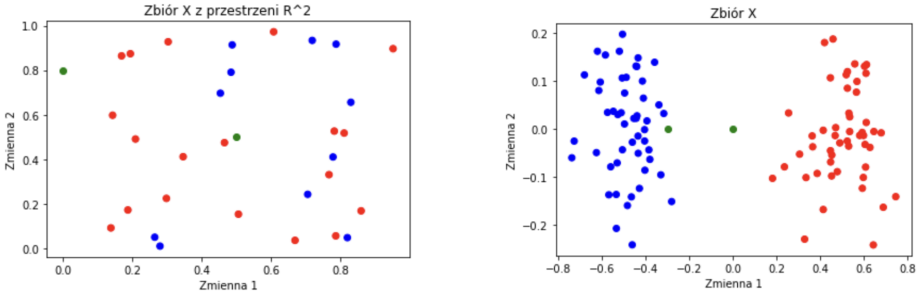
Metryka	$k = 5$	$k = 15$	$k = 25$	$k = 35$
L_1	75%	58%	50%	50%
L_2	83%	50%	50%	50%
L_{inf}	67%	50%	50%	50%

Jak możemy zobaczyć na przykładzie powyższej tabeli, zaimplementowany algorytm okazał się być najskuteczniejszy dla najmniejszej liczby sąsiadów oraz dla metryki L_2. To dla nich przyjął on największą miarę dokładności. Na tej

podstawie kolejne testy wykonywać będę dla miary L_2 oraz przy liczbie sąsiadów k = 5. Dodatkowo warto podkreślić, iż przy jednym sąsiedzie algorytm z dokładnością = 100% kwalifikuje dane treningowe tożsame z testowymi, co potwierdza jego poprawność.

3.1 Testy na zbiorach w R^2

Jako dane do testowania przyjąłem dwie skrajnie różne ramki z przestrzeni R^2. Pierwsza z nich powstała dla wartości losowych, zaś druga dzieli przestrzeń na dwa obszary względem klas. Klasy rozdzieliłem kolorami czerwonym i niebieskim, zaś nowy punkt do kwalifikacji kolorem zielonym. Wyniki i próby kwalifikacji kolejnego punktu przedstawiłem poniżej.



Przypadek po lewej stronie (punkty dobrane losowo) otrzymał dokładność na poziomie 67% zaś ten, w którym punkty są podzielone na obszary (z prawej strony) na poziomie 100%, a więc ze stu procentową skutecznością klasyfikuje on poszczególne punkty ze zbioru testowego do klas. Jako punkty do klasyfikacji w przypadku przykładu po lewej stronie użyłem punktów [0,0.8] i [0.5,0.5]. Zostały one zakwalifikowane przez algorytm jako odpowiednio klasy czerwona i niebieska. Widzimy, że lewy górny róg jest zdominowany przez klasę czerwoną, więc decyzja algorytmu jest jak najbardziej trafna, zaś punkt na środku pola został zakwalifikowany jako klasa niebieska, mimo dwóch bardzo bliskich punktów czerwonych. W przypadku powierzchni na wykresie po prawej stronie algorytm ze 100% pewnością dla punktów [0,0] i [-0.3,0] rozpoznał, że odpowiadające im klasy to odpowiednio czerwony i niebieski. Widzimy, że rozmieszczenie klas na powierzchni ma olbrzymie znaczenie, na dokładność klasyfikacji punktów.

4 Podsumowanie

Pokazaliśmy, że zaimplementowanie własnego algorytmu potrafi odpowiedzieć na więcej pytań dotyczących rozpatrywanego modelu. Postawione w ten sposób rozwiązanie pozwala przy testowaniu na większą dowolność i analizę wyników, co pozwoli nam wskazać najlepsze parametry dla konkretnych problemów.