# Uczenie maszynowe, różne warianty (2)

Paweł Rychlikowski

Instytut Informatyki UWr

31 maja 2019

# Klasteryzacja. Przypomnienie

## Definicja

Klasteryzacja (grupowanie) to zadanie identyfikacji w próbce uczącej naturalnych grup związanych ze sobą obiektów.

## Algorytm $K$-średnich

Algorytm przeplata dwie fazy:

1. Przypisanie każdego punktu do najbliższego mu prototypu
2. Obliczenie nowych prototypów jako **średnich** wszystkich punktów przypisanych do tego samego prototypu

# Co oblicza algorytm $K$-średnich?

**Cel**

Chcemy, żeby prototyp przybliżał wszystkie przypisane mu elementy.

**Daleka analogia:** Prototyp jest takim elektorem, który przybliża poglądy swoich wyborców.
Każdy wyborca jest zadowolony, jeżeli ma elektora dobrze rozumiejącego jego preferencje.

# $K$-średnich. Przykładowe zastosowania

1. Wybór palety $K$ kolorów, dostosowanej do obrazka.
2. Sklejanie obrazka z kwadratów (kompresja wektorowa)
3. Grupowanie słów podobnych (jeżeli reprezentujemy słowa jako wektory)
    - Po zastosowaniu do fraz: odpoczynek w pięknym miasteczku $\approx$ wypoczynek w uroczym kurorcie

# Co oblicza algorytm $K$-średnich? (2)

- Interesuje nas, aby **każdy element, jak najmniej się różnił od swojego reprezentanta** (czyli średniej, prototypu, centroidu)
- Miara: błąd średniokwadratowy, czyli

$$\sum_{x \in \text{Dane}} (x - \text{reprezentant}(x))^2$$

**Definicja**

Powyżej zdefiniowaną wielkość nazwiemy błędem klasteryzacji.

# $K$-średnich jako minimalizacja błędu

## Etap przypisywania

Prototypy ustalone. Każdy egzemplarz trafia do bliższego prototypu (czyli błąd maleje).

## Etap liczenia średnich

Patrzymy na 1 klaster. Policzmy pochodną po $c$ dla

$$\sum_{i=1}^{N}(x_i - c)^2$$

Błąd klasteryzacji dla jednego klastra osiąga minimum w punkcie będącym średnią punktów klastra.

## Wniosek

Oba etapy nie zwiększają błędu (tzn. jeżeli coś robią, to błąd maleje). Czyli osiągamy lokalne minimum.

# Algorytm K-średnich. Demonstracja (2)

- Losujemy pewną liczbe punktow na plaszczyźnie, tak aby w naturalny sposób tworzyły klastry.
- Wybieramy początkowe centra z populacji punktów
- Obserwujemy, jak działa algorytm

Popatrzmy na demonstrację `kmeans.py`
Dodatkowe funkcje: `restart`, `new`

# K-średnich. Kilka uwag końcowych

- Możemy powtarzać losowanie punktów kilka razy i wybrać najmniejszy błąd
- Możemy wykonać (w celu przyspieszenia) algorytm dla podpopulacji punktów (i potem tylko 1-2 etapy dla wszystkich punktów)
- Zauważmy, że im większe $K$, tym (średnio) mniejszy błąd grupowania

**Pytanie**

Jak wykorzystać ostatnie spostrzeżenie do wyboru wartości $K$ (wskazówka: jak wygląda wykres wartości błędu w zależności od $K$)

# Wykrywanie nieprawidłowości

Potencjalnie bardzo użyteczne zadanie: można na przykład analizować pomiary różnych parametrów jakiegoś skomplikowanego systemu (skrzydło samolotu pasażerskiego) i zauważać, że coś dziwnego się dzieje

- **Pytanie:** Jak wykorzystać algorytm K-means do wykrywania anomalii?
- Charakterystyka anomalii:
  - Daleko od centrum
  - Najbliżsi sąsiedzi należą do różnych klastrów

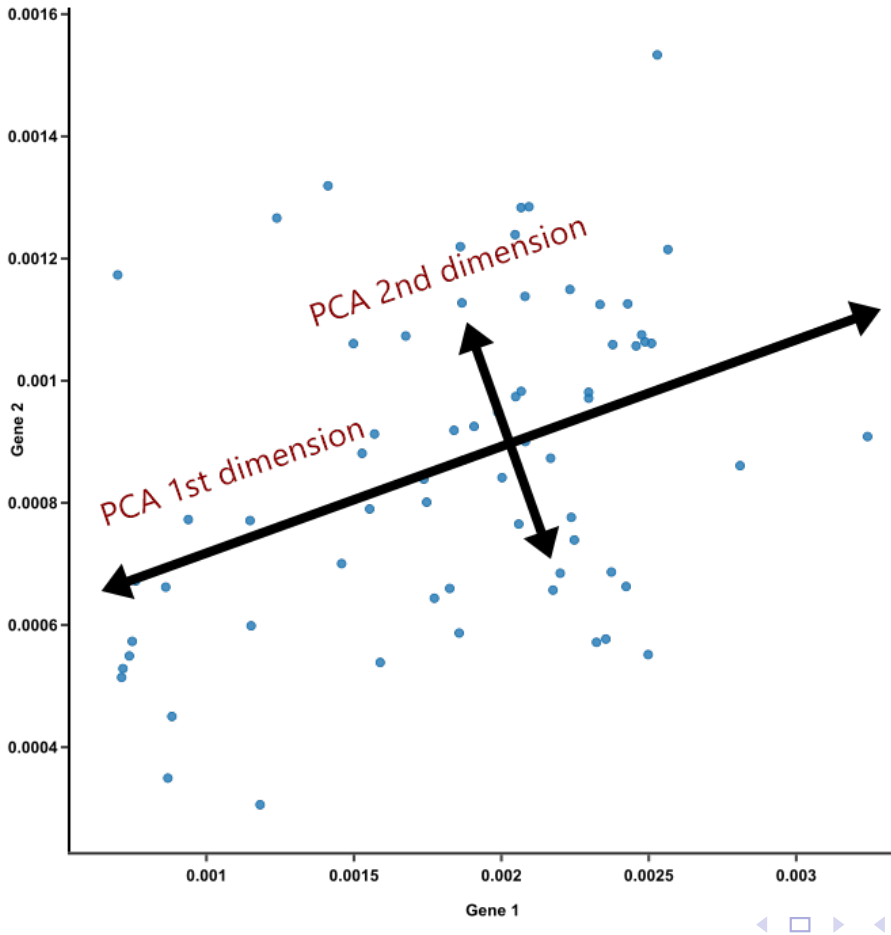
Wykonanie algorytmu $K$-średnich i analiza poszczególnych punktów daje możliwość zidentyfikowania „dziwnych" elementów (potencjalnych nieprawidłowości).

# Redukcja wymiarów

- **Cel**: „zagęszczenie danych” (umożliwiające, być może, lepsze działanie innych algorytmów)
- **Dodatkowa korzyść**: jak zredukujemy liczbę wymiarów do 2, to możemy zbiór danych ładnie narysować (i być może zobaczyć jakieś prawidłowości)
- Redukcja wymiarów oznacza usunięcie informacji, ale być może usuniemy nieistotne informacje, czyli szum.
- Przykładową metodą (omawianą na algebrze) jest PCA, czyli analiza głównych składowych

# Principal Component Analysis

- Identyfikujemy osie, które odpowiadają za największą zmienność danych
- Obracamy przestrzeń i pozostawiamy tylko najważniejsze wymiary.



# Algorytmy wizualizacji

- Mamy punkty w przestrzeni wielowymiarowej.
- Chcemy przypisać im punkty na płaszczyźnie (2D)
- Jak? (wskazówka: potrafimy liczyć odległość w przestrzeni wielowymiarowej)

**Ogólna zasada**

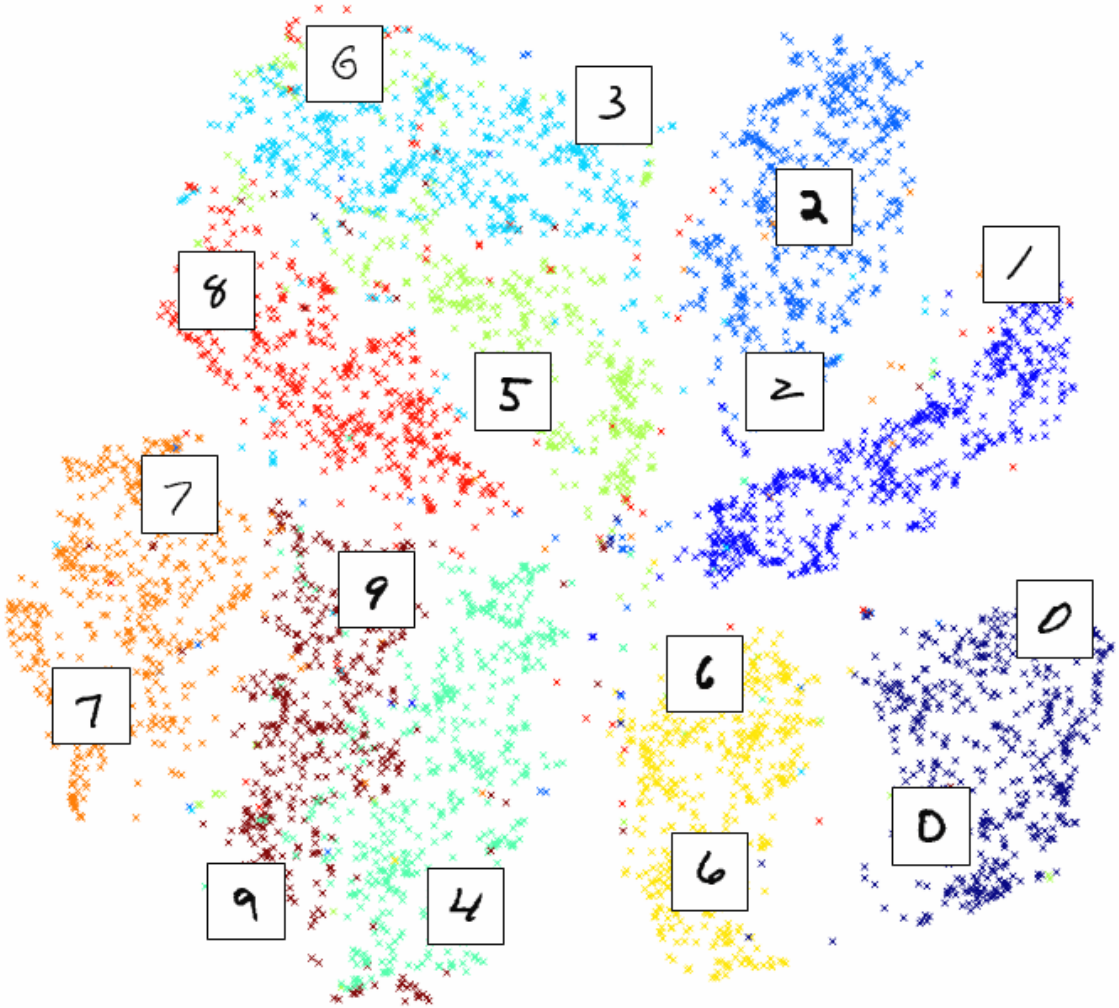
Staramy się, by odległości w 2D odpowiadały tym w oryginalnej przestrzeni (np. 500D)

Niektóre algorytmy można interpretować jako tworzenie układu punktów połączonych sprężynkami, punkty „podobne” się przyciągają, odległe – odpychają, szukamy równowagi tego układu dynamicznego.

# Wizualizacja obrazów cyfr (MNIST)

Algorytm t-SNE (który można intepretować „sprężynkowo”):

**MNIST dataset** – Two-dimensional embedding of 70,000 handwritten digits with t-SNE



# Regresja liniowa w uczeniu ze wzmocnieniem

- Definiujemy
$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$
- Weźmy grę w Dżunglę z losowym przeciwnikiem (zauważmy, że to jest MDP)
- Przykładowe cechy (propozycje?):
  - Czy jest bicie (0/1)?
  - Czy jest bicie słonia, szczura, kota (dla każdej bierki cecha)?
  - Czy ruch zbliża do jamy przeciwnika? (0/1)
  - Czy ruch jest skokiem zbliżającym do jamy?
  - Czy najbliższa jamie bierka się zbliżyła?
  - Czy podchodzimy pod bicie?
  - Czy bijemy bierkę w pułapce
  - Czy ruch jest do przodu (w lewo, w prawo, w dół)?
  - Czy zajmujemy któreś z wskazanych pól (związanych z atakiem bądź obroną)

# TD-learning

- Temporal Difference Learning – metoda uczenia wartości $V$ (używana na przykład w grach)
- Idea:
  - Generuj dane z rozgrywek
  - Naucz się wag funkcji heurystycznej analizując te dane

**Uwaga**

W najprostszym przypadku (czyli liniowym) mamy:

$$V(s; w) = w \cdot \phi(s)$$

# Generowanie danych

**Definicja**

Polityka odruchów (reflex policy) – to strategia agenta, w której podejmuje decyzje analizując przybliżoną funkcję oceniającą konsekwencje działań (w grach: stany po ruchu)

- Do generowania danych możemy wykorzystać politykę odruchową (czyli naszą aktualną funkcję oceniającą)
- **Problem**: tak wygenerujemy tylko jedną rozgrywkę (albo bardzo niezróżnicowaną populację rozgrywek)

# Generowanie danych (2)

Konieczne jest wprowadzenie losowości:

a) Polityka $\varepsilon$-zachłanna (pamiętajmy o zmianie znaczenia $V$ dla Mina i Maxa)

b) Losowanie zgodne z prawdopodobieństwem „softmaxowym", czyli:

$$P(s,a) = \frac{e^{V(\text{succ}(s,a))}}{\sum_{a' \in \text{Actions(s)}} e^{V(\text{succ}(s,a'))}}$$

c) Dla gier z z rzucaniem kostkami (z elementem losowym) można wybierać zawsze optymalne ruchy (sama gra zapewnia czynnik eksploracyjny)

# Reguła TD-learningu

- **Predykcja**: $V(s; w)$
- **Cel**: $r + \gamma V(s'; w)$ ($s'$ to kolejny stan w rozgrywce)

## Streszczenie reguły

W przypadku większości gier ($\gamma = 1$, nagroda na końcu), sprowadza się to do:

a) Staraj się, by podczas dobrych gier wartość planszy po ruchu zbytnio się nie zmieniała (dla minmaxa i optymalnej strategii powinna być ona stała)

b) Zwiększaj wartośc sytuacji bliskich zwycięstwa (wykorzystanie $r$ pod koniec).

# Ogólne zasady

- Funkcja celu:

$$\frac{1}{2}(\text{prediction}(w) - \text{target})^2$$

- Gradient:

$$(\text{prediction}(w) - \text{target})\nabla_w(\text{prediction}(w))$$

- Reguła uaktualniania:

$$w \leftarrow w - \eta((\text{prediction}(w) - \text{target})\nabla_w(\text{prediction}(w)))$$

# Algorytm TD-learning

## Algorytm

Dla każdego $s, a, r, s'$ wykonuj:

$$w \leftarrow w - \eta(V(s; w) - (r + \gamma V(s', w)))\nabla_w V(s; w)$$

Dla funkcji liniowej:

$$V(s, w) = w \cdot \phi(s)$$

mamy

$$\nabla_w V(s, w) = \phi(s)$$

# Porównanie TD-learning i Q-learning

**Algorithm: TD learning**

On each $(s, a, r, s')$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$

**Algorithm: Q-learning**

On each $(s, a, r, s')$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta[\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}]\nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

# Inne uwagi o uczeniu w grach

- Można łączyć TD learning z uczeniem polityki. AlphaGo Zero tak właśnie robiło.
- Można stosować metody poprawiania polityki/oceny:
  a) Bazujące na alpha-beta search (tak uczyć funkcję oceniającą, żeby miała „inteligencję" taką, jak poprzednia wersja)
  b) MCTS policy improvement (żeby nowa polityka udawała jak najlepiej starą wspomagającą się symulacjami MCTS)

**Uwaga**

Takie metody były używane w różnych słynnych programach:

1. Warcaby (Samuel, 1965)
2. Tryktrak, czyli Backgammon (Tesauro, ok. 1990)
3. AlphaGoZero (DeepMing, 2017)