Podstawy Sztucznej Inteligencji (PSI)

Lab 2 - omówienie

Pytania? Wątpliwości? Uwagi?

Częste kwestie

Ekstrakcja kolumny

- zamiast robić osobno wyciągnięcie i `.drop()`, można metodą `.pop()`
- konwersje typu najlepiej robić typami Pythonowymi, chyba, że szczególnie potrzebujemy innego

```
y = X.pop("class").astype(int)
```

random_state

- dobrze jest sobie ustawić globalną zmienną i z niej wszędzie korzystać

```
RANDOM_STATE = 0
```

- dobre wartości: 0, 1, 42

SMOTE - czy pomaga?

- zazwyczaj tak… ale nie zawsze
- zależy od kształtu chmur punktów i tego, czy zagęszczenie obszaru nam w ogóle pomaga

Imputacja zmiennej kategorycznej

- moda (dominanta)
- klasyfikacja, szczególnie metody najbliższych sąsiadów (k nearest neighbors)

Podejmowanie decyzji

Podejmowanie decyzji

- **zadanie:** mamy pewne sytuacje i chcemy podejmować w nich decyzje, tak, aby mieć jak najlepszy wynik
- formalnie:
  - **przestrzeń stanów** - wszystkie możliwe sytuacje
  - **stan** - sytuacja, opisywana zwykle wektorem zmiennych
  - **akcja** - decyzja do podjęcia, wykonanie czynności
  - **nagroda** - wynik akcji lub sekwencji akcji; może być dodatnia lub ujemna (kara)

Przykład - szachy

- przestrzeń stanów - możliwe ułożenia planszy
- stan - aktualne ułożenie planszy
- akcja - ruch wybraną figurą
- nagroda:
  - natychmiastowa - zmiana ułożenia planszy na lepsze/gorsze
  - finalna - wygrana / przegrana / pat

Podejścia

Algorytmy klasyczne:

- proste
- szybkie
- często małe wymagania sprzętowe
- wystarczające dla wielu problemów

Uczenie ze wzmocnieniem (reinforcement learning):

- bardziej złożone
- często (bardzo) duże wymagania sprzętowe
- długotrwały i często trudny trening, szybkie wnioskowanie
- doskonałe wyniki, lepsze od ludzi, dla wielu trudnych problemów

Zastosowania

- **gry** - oraz wszystko, co można przedstawić jako grę:
  - szachy - StockFish, AlphaZero
  - go - AlphaGo
  - Total War: Rome II
  - wybór kart w Magic: The Gathering
  - przewidywanie struktury trzeciorzędowej (przestrzennej) białek - AlphaFold
  - mnożenie macierzy - AlphaTensor

Drzewo stanów

- ogólną reprezentacją przestrzeni jest **graf stanów (state graph)**
- w szczególnym przypadku, kiedy planujemy decyzje, mamy **drzewo stanów (state tree)**
- węzły - stany, przejścia - akcje
- nagrody w liściach drzewa, lub na krawędziach

Algorytm minimax

Teoria gier

- **teoria gier (game theory)** - "the study of mathematical models of strategic interactions among rational agents"
- mamy graczy, którzy starają się zyskać jak największą nagrodę w grze przeciwko sobie
- najprościej - **gra dla 2 graczy (two player games)** - jeden maksymalizuje, a drugi minimalizuje nagrodę
- **gry o sumie zerowej (zero sum games)** - takie, w których zysk jednego gracza jest odpowiednią stratą drugiego gracza; gdy jeden zyskuje X, drugi traci X
- każdy gracz ma pewną **strategię (strategy)** podejmowania decyzji

Twierdzenie o minimaksie

Dla każdej dwuosobowej gry o sumie zerowej istnieje wartość V i mieszana strategia dla każdego gracza, takie, że:

- biorąc pod uwagę strategię gracza drugiego, najlepszą możliwą spłatą dla gracza pierwszego jest V
- biorąc pod uwagę strategię gracza pierwszego, najlepszą możliwą spłatą dla gracza drugiego jest -V

Odpowiednia strategia gracza 1. gwarantuje mu spłatę V niezależnie od strategii gracza 2. i podobnie gracz 2. może zagwarantować sobie spłatę -V.

Wnioski z twierdzenia o minimaksie

- minimalizujemy maksymalną spłatę dla drugiego gracza
- jeżeli rozważamy grę o sumie zerowej, to my zyskujemy V, a drugi gracz traci V, to **jednocześnie** mamy **strategię optymalną** - maksymalizujemy własną minimalną spłatę
- jest to **równowaga Nasha (Nash equilibrium)** - każdy z graczy wybrał optymalną strategię, biorąc pod uwagę strategie innych rzeczy, i nie można zyskać więcej, zmieniając swoją strategię

- **wniosek:** metoda minimaksu (minimalizacji maksymalnej spłaty oponenta) daje nam optymalną strategię w grach o sumie zerowej

Przykłady

Kod (3:59)

Algorytm minimax

```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

Algorytm minimax - złożoność obliczeniowa

- **współczynnik rozgałęzienia (branching factor)** - `b`, średnia liczba decyzji możliwych do podjęcia w każdym węźle drzewa
- **głębokość (depth)** - `d`, liczba decyzji, dla której rozważamy drzewo

- **złożoność obliczeniowa minimaksu:** `O(b^d)`

Alfa-beta pruning

Alfa-beta pruning

- metoda **ulepszająca złożoność obliczeniową** dla minimaxu
- bardzo znaczące ulepszenie, stosowane zawsze w praktyce
- **idea:** kiedy wiemy, że w danym poddrzewie na 100% nie ma lepszego wyniku niż najlepszy dotychczasowy, to można je zignorować
- DeepBlue, który pokonał Kasparova, opierał się głównie na tej metodzie

Alfa-beta pruning

- utrzymujemy 2 wartości:
  - alfa - minimalna wartość, co do której ma pewność gracz maksymalizujący
  - beta - maksymalna wartość, co do której ma pewność gracz minimalizujący
- kiedy dla pewnego poddrzewa beta < alpha, to można przerwać sprawdzanie - grając optymalnie, nigdy nie doprowadzimy do tej sytuacji
- odpowiada to sytuacji, gdy maksymalny wynik gracza minimalizującego jest mniejszy od minimalnego wyniku gracza maksymalizującego

Przykłady

Kod (8:48)

Algorytm alfa-beta pruning

```
function minimax(position, depth, alpha, beta, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha
                break
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, alpha, beta, true)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
        return minEval
```

Złożoność obliczeniowa

- **oczekiwana:** O(sqrt(b^d))
- możemy zbudować 2 razy większe drzewo tym samym kosztem obliczeniowym
- zależy głównie od **kolejności węzłów** - dobre ułożenie to klucz do efektywności
- sortowanie - zwykle heurystyka, zależna od problemu

Monte Carlo Tree Search (MCTS)

Metody typu Monte Carlo

- metody dla problemów **trudnych obliczeniowo**, oparte o **wielokrotne losowe symulacje**
- pierwotnie zaproponowane w laboratorium Los Alamos przez Stanisława Ulama (matematyk lwowski), zainspirowane grą w pasjansa, kiedy rozważał problem obliczenia energii generowanej przy zderzeniu neutronów
- **idea:** jak problem jest za trudny, przeprowadźmy bardzo dużo losowych symulacji i zagregujmy wyniki

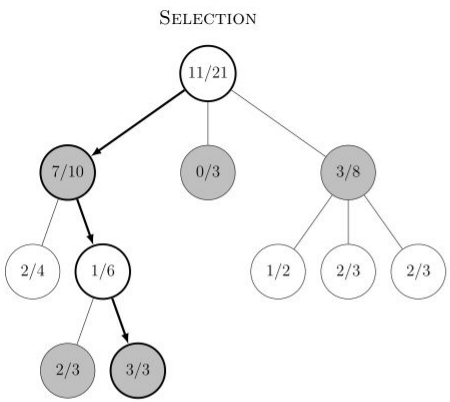
Pure Monte Carlo game search

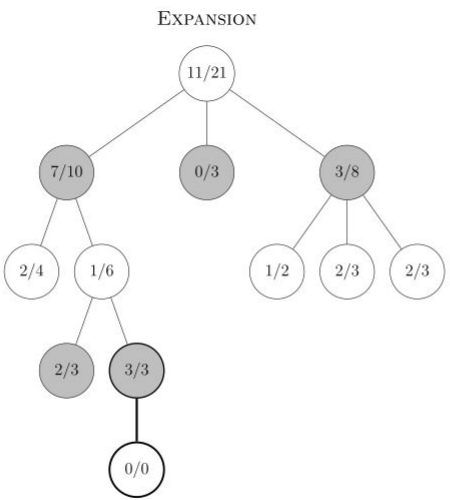
- **idea:** czysty brute force
- zakładamy, że mamy dane **drzewo gry** i najbliższe możliwe ruchy
- dla każdego z ruchów rozgrywamy **k** gier, grając całkowicie losowo
- wybieramy ten ruch, który daje największe prawdopodobieństwo wygranej
- **wyniki:** całkiem niezłe, zwykle im większe k, tym lepsze, a problem jest idealnie uwspółbieżnialny

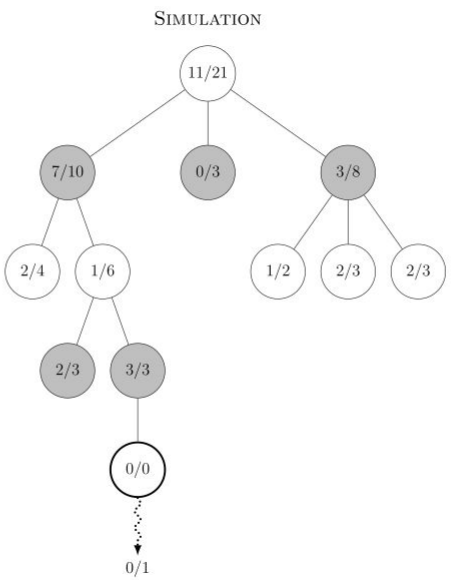
Monte Carlo Tree Search

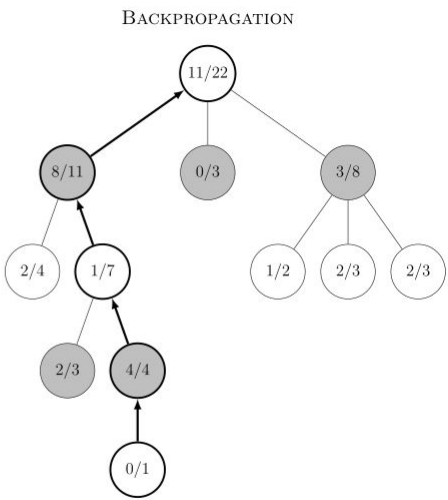
1. **Selekcja:** wystartuj z aktualnego stanu gry, korzenia R. Wybieraj kolejne dzieci, aż do liścia L, który jeszcze nie był wybrany.
2. **Ekspansja:** o ile L nie jest zakończeniem gry, stwórz z niego dzieci (możliwe ruchy) i wybierz dziecko C.
3. **Symulacja:** rozegraj losową grę, startując od dziecka C.
4. **Propagacja wsteczna:** użyj informacji o wyniku gry, aby zaktualizować ścieżkę od C do R

Monte Carlo Tree Search









MCTS - jak to zrobić?

- mamy 3 problemy:
  a. Jak wybierać węzły-dzieci?
  b. Jak grać grę z dziecka?
  c. Jak aktualizować wartości po grze?
- problem b jest najprostszy - można grać nawet losowo
- problemy a i c muszą równoważyć **exploration** oraz **exploitation** - sprawdzanie nowych możliwości oraz wykorzystanie zdobytej wiedzy

Wybór węzłów - UCT

- rozwiązanie obu problemów naraz, wyprowadzone matematycznie, często daje bardzo dobre wyniki
- **zaleta:** automatycznie balansuje exploration oraz explotation
- wzór: $\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$
- `w_i` - liczba zwycięstw (wins) dla węzła po i-tym ruchu
- `n_i` - liczba symulacji dla węzła po i-tym ruchu
- `N_i` - łączna liczba symulacji po i-tym ruchu
- c - stała, balansuje exploration-exploitation tradeoff, zwykle sqrt(2)