SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Dolores Čaprić

Zagreb, 2022./2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentori:

izv. prof. dr. sc. Tomislav Stipančić, dipl. ing.

Student:

Dolores Čaprić

Zagreb, 2022./2023.

Izjavljujem da sam ovaj rad izradila samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se mentoru izv. prof. dr. sc. Tomislavu Stipančiću na pruženoj pomoći te uloženom trudu i vremenu tijekom izrade ovog rada.

Posebno se zahvaljujem svojoj obitelji na podršci i razumijevanju te svim prijateljima koji su mi olakšali dosadašnji dio studija.

Dolores Čaprić

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za završne i diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: 602 – 04 / 23 – 6 / 1	
Ur.broj: 15 - 1703 - 23 -	

ZAVRŠNI ZADATAK

Student: **Dolores Čaprić** JMBAG: **0035219745**

Naslov rada na hrvatskom jeziku: **Problem višerukog razbojnika kod razvoja modela podržanog učenja**

Naslov rada na engleskom jeziku: **Multi-armed bandit problem in the development of reinforcement learning models**

Opis zadatka:

Algoritmi podržanog / ojačanog učenja omogućuju djelovanje računalnim agentima u sklopu realnog svijeta. Računalni agent odabire akciju i dobiva povratnu informaciju iz okoline u obliku stanja (ili opažanja) i nagrada. Postupak učenja računalnog modela se temelji na sustavu nagrađivanja / kažnjavanja gdje je cilj računalnog agenta postići neki zadatak sa što više nagrađivanja, odnosno što manje kazni.
Algoritam višerukog razbojnika (engl. multi-armed bandit) predstavlja jedan od najosnovnijih načina primjene podržanog učenja. Naziv višeruki razbojnik dolazi od automata u kockarnicama, gdje jednoruki razbojnik odgovara jednom automatu. Za više automata za igre na sreću, potrebno je znati koju polugu treba povući da bi se osvojio najveći iznos novca. U tome slučaju radi se o procjeni igrača koji treba odlučiti hoće li koristiti svoje iskustvo igranja na postojećim automatima ili će probati sreću na automatu koji još nije koristio kako bi dobio nagradu u vidu osvojenog novčanog dobitka.
U radu je potrebno ostvariti računalnog agenta koji djeluje po principima podržanog učenja temeljem algoritma višerukog razbojnika koristeći nekoliko tehnika nagrađivanja. Računalnog agenta je potrebno ostvariti koristeći programski jezik Python.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:
30. 11. 2022.

Zadatak zadao:
Doc. dr. sc. Tomislav Stipančić

Datum predaje rada:
1. rok: 20. 2. 2023.
2. rok (izvanredni): 10. 7. 2023.
3. rok: 18. 9. 2023.

Predviđeni datumi obrane:
1. rok: 27. 2. – 3. 3. 2023.
2. rok (izvanredni): 14. 7. 2023.
3. rok: 25. 9. – 29. 9. 2023.

Predsjednik Povjerenstva:
Prof. dr. sc. Branko Bauer

SADRŽAJ

POPIS SLIKA

POPIS OZNAKA

Oznaka	Jedinica	Opis
t	/	Vremenski korak
A_t	/	Radnja poduzeta u vremenskom koraku *t*
R_t	/	Nagrada dobivena u vremenskom koraku *t*
k	/	Broj radnji
a	/	Radnja u vremenskom koraku *t*
$q(a)$	/	Vrijednost radnje *a*
$Q_t(a)$	/	Procijenjena vrijednost nagrade *a*
n	/	Broj poduzimanja radnje *a*

SAŽETAK

Tema ovog rada je razvoj algoritama koji pomoću metode istraživanja i iskorištavanja rješavaju problem višerukog razbojnika. To je problem u okviru strojnog učenja u kojem agent odabire radnje kako bi maksimalno povećao dobivenu nagradu. Ideja algoritama je implementirana u programski jezik Python, dok su rezultati prikazani u web aplikaciji Jupyter Notebook. Prikaz rezultata sadrži numerički i grafički dio. U prvom dijelu rada razrađena je teorijska i matematička osnova koja omogućava razumijevanje i primjenu algoritama. U drugom dijelu je za svaki algoritam detaljno objašnjen princip te implementacija u Python-u. Također su dani dobiveni rezultati koji prikazuju razinu uspješnost svakog algoritma.

Ključne riječi: višeruki razbojnik, Python , Jupyter Notebook, istraživanje, iskorištavanje

SUMMARY

The topic of this paper is the development of algorithms that solve the multi-armed bandit problem using exploration and exploitation methods. It is a machine learning problem in which an agent chooses actions to maximize the obtained reward. The algorithms are implemented in the Python programming language, while the results are presented in the Jupyter Notebook web application. The display of results contains a numerical and graphic part. Theoretical and mathematical basis that enables the understanding and application of algorithms is explained in the first part of this paper. In the second part, the principle and implementation in Python are explained for each algorithm. The obtained results that show the level of success of each algorithm are also given.

Key words: multi-armed bandit, Python, Jupyter Notebook, exploitation, exploration

1. UVOD

Kako tehnologija napreduje, a svjetska populacija raste, također raste i količina podataka koju stvaramo. Zbog težnje za neistraženim i do sada nemogućim stvarima, sve više se razvija umjetna inteligencija i njeni elementi. Jedan od poznatijih elemenata umjetne inteligencije je problem višerukog razbojnika (eng. *multi armed bandit problem*), temeljni izazov u donošenju odluka u uvjetima nesigurnosti. Ovaj problem se javlja u mnogim scenarijima stvarnog svijeta te je glavni dio čovjekove svakodnevnice. Temelji se na donositelju odluka koji se suočava s ograničenim resursima, vremenom i povratnim informacijama. Problem višerukog razbojnika koristi se u online oglašavanju, kliničkim ispitivanjima, sustavima preporuka, financijama, robotici te mnogim drugim granama.

Tijekom proteklih desetljeća, problem višerukog razbojnika je opsežno proučavan, što je dovelo do razvoja algoritama koji se kreću od jednostavnijih pa sve do sofisticarnijih. Dosadašnje istraživanje je pokazalo da ovi algoritmi dobro funkcioniraju u različitim postavkama, ali također imaju ograničenja koja ovise o strukturi problema i složenosti zahtjeva.

U ovom radu će se prvo opisati metode umjetne inteligencije, koje čine teorijsku podlogu ovog problema, a zatim će se osmišljeni algoritmi implementirati u programski jezik Python. Pomoću Jupyter Notebook-a dobiveni rezultati će se prikazati u numeričkom i grafičkom obliku.

2. STROJNO UČENJE

Strojno učenje (eng. *machine learning*) je potpodručje umjetne inteligencije (eng. *artificial inteligence*), koja se definira kao sposobnost stroja da oponaša inteligentno ljudsko ponašanje. Svaki sustav umjetne inteligencije koristi se za obavljanje složenih zadataka, na način koji je najsličniji ljudskom rješavanju problema. Prema navedenome, strojno učenje se opisuje kao jedan od načina korištenja umjetne inteligencije. Informatičar Arthur Samuel je 1950. godine, pomoću igre dame, definirao strojno učenje kao područje proučavanja koje računalima daje sposobnost učenja bez eksplicitnog programiranja. Robert Nealey, samoproglašeni majstor dame, 1962. godine je izgubio igru od računala te se to zabilježilo kao glavna prekretnica u području strojnog učenja [1].

Za jednostavnije zadatke tradicionalno programiranje je bilo dovoljno, ručno izrađen program s ulaznim podacima se unio u računalo te se dobio željeni izlaz. Razvojem i napretkom tehnologije zadaci su postajali sve složeniji te klasičnom programiranju nedostižni. Složeni zadatak poput prepoznavanja ljudi s fotografije je bio nemoguć, kako za programere, tako i za računala, sve do pojave strojnog učenja. Točno kod tih zadataka se izdvojila jedna od glavnih prednosti i prepoznatljivosti strojnog učenja, a to je naučiti računalo da radi ono što je ljudima prirodno, to jest naučiti nešto iz prethodnog iskustva. Princip je da pomoću naučenog računalo lakše predviđa određene stvari ili situacije [1].

Strojno učenje počinje s određenim podacima, kao što su brojevi, razni tekstovi ili fotografije. To mogu biti bankovne transakcije, zapisi o popravcima, razna izvješća, vremenski niz podataka iz senzora te slike ljudi ili nekih određenih artikala. Podaci se prikupljaju i pripremaju za korištenje kao podaci za obuku na kojima će se računalo trenirati. Što je više podataka, vjerojatnost točnosti programa će biti veća. Za prikupljanje i dostavljanje podataka su zaslužni programeri. Njihov zadatak je pažljivo odabrati podatke te pustiti računalni model da se sam osposobi za pronalaženje obrazaca ili predviđanja. U svrhu poguravanja modela strojnog učenja prema što točnijim rezultatima, ljudski programer može promijeniti ili dodati pojedine parametre. Postoje određeni podaci koji se izvlače iz podataka za obuku te se koriste u evaluacijske svrhe. Prikazivanjem tih podataka, testira se model strojnog učenja. Rezultat je model koji se u budućnosti može koristiti na različitim skupovima podataka [1] [2].

S obzirom na vrste problema koji se nastoje riješiti, područje strojnog učenja često se dijeli na određena potpodručja.

Osnovna tri tipa strojnog učenja su:

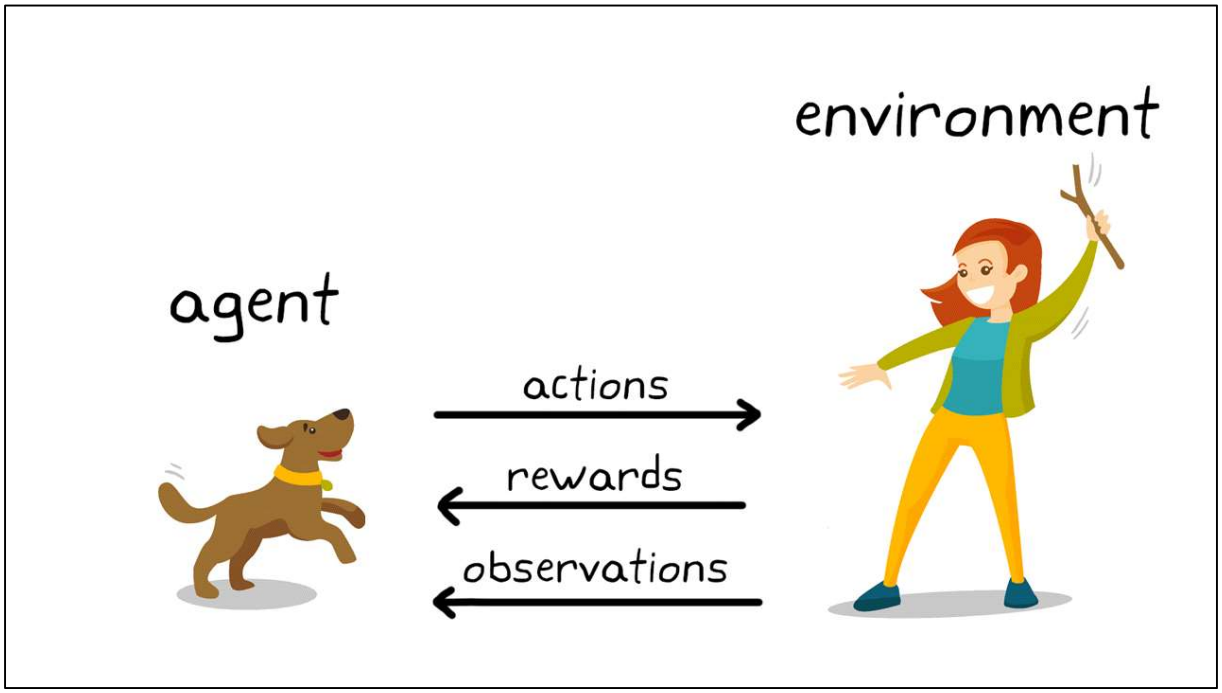
I. nadzirano učenje (eng. *supervised learning*),
II. nenadzirano učenje (eng. *unsupervised learning* i
III. podržano/ojačano učenje (eng. *reinforcement learning*).

Kao što samo ime govori, nadzirano učenje temelji se na nadzoru. To znači da se stroj trenira pomoću „označenog“ skupa podataka, a na temelju treninga se predviđa izlaz. U početku se stroj trenira s ulazom i određenim izlazom, a zatim se od njega očekuje da predvidi izlaz pomoću skupa testnih podataka. Neke od stvarnih aplikacija nadziranog učenja su procjena rizika, otkrivanje prijevara, filtriranje neželjene pošte, segmentacija slike i prepoznavanje govora. Kod nenadziranog učenja situacija je u potpunosti obrnuta. Stroj se obučava pomoću neoznačenog skupa podataka i daje izlazne podatke bez ikakvog nadzora. Glavni cilj algoritma nenadziranog učenja je grupirati ili kategorizirati nerazvrstani skup podataka prema sličnosti, obrascima i razlikama. Strojevi dobivaju upute da iz ulaznog skupa podataka pronađu skrivene uzorke. Neki od primjera nenadziranog učenja su mrežna analiza, sustav preporuka te detekcija anomalija [3]. S obzirom da je podržano učenje glavni temelj ovog rada, ono će biti detaljnije opisano u nastavku.

2.1. Podržano učenje

Kao što je prethodno rečeno, podržano učenje spada u potpodručje strojnog učenja. Definira se kao tehnika u kojoj agent izvodeći radnje i gledajući rezultate, uči ponašati se u određenom okruženju. Za razliku od nadziranog učenja, agent uči automatski, koristeći povratne informacije bez ikakvih označenih podataka. Drugim riječima, agent uči samo na temelju vlastitog iskustva, odabire akciju i dobiva povratnu informaciju u obliku stanja (ili opažanja) i nagrada. Za svaku dobru akciju agent dobiva pozitivnu povratnu informaciju, a za svaku lošu akciju dobiva negativnu povratnu informaciju ili kaznu. Podržano učenje rješava specifičnu vrstu problema gdje je donošenje odluka sekvencijalno, a cilj dugoročan. Kao jedan od primjera iz stvarnog života može se istaknuti treniranje psa. Cilj učenja u ovom slučaju je treniranje psa (agenta) da izvrši zadatak unutar okruženja, koje uključuje okolinu psa i trenera. Pas reagira na način da poduzima određene akcije.

Ako je radnja blizu željenog ponašanja, pas će dobiti nagradu (poslastica ili igračka), a u suprotnom neće dobiti ništa. Na taj način će u budućnosti lakše odlučivati koju radnju je potrebno poduzeti [4].



Slika 1. Prikaz primjera strojnog učenja u stvarnom životu [4]

Za bolje razumijevanje podržanog učenja važno je definirati dvije glavne stvari. Jedna od njih je okoliš, prostor poput sobe, labirinta ili igrališta, dok je druga bitna stvar agent, u ovom slučaju inteligentni robot.

Osim okoline i agenta postoje četiri podelementa strojnog učenja:

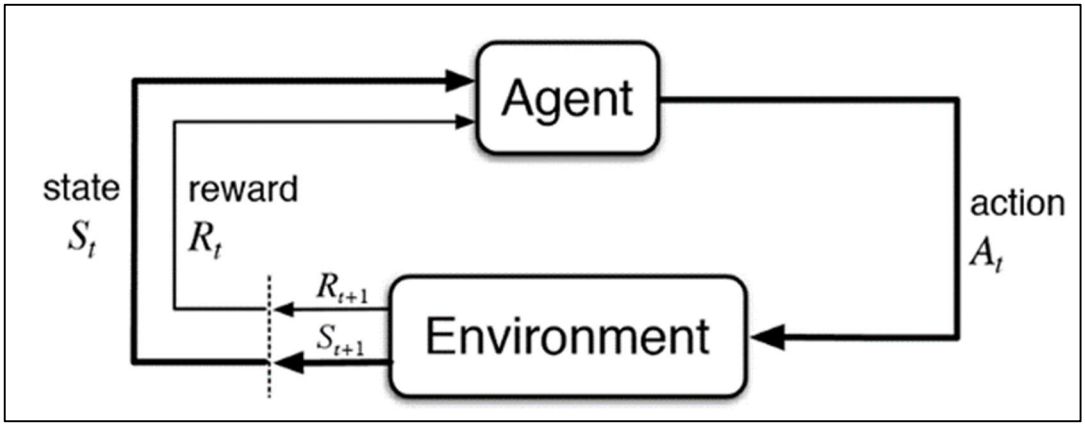
I. politika (eng. *policy*),
II. signal za nagradu (eng. *reward signal)*,
III. funkcija vrijednosti (eng. *value function*) i
IV. model okoline (eng. *model of environment*).

Politika se definira kao način na koji se agent ponaša u određenom trenutku. Ona preslikava percipirana stanja okoliša na akcije koje su poduzete u tim stanjima. S obzirom da sama može definirati ponašanje agenta, smatra se ključnim elementom podržanog učenja. Ponašanje agenta se u nekim slučajevima može opisati kao jednostavna funkcija ili pregledna tablica, dok u drugim slučajevima može uključivati opće izračunavanje kao proces pretraživanja [5].

Signal nagrade definiran je kao cilj problema podržanog učenja. U svakom stanju okolina agentu šalje trenutni signal, a taj signal je poznat kao signal nagrade. Jedini cilj agenta je maksimalno povećati ukupnu nagradu koju dugoročno prima. Signal nagrade tako definira koji su dobri i loši događaji za agenta. Također, signal nagrade može biti primarna osnova za promijenu politike, na primjer, ako radnja koju je odabrao agent dovede do niske nagrade, tada se politika može promijeniti kako bi se u budućnosti odabrale bolje radnje [5].

Dok signal nagrade pokazuje što je dobro u neposrednom smislu, funkcija vrijednosti određuje što je dobro u dugoročnom smislu. Odnosno, funkcija vrijednosti označava ukupan iznos nagrade koju agent može očekivati da će akumulirati u budućnosti. Na primjer, stanje može uvijek donijeti nisku trenutnu nagradu, ali još uvijek ima visoku vrijednost jer ga redovito slijede druga stanja koja donose visoke nagrade. Bitno je definirati povezanost između nagrade i vrijednosti, jer bez nagrade vrijednost ne bi ni postojala [5].

Četvrti i posljednji element sustava podržanog učenja je model koji oponaša ponašanje okoline. Uz pomoć modela može se zaključiti kako će se okolina ponašati. S obzirom na stanje i radnju, model može predvidjeti rezultirajuće sljedeće stanje i sljedeću nagradu [5].



Slika 2. Povratna petlja akcija-nagrada kod podržanog učenja [18]

2.2. Višeruki razbojnik

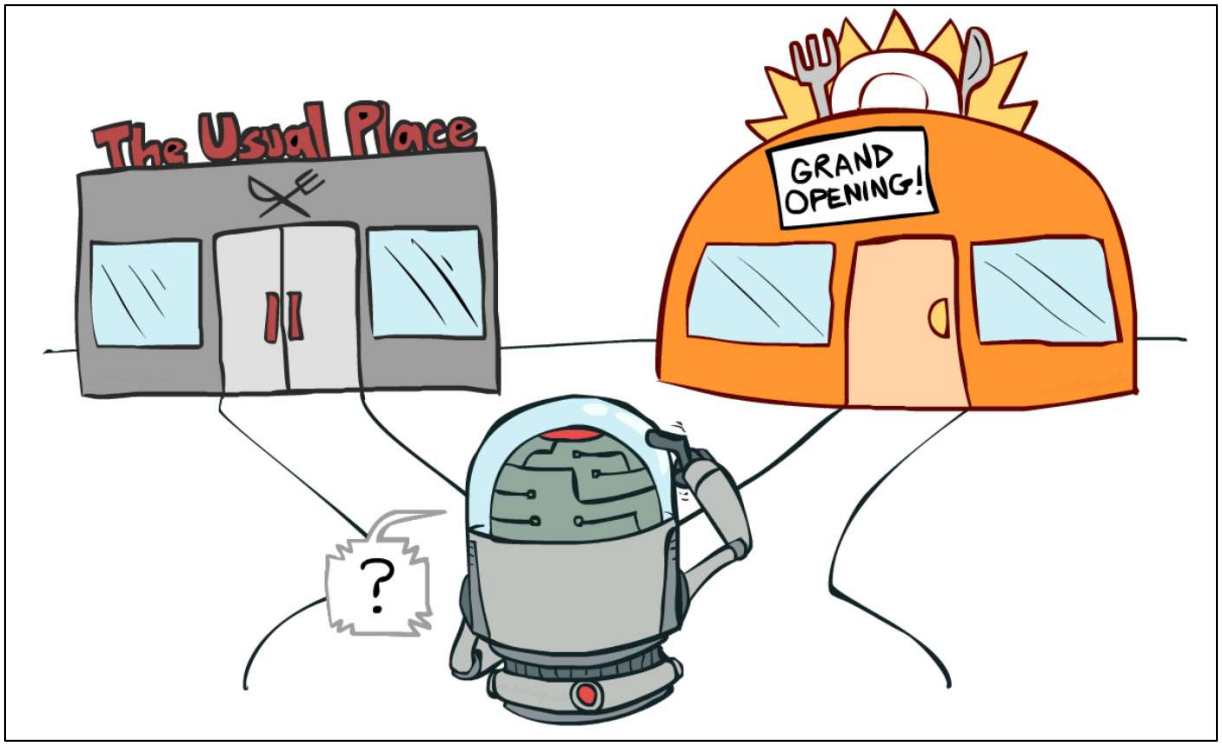
Problem višerukog razbojnika jedan je od osnovnih primjera podržanog učenja. Izraz dolazi od automata u kockarnicama, gdje je jednoruki agent označen kao jedan automat. Algoritam višerukog razbojnika nastao je iz hipotetskog eksperimenta u kojem osoba mora birati između

više akcija, od kojih svaka ima nepoznatu isplatu. Glavni cilj je nizom izbora odrediti najbolji ili najprofitabilniji ishod.

Slika 3. Automati u kockarnicama kao predstavnici višerukih razbojnika [6]

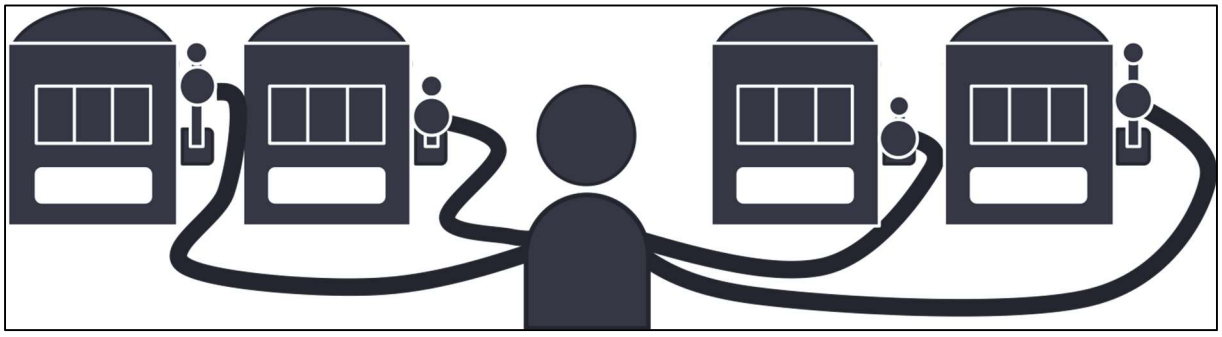
U svrhu detaljnijeg objašnjena problematike višerukog razbojnika, potrebno je odrediti razliku između iskorištavanja (eng. *exploitation*) i istraživanja (eng. *exploration*). Iskorištavanje se definira kao pohlepan pristup u kojem agenti pokušavaju dobiti što više nagrada koristeći procijenjenu, ali ne i stvarnu vrijednost. U ovoj tehnici agenti donose najbolju odluku na temelju trenutnih informacija. Za razliku od iskorištavanja, u tehnikama istraživanja agenti se umjesto na dobivanje više nagrada, primarno fokusiraju na poboljšanje svog znanja o svakoj radnji. Točnije, u ovoj tehnici agenti rade na prikupljanju većeg broja informacija kako bi mogli donijeti što bolju odluku. Na taj način ostvaruju dugoročnu korist. Kao primjer se može uzeti scenarij online odabira restorana za narudžbu hrane. Prva opcija je da se odabere najdraži restoran iz kojeg se naručivalo u prošlosti. Budući da su poznate samo trenutne informacije o tom određenom restoranu, taj postupak se naziva iskorištavanje. U drugom slučaju postoji opcija istraživanja novog restorana. Na taj način se otkrivaju nove vrste i okusi hrane. Postupkom iskorištavanja može se osigurati siguran ishod, a to je provjereno dobra hrana, no istraživanjem se pojavljuje mogućnost pronalaska još bolje hrane. Opisana problematika

između iskorištavanja i istraživanja je glavni problem u algoritmu višerukog razbojnika [7].



Slika 4. Dvojba između istraživanja i iskorištavanja [8]

Po uzoru na prethodno objašnjeni primjer, problem višerukog razbojnika je koncipiran na način da se agent (robot) suočava s nizom kockarskih automata, od kojih svaki ima svoju vjerojatnost isplate. Primarni cilj agenta je odrediti na kojem automatu će igrati kako bi maksimalno povećao svoju ukupnu isplatu. Kako bi to uspio od iznimne važnosti je uravnotežiti iskorištavanje automata koji su već igrani s istraživanjem novih kockarskih automata.



Slika 5. Prikaz agenta kako isprobava više automata (razbojnika) [9]

Osim teorijskog opisa višerukog razbojnika, potrebna je i matematička podloga, koja je glavni polaz u rješavanju problema višerukog razbojnika. Kako bi formule bile razumljive, neophodno je definiranje uobičajenih pojmova koji se koriste kod algoritma višerukog bandita.

Prvi pojam je akcija (eng. *action*), odabir između dostupnih opcija i izbor jedne od njih. U problemu s višerukim banditom to bi bio odabir i igranje jednog od niza automata. Kod problema s razbojnikom radnje se poduzimaju u diskretnim vremenskim koracima, to jest jedna akcija se poduzima za drugom i, u ovom problemu, postoji fiksan broj ukupnih radnji. Radnja poduzeta u vremenskom koraku *t* označuje se kao A_t [9].

Svaki put kad se poduzme neka radnja dobiva se nagrada (eng. *reward*). U problemu višerukog razbojnika to je iznos novca koji je osvojen na kockarskom automatu. Nagrada dobivena za poduzimanje određene radnje je slučajna vjerojatnost, izvučena iz osnovne distribucije vjerojatnosti specifične za tu radnju. Odnosno, svaki put kad se poduzme radnja, vraćena nagrada može imati drugačiju vrijednost. Dobivanje nagrade je glavna značajka koja definira problem višerukog razbojnika te poboljšava i usmjerava učenje. Nagrada dobivena u vremenskom koraku *t*, nakon poduzimanja akcije A_t označava se kao R_t [9].

Uz akciju i nagradu, broj dostupnih radnji (eng. *number of available actions*) je također važan pojam. Označava se slovom *k*. Kod problema višerukog razbojnika broj dostupnih radnji je definiran kao ukupan broj automata koji se mogu odabrat. Iz tog razloga se problem višerukog razbojnika zna pojaviti i pod imenom „K-armed bandit problem“ [9].

Svaka od dostupnih akcija *k* ima očekivanu nagradu (eng. *expected reward*), pri čemu se izraz „očekivana“ odnosi na srednju vrijednost koja bi se dobila u slučaju da se radnja ponovi više puta. Očekivana vrijednost akcije označava se oznakom $\mathbb{E}$ [9].

Očekivana nagrada za određenu radnju poznata je kao vrijednost (eng. *value*) te radnje i označena je kao *q(a)* gdje je *a* određena radnja odabrana u vremenskom koraku *t* (tj. $A_t = a$). Vrijednost radnje se definira pomoću formule koja glasi:

$$q(a) = \mathbb{E}[R_t|A_t].$$

Vrijednost radnje *a*, kako je prikazano formulom, jednaka je očekivanoj (srednjoj) vrijednosti nagrade, gdje je radnja *a* odabrana u vremenskom koraku *t*. Točnije, ako se radnja *a* ponovi više puta dobit će se prosječna vrijednost nagrade koja će s vremenom postati prava vrijednost [9] [10].

Kada bi prava vrijednost nagrade bila poznata, problem višerukog razbojnika bi bio jednostavno riješen. Ne bi bilo potrebe za istraživanjem jer bi se iskorištavao najbolji automat koji bi davao najviše novaca. S obzirom da je u ovom slučaju prava vrijednost nepoznata, istraživanje je prijeko potrebno. Jedan od jednostavnijih načina procjene prave vrijednosti je izračunavanje

prosječne vrijednosti nagrada koje su bile vraćene za tu određenu radnju. Dakle, procijenjena vrijednost $Q_t(a)$ u vremenskom koraku t izračunava se pomoću formula, a jedna od njih je:

$$Q_t(a) = \frac{1}{n}\sum_{i=1}^{n} R_i\,,$$

gdje je n broj puta kada je radnja a poduzeta (prije vremena t), a R_i je nagrada dobivena u svakom od vremenskih koraka. U slučaju kada nazivnik ide u beskonačnost, prema zakonu velikih brojeva, $Q_t(a)$ konvergira u $q(a)$. Ovo se naziva metodom uzorka prosjeka (eng. *sample-average method*) jer je svaka procjena prosjek uzorka relevantnih nagrada. To je samo jedan od načina za procjenu akcijskih vrijednosti, iako jednostavan, nije nužno i najbolji [10].

3. IZVEDBA ZADATKA U PYTHONU

U idućim poglavljima razvijen je praktični dio rada, to jest izvedba zadatka u programskom jeziku Python. U prvom dijelu definiran je programski jezik Python te korišteni modul random i biblioteka NumPy. U drugom dijelu je problem višerukog razbojnika riješen na više načina. Svaki od načina ima svoje ime agenta te je posebno opisan i implementiran u Python pomoću koda.

3.1. Python

Python je interpretirani, objektno orijentirani programski jezik visoke razine s dinamičkom semantikom koju je razvio Guido van Rossum. Izvorno je objavljen 1991. godine. Poznat je po svojoj jednostavnosti, lakoći korištenja i svestranosti. Zbog svoje lako razumljive i jednostavne sintakse, smatra se idealnim jezikom za početnike. S obzirom da je interpretirani programski jezik, potreba za kompilacijom nije potrebna jer se kod izvršava red po red, a ne odjednom. Kompatibilan je s raznim platformama, a neke od njih su Linux, Windows, Mac, Raspberry Pi i mnoge druge. Također podržava više načina programiranja, uključujući strukturirano, objektno orijentirano i funkcionalno. Jedna od glavnih prednosti Python-a je njegova opsežna standardna biblioteka koja je dostupna svima i pomoću koje se na brzi i lagan način rješavaju zadaci bez opsežnog pisanja koda [11]. Python se često koristi za izradu web stranica i softvera, automatizaciju zadataka i provođenje analize podataka. Opće je namjene, što znači da se može koristiti za stvaranje niza različitih programa i nije specijaliziran za bilo kakve specifične probleme. Njegova svestranost, zajedno s jednostavnošću, učinila ga je jednim od najčešće korištenih programskih jezika u posljednje vrijeme. Istraživanje koje je provela analitičarska tvrtka RedMonk pokazalo je da je Python bio drugi najpopularniji programski jezik među programerima u 2021. godini [12].

3.1.1. NumPy

NumPy je Python biblioteka koja se koristi za izračunavanje i obradu višedimenzionalnih elemenata polja (eng. *array*). Osim toga, koristi se za operacije s matricama, rad u području linearne algebre te kod Fourierovih transformacija. Izraz „NumPy“ potječe od kratice za

Numerički Python (eng. *numerical python*). Travis Oliphant je stvorio NumPy paket 2005. godine ubacivanjem značajki pretka modula Numeric u drugi modul Numarray. NumPy biblioteka zbog svoje brzine pruža prikladan i učinkovit način za rukovanje s velikom količinom podataka. U današnje vrijeme se NumPy u kombinaciji sa SciPy i Mat-plotlib koristi kao zamjena za programski jezik Matlab [13].

3.1.2. Random

Random je ugrađeni modul u Python-u koji pruža razne funkcije za generiranje pseudo-slučajnih brojeva. Ovi brojevi nisu potpuno slučajni, već se generiraju pomoću deterministčkog algoritma koji proizvodi brojeve koji samo izgledaju slučajno. Modul random se koristi za generiranje slučajnih cijelih brojeva, decimalnih brojeva i slučajnih sekvenci. Također pruža funkcije za postavljenje generatora slučajnih brojeva, što omogućuje generiranje istog niza slučajnih brojeva svaki put [14].

3.2. Algoritmi za rješavanje problema višerukog razbojnika

Budući da rješenje ovog problema sadrži nizove i slučajne brojeve, na početku koda je bilo potrebno implementirati prethodno objašnjenu biblioteku NumPy te modul random. Dio koda u kojem je to napravljeno je prikazan na slici [slika 6.].

```
import random
import numpy as np
```

Slika 6. Implementacija biblioteke NumPy i metode random

Nakon implementacije potrebne biblioteke i modula, kreirana je klasa *Bandit*, predložak koda za stvaranje objekata. Klasa se koristi kako bi se povezane stvari držale zajedno te se u Python-u stvara pomoću riječi class. Kod u kojem je definirana klasa prikazan je na slici [slika 7.].

```
class Bandit:
    def __init__(self, payoff_probs):
        self.actions = range(len(payoff_probs))
        self.pay_offs = payoff_probs

    def sample(self, action):
        selector = random.random()
        return 1 if selector <= self.pay_offs[action] else 0
```

Slika 7. Definiranje klase

Navedeni dio koda predstavlja definiciju klase za višerukog razbojnika, gdje svaka „ruka“ ili akcija ima različitu vjerojatnost davanja nagrade. Klasa sadrži metodu konstruktora *__init__* koja kao ulaz uzima popis vjerojatnosti isplate (payoff_probs). U klasi su sadržana dva atributa, od kojih je jedan *self.actions*. On se inicijalizira rasponom cijelih brojeva od 0 do *len(payoff_probs) – 1* te se na taj način stvara popis indeksa radnji koji se kasnije koriste za odabir akcije. Atribut *self.pay_offs* inicijalizira se na popis unosa vjerojatnosti isplate. Metoda *sample* uzima radnju kao ulaz i simulira nagradu za tu radnju na temelju vjerojatnosti. Pomoću funkcije *random.random()* generira slučajni broj između 0 i 1 te ga uspoređuje s vjerojatnosti dobivanja nagrade za odabranu radnju. Ako je slučajni broj manji ili jednak vjerojatnosti dobivanja nagrade za odabranu radnju, tada metoda vraća 1 (nagrada je dobivena), inače vraća 0 (nema nagrade). Dakle, metoda *sample* generira binarnu nagradu za danu akciju, gdje je vjerojatnost dobivanja nagrade određena odgovarajućom vrijednošću u listi *self.pay_offs*. Ova metoda je ključna komponenta raznih algoritama pojačanog učenja koji rješavaju problem višerukog razbojnika. Iako jako bitna, metoda *sample* je samo osnovni i početni dio rješenja. Glavni problem je pronalazak balansa između iskorištavanja i istraživanja.

U nastavku ovog rada bit će definirani različiti algoritmi za rješavanje problema višerukog razbojnika. Uz određenu ravnotežu iskorištavanja i istraživanja, svaki algoritam će imati različit stupanj uspjeha.

3.2.1. Pohlepni agent s određenim brojem iteracija istraživanja

Jedan od najpoznatijih načina rješavanja problema višerukog razbojnika je upotreba pohlepnog agenta (eng. *greedy agent*), algoritma koji uvijek odabire radnju s najvećom procijenjenom nagradom. Takav algoritam radi korak po korak i uvijek bira akcije koje daju neposrednu korist.

Drugim riječima, bira „lokalno optimalno rješenje“, ne razmišljajući o budućim posljedicama. Nikad ne poništava raniju odluku, čak iako su posljedice loše. Ovakva tehnika je najprikladnija za sagledavanje trenutne situacije, no nije baš dobar izbor za akcije u budućnosti [15].

U ovom dijelu rada bit će opisan pohlepni agent s određenim brojem iteracija istraživanja. Algoritam je koncipiran na način da agent određen broj iteracija nasumično odabire automat te se nakon toga odluči za onaj s najvećim dobitkom i svaki idući put igra samo njega. U nastavku koda u Python-u se definira funkcija *explore_greedy agent* koja ima četiri argumenta. Kod je prikazan na slici [slika 8.].

```
def explore_greedy_agent(bandit, iterations, initial_rounds=10, random_agent=False):

    pay_offs = dict()
    best_action = -1
    actions = []
    rewards = []
```

Slika 8. Definiranje funkcije za pohlepnog agenta s određenim brojem iteracija istraživanja

Prvi argument je *bandit*, instanca klase *Bandit* koja definira problem višerukog razbojnika. Drugi argument je *iterations*, cijeli broj koji definira broj puta kada će automat biti odigran. Idući po redu je izborni argument cijelog broja koji navodi koliko će puta algoritam nasumično odabrati akciju prije nego što se prebaci na pohlepnu strategiju. Posljednji argument je boolean koji određuje treba li se algoritam ponašati kao nasumični agent. Funkcija inicijalizira prazan riječnik (eng. *dictionary*) pod nazivom *pay_offs* koji će se koristiti za praćenje povijesti nagrada za svaku radnju. Rječnik je Python-ova implementacija strukture podataka koja je općenitije poznata kao asocijativni niz. Riječnik se sastoji od zbirke parova ključ-vrijednost (eng. *key-value pairs*). Ključ se koristi za pristup odgovarajućim vrijednostima. Ogromna prednost rječnika je ta da su vrlo učinkoviti za dohvaćanje, dodavanje i brisanje elemenata [16]. Osim riječnika, funkcija inicijalizira prazan popis akcija i nagrada te varijablu *best_action* na -1.

Nakon početnih par linija koda potrebno je napisati ostatak koda koji će omogućiti agentu nasumični broj odabira akcija (u ovom slučaju 10) te određivanje najbolje akcije. To se u Python-u najlakše implementira pomoću *for* i *if* petlji. Kod je prikazan na slici [slika 9.].

```
if random_agent:
    initial_rounds = iterations

for t in range(iterations):
    # for the initial rounds pick a random action
    if t < initial_rounds:
        a = random.choice(bandit.actions)
        r = bandit.sample(a)

        # update rewards
        if a in pay_offs:
            pay_offs[a].append(r)
        else:
            pay_offs[a] = [r]
    #  otherwise choose the best action so far
    else:
        if (best_action == -1):
            # check for the action with the best average payoff
            mean_dict = {}
            for key,val in pay_offs.items():
                mean_dict[key] = np.mean(val)
            best_action = max(mean_dict, key=mean_dict.get)
        a = best_action

        r = bandit.sample(a)

    actions.append(a)
    rewards.append(r)

return actions, rewards
```

Slika 9. Prikaz koda koji pomoću nasumičnih akcija odabire onu najbolju

Pomoću prve *if* petlje postavljen je uvjet za nasumičnog agenta, to jest varijabla *initial_rounds* je ažurirana na vrijednost iteracija. Sve dok je taj uvjet zadovoljen agent će pomoću metode *random.choice()* istraživati, odnosno nasumično odabirati akcije. U ovom slučaju će to biti deset akcija. Zatim, pomoću metode *sample()*, nagrada se postavlja na vrijednost 1 u dobitnom te na vrijednost 0 u negativnom slučaju. Zatim se u rječnik *pay_offs* ažurira radnja *a* kojoj je pridružena dobivena nagrada *r*. Na taj način se stvara novi par ključ-vrijednost gdje je radnja *a* ključ, a vrijednost je nagrada *r*. U slučaju da radnja *a* već postoji u riječniku pay_offs, metodom *append()* se nagrada *r* dodaje na popis nagrada vezanih uz radnju *a*. Nakon deset nasumičnih akcija agent odabire onu najbolju te ju iskorištava u narednim iteracijama. Petljom *if (best_action == -1)* se provjerava je li ovo prva runda iskoštavanja i ako je, kreira se rječnik *mean_dict*. Rječnik se pomoću vitičastih zagrada stvara kao prazan rječnik te se kasnije pomoću metode *np.mean* popunjava prosječnim vrijednostima isplate. Na kraju se funkcijom *max()*

vraća ključ s maksimalnom vrijednošću u rječniku *mean.dict*. Ključ predstavlja radnju s najvećom procjenom akcijske vrijednosti. Tu radnju tada odabire agent i svaki idući put samo nju iskorištava.

3.2.2. Nasumični agent

Nasumični agent (eng. *random agent*) je algoritam koji cijelo vrijeme istražuje, odnosno nasumično odabire akcije. S obzirom da samo istražuje, a ne koristi otkriveno znanje smatra se naivnim. Takav agent će se u Python implementirati pomoću prethodno opisanog algoritma za pohlepnog agenta s određenim brojem istraživanja. Na slici [slika 10.] je prikazan dio koda.

```
def random_agent(bandit, iterations):
    return explore_greedy_agent(bandit, iterations, random_agent=True)
```

Slika 10. Implementacija random algoritma u Python

Funkcija *random_agent* ima dva ulazna argumenta, a to su *bandit* i *iterations*. Ona poziva funkciju *explore_greedy agent* s predhodno spomenutim argumentima te parametar *random_agent* postavlja na True. To znači da funkcija *explore_greedy agent* koristi nasumičnu strategiju za istraživanje početnih akcija i nasumično odabire akcije za sve iteracije. Funkcija *random_agent()* vraća listu akcija i nagrada koje je dala funkcija *explore_greedy_agent()*. Te se liste mogu koristiti za analizu učinka agenta na problemu višerukog razbojnika.

3.2.3. Epsilon-pohlepni agent i njegove varijacije

Epsilon-pohlepni agent (eng. *epsilon-greedy agent*) je popularna strategija koja rješava problem višerukog razbojnika. Ovaj algoritam bira između istraživanja, gdje se akcija odabire s vjerojatnošću *epsilon*, te iskorištavanja, gdje se radnja odabire s vjerojatnošću *1-epsilon*. Na primjer, ako se vrijednost epsilona postavi na 0.1, oko 10% radnji bit će odabrano nasumično (istraživanje), dok će za ostalih 90% agent odabrati najbolju radnju od prethodno učinjenih (eksploatacija) [10]. Implementacija ovog algoritma u Python-u je izvršena na sličan način kao

i kod pohlepnog agenta s određenim brojem iteracija istraživanja. Na slici [slika 11.] je prikazan početni dio koda.

```
def epsilon_greedy_agent(bandit, iterations, epsilon=0.2, initial_rounds=10, decay=1, optimistic=False):

    pay_offs = dict()
    actions = []
    rewards = []
```

Slika 11. Definiranje funkcije za epsilon-pohlepni algoritam

U usporedbi s funkcijom *explore-greedy agent* kod ove funkcije dodani su određeni argumenti. Jedan od njih je *epsilon*, broj između 0 i 1 koji predstavlja vjerojatnost poduzimanja nasumične radnje. Argument *decay* je također broj između 0 i 1 te predstavlja faktor pada epsilon vrijednosti. Vrijednost epsilona se množi s decay faktorom nakon svake iteracije. Zadnji argument je *optimistic*, boolean koji određuje početne prosječne isplate svih radnji. Također, kao što je prije spomenuto, funkcija inicijalizira prazan rječnik te prazne liste akcije i nagrade.

Kada su početne linije koda definirane potrebno je nastaviti implementaciju u Python-u. Nastavak koda prikazan je na slici [slika 12.].

```
if optimistic:
    pay_offs = {a:[1] for a in bandit.actions}

for t in range(iterations):
    # sometimes randomly pick an action to explore
    if random.random() < epsilon or t < initial_rounds:
        a = random.choice(bandit.actions)
    # otherwise choose the best one thus far
    else:
        new_dict = {}
        for key,val in pay_offs.items():
            new_dict[key] = np.mean(val)
        a = max(new_dict, key=new_dict.get)

    r = bandit.sample(a)

    # update rewards
    if a in pay_offs:
        pay_offs[a].append(r)
    else:
        pay_offs[a] = [r]

    epsilon *= decay

    actions.append(a)
    rewards.append(r)

return actions, rewards
```

Slika 12. Impementacija epsilon-pohlepnog agenta

Svrha prve *if* petlje je postavljanje početnih vrijednosti isplate za svaku akciju u rječniku pay_offs. Ako je argument *optimistic* True, početne vrijednosti isplate postavljaju se na vrijednost 1, dok su za False sve početne vrijednosti isplate 0. Cilj ovog algoritma je da agent određeni broj iteracija samo istražuje, a nakon toga počne koristiti pohlepnu strategiju s vjerojatnošću *1-epsilon* i istraživačku strategiju s vjerojatnošću *epsilon*. Takav način razmišljanja se postigao pomoću *if random.random() < epsilon or t < initial_rounds*, linije koda koja sadrži dva uvjeta. Prvih deset iteracija agent će samo istraživati, a ostatak će ili istraživati ili iskorištavani, ovisno o slučajnom broju koji se generira metodom *random.random()*. U slučaju kada će nasumični broj biti veći od *epsilona* algoritam će tražiti ključ s maksimalnom vrijednošću u rječniku *new_dict*.

Prethodno definirana verzija epsilon-pohlepnog agenta može se poboljšati na određene načine, a jedan od njih je promjena *decay* parametra. Na primjer, ako se *decay* parametar postavi na 0.99, vrijednost *epsilona* će u svakoj iteraciji biti smanjena za 0.99. U Pythonu se to postiže linijom *epsilon* *= *decay*. Na taj način se osigurava istraživanje koje je s vremenom sve manje, što je potpuno smisleno jer agent stječe znanje i njegova potreba za dodatnim istraživanjem postaje sve manja. Ime ovog algoritma je „propadajući epsilon-pohlepni agent (eng. *epsilon greedy with decay*)“ i njegova implementacija u Python-u je prikazana na idućoj slici.

```
def decaying_epsilon_greedy_agent(bandit, iterations,
  epsilon=0.2, initial_rounds=1, decay=0.99):

    return epsilon_greedy_agent(bandit, iterations, epsilon=epsilon,
     initial_rounds=initial_rounds, decay=decay)
```

Slika 13. Implementacija epsilon-pohlepnog agenta s propadanjem

U kodu je pozvana funkcija *epsilon_greedy agent* u kojoj su promijenjeni parametri *decay* i *initial_rounds*. S obzirom na to da agent već ima neko prethodno znanje, parametar *initial_rounds* je postavljen na vrijednost 1.

Postavljanjem početnih vrijednosti nagrade za svaku radnju na najveću moguću (na primjer na 1) dobiva se još jedna varijanta epsilon-pohlepnog agenta. Ovakav algoritam se zove optimstični epsilon-pohlepni agent (eng. *optimistic epsilon-greedy agent*). U Python-u je implementiran pomoću prethodno objašnjenog parametra *optimistic* koji se u ovom slučaju postavlja na True. Kod je prikazan na slici [slika 14.].

```
def optimistic_greedy_agent(bandit, iterations, epsilon=0.2,
  initial_rounds=1, decay=0.99):

    return epsilon_greedy_agent(bandit, iterations, epsilon=epsilon,
     initial_rounds=initial_rounds, decay=decay, optimistic=True)
```

Slika 14. Implementacija optimističnog epsilon-pohlepnog agenta

Na ovaj način agent preferira akcije koje nije toliko isprobao, jer misli da svaka „ruka“ ima visoku nagradu. Drugim riječima, potiče se ranije istraživanje i brži pronalazak najbolje akcije.

3.2.4. Optimalni agent

Optimalni agent (eng. *optimal agent*) je nerealni algoritam koji uvijek odabire akciju s najvećim prosječnim dobitkom. Pretpostavka je da agent ima podatke isplata za svaku akciju te se iz tog razloga naziva „nerealnim“. S obzirom na to da ovakav algoritam uvijek zna što je najbolje poduzeti, koristi se kao referentna vrijednost za usporedbu učinka ostalih agenata. Implementacija ovog algoritma u Python-u je prikazana na slici [slika 15.].

```
def optimal_agent(bandit, iterations):

    a = np.argmax(bandit.pay_offs)
    actions = []
    rewards = []

    for _ in range(iterations):
        r = bandit.sample(a)
        actions.append(a)
        rewards.append(r)

    return actions, rewards
```

Slika 15. Definiranje funkcije optimalnog agenta

Linija koda *a = np.argmax(bandit.pay_offs)* pronalazi indeks radnje koja ima najveći očekivani dobitak, a funkcija *optimal_agent* izračunava nagradu za tu radnju te, kao i u prethodnim primjerima, zapisuje akciju i nagradu u prazne liste.

3.2.5. Agent s gornjom granicom povjerenja

Jedan od agenata koji se zna nositi s istraživanjem i iskorištavanjem je agent s gornjom granicom povjerenja (eng. *upper confidence bound agent (UCB)*). Takav algoritam za iskorištavanje koristi prosječnu nagradu svih razbojnika, baš kao što je i slučaj kod epsilon-pohlepnog agenta. Bitna razlika je to da kod istraživanja ovaj agent koristi parametar *c*. Na primjer, ako je agent na vremenskom koraku 100 i određenu radnju je odabrao samo jednom, parametar *c* postaje visok i osigurava da agent odabire tu radnju. Na taj način agent prirodno istražuje radnje koje nije često odabirao, ali također i iskorištava. Na slici [slika 16.] je prikazan početni dio koda ovog algoritma.

```
def ucb_agent(bandit, iterations, c=0.3):

    pay_offs = dict()
    no_of_times_action_chosen = {a:0 for a in bandit.actions}
    q_values = {a:0 for a in bandit.actions}
    actions = []
    rewards = []
```

Slika 16. Definiranje funkcije UCB agenta

Osim praznog rječnika *pay_offs* te praznih listi nagrada i akcija, funkcija *ucb_agent* također inicijalizira prazne rječnike *no_of_times_action_chosen* i *q_values*. Rječnik *no_of_times_action_chosen* koristi se za praćenje koliko je puta UCB agent odabrao pojedinu radnju, a rječnik *q_values* je bitan za ažuriranje Q-vrijednosti, odnosno očekivane nagrade za svaku akciju. U nastavku je prikazan daljnji tijek koda.

```
for a in bandit.actions:
    for _ in range(2):
        r = bandit.sample(a)
        if a in pay_offs:
            pay_offs[a].append(r)
        else:
            pay_offs[a] = [r]
        actions.append(a)
        rewards.append(r)
        no_of_times_action_chosen[a] += 1
        q_values[a] = sum(pay_offs[a])/no_of_times_action_chosen[a]

for t in range(2*len(bandit.actions), iterations):
    values = [q_values[a] + c*np.sqrt(np.log(t)/no_of_times_action_chosen[a]) for a in bandit.actions]
    a = np.argmax(values)
    r = bandit.sample(a)
    pay_offs[a].append(r)
    actions.append(a)
    rewards.append(r)

    # update q values and no_of_times_per_action
    no_of_times_action_chosen[a] += 1
    q_values[a] = sum(pay_offs[a])/no_of_times_action_chosen[a]

return actions, rewards
```

Slika 17. Prikaz koda koji definira UCB agenta

Prvi blok koda govori agentu da na početku algoritma svaku radnju odigra dva puta. Ovo je potrebno kako bi se izbjegla greška dijeljenja s nulom prilikom izračunavanja srednjih vrijednosti za svaku akciju. U *if* petlji se, kao i u prethodnim primjerima, zapisuju parovi ključ-vrijednost u rječnik *pay_offs*. Zatim se kod dodavanja radnji i nagrada u prazne liste, također ažurira rječnik *no_of_times_action_chosen*, koji označava da je radnja još jednom odabrana. Linijom koda *q_values[a] = sum(pay_offs[a])/no_of_times_action_chosen[a]* se ažurira rječnik *q_values* na način da se izračunava srednja srednja vrijednost nagrade za pojedinu radnju. Drugi blok koda predstavlja glavnu petlju za UCB algoritam. Algoritam se izvodi za fiksan broj ponavljanja te odabire akcije na temelju balansa između istraživanja i iskorištavanja. Druga linija koda u drugom bloku izračunava gornju granicu pouzdanosti za svaku radnju u svakom vremenskom koraku. Gornja granica pouzdanosti je mjera koja određuje koliko je algoritam nesiguran u određivanju prave vrijednosti svake akcije. To potiče agenta da odabire radnje koje imaju potencijal za visoke nagrade, ali nisu puno puta isprobane. To su radnje za koje agent nije siguran u njihovu pravu vrijednost. Nakon toga, agent pomoću funkcije *np.argmax* iz liste *values* odabire indeks radnje s maksimalnom gornjom granicom pouzdanosti. Metoda *sample* tada daje binarnu vrijednost za odabranu radnju te se rječnik *pay_offs* ažurira. Također se ažurira broj odabira radnje *a* te njena srednja vrijednost.

3.2.6. Bernoulli Thompson agent

Posljednji algoritam za rješavanje problema višerukog razbojnika je Bernoulli Thompson agent. Ovaj agent ima potpuno drugačiji pristup od svih prethodno objašnjenih. Za svaki automat, agent prati broj uspjeha α te broj neuspjeha β. Te vrijednosti koristi za stvaranje modela vjerojatnosti za svakog razbojnika. Na slici [slika 18.] je prikazan kod za ovaj algoritam.

```
def thompson_agent(bandit, iterations):

    pay_offs = dict()
    actions = []
    rewards = []

    alphas, betas = {a:1 for a in bandit.actions}, {a:1 for a in bandit.actions}

    for t in range(iterations):
        a = np.argmax([np.random.beta(alphas[a], betas[a]) for a in bandit.actions])
        r = bandit.sample(a)
        if a in pay_offs:
            pay_offs[a].append(r)
        else:
            pay_offs[a] = [r]

        actions.append(a)
        rewards.append(r)

        alphas[a] += r
        betas[a] += (1-r)

    return actions, rewards
```

Slika 18. Prikaz koda za Bernoulli Thompson-ov algoritam

Budući da algoritam počinje bez prethodnog znanja o nagradama, početna vrijednost u α i β rječniku se inicijalizira na 1. To znači da algoritam u početku jednako vjeruje u nagrade za sve radnje. Za razumijevanje ovog algoritma u potpunosti, potrebno je definirati Beta distribuciju (eng. *Beta distribution*). To je kontinuirana distribucija vjerojatnosti s dva pozitivna parametra, α i β. Distribucija je definirana na intervalu [0,1], a njezina funkcija gustoće vjerojatnost (eng. *probability density function*) dana je formulom:

$$f(x;\ \alpha,\beta) = \frac{x^{(\alpha-1)} * (1-x)^{(\beta-1)}}{B(\alpha,\beta)},$$

gdje je $B(\alpha, \beta)$ Beta funkcija, to jest normalizacijska funkcija koja osigurava da se funkcija gustoće vjerojatnosti integrira na 1 u intervalu [0,1].

Dio koda *np.random.beta(alphas[a],betas[a])* generira jedan nasumični uzorak iz Beta distribucije s navedenim α i β parametrima. Tako slijedi da prošireni dio koda [*np.random.beta(alphas[a], betas[a]) for a in bandit.action]* stvara popis koji sadrži nasumični uzorak iz Beta distribucije za svaku akciju koja se nalazi u klasi *Bandit*. Na kraju, potpuna linija koda *a = np.argmax([np.random.beta(alphas[a], betas[a]) for a in bandit.actions])* koristi Beta distribuciju za stvaranje uvjerenja agenta o raspodjeli nagrada za svaku „ruku" te odabire „ruku" s najvećom očekivanom nagradom kao radnju koju treba poduzeti. Bitno za naglasiti je to da agent ažurira svoja uvjerenja o raspodjeli nagrade za radnju koju je upravo napravio. To čini povećanjem vrijednosti *alphas[a]* za nagradu *r* i povećanjem vrijednosti *betas[a]* za *(1 – r)*. To utječe na njegov odabir radnji u budućim vremenskim koracima. Funkcija na kraju vraća popis akcija i nagrada.

4. PRIKAZ REZULTATA U JUPYTER NOTEBOOK-U

Jupyter Notebook je web aplikacija otvorenog koda koja se koristi za stvaranje i dijeljenje dokumenata koji sadrže jednadžbe, živi kod, vizualizacije i tekst. Aplikacija podržava mnoge programske jezike, uključujući Javu, C++ i Matlab. Jedna od glavnih prednosti Jupyter Notebook-a je njegova sposobnost da, uz vizualizaciju, podržava i interaktivnu analizu podataka [17].

Za početak je u Jupyter Notebook potrebno uvesti klasu *Bandit* te sve ostale funkcije iz Python-ovog programa, koji je nazvan *multiarmed.* Nakon toga se uvodi modul *pyplot* iz biblioteke *matplotlib*, koja je popularna Python biblioteka za vizualizaciju podataka. Modul *pyplot* pruža jednostavno sučelje za izradu dijagrama te se koristi u aplikacijama za strojno učenje i analizu podataka. Na slici [slika 19.] je prikazan kod u Jupyter Notebook-u.

```
In [1]: from multiarmed import Bandit
        from multiarmed import *
        import matplotlib.pyplot as plt

        pay_offs = [0.25, 0.3, 0.5, 0.1, 0.3, 0.25, 0]
        bandit = Bandit(pay_offs)

        methods = {'random_agent':random_agent, 'explore_greedy_agent':explore_greedy_agent,
                   'epsilon_greedy_agent':epsilon_greedy_agent, 'decaying_epsilon_greedy_agent':decaying_epsilon_greedy_agent,
                   'optimistic_greedy_agent':optimistic_greedy_agent, 'ucb_agent':ucb_agent,
                   'thompson_agent':thompson_agent, 'optimal_agent':optimal_agent}

        results = dict()

        number_of_iterations = 2000
        number_of_trials = 50

        for m in methods.keys():
            method = methods[m]
            all_rewards = []
            total_rewards = []
            cumulative_reward = []

            for trial in range(number_of_trials):
                total_reward = 0

                actions, rewards = method(bandit, number_of_iterations)
                all_rewards.append(rewards)
                total_reward += sum(rewards)
                cumulative_reward.append(total_reward)

            results[m] = all_rewards
            print('Agent:                ', m)
            print('Average cumulative reward: ', np.mean(cumulative_reward))
```

Slika 19. Kod u Jupyter Notebook-u

Zadaje se popis *pay_offs* koji sadrži prave isplate za svakog razbojnika te rječnik *methods* koji uključuje različite funkcije agenta koje treba testirati, gdje su ključevi nazivi za svaku metodu, a vrijednosti su odgovarajuće funkcije. U prazan rječnik *results* se upisuju dobiveni rezultati. Zadatak se izvršava za 50 rundi s 2000 ponavljanja. U konačnici kod izračunava prosječnu

kumulativnu nagradu svakog agenta. Rezultati su prikazani na slici [slika 20.].

```
Agent:                      random_agent
Average cumulative reward:  486.86
Agent:                      explore_greedy_agent
Average cumulative reward:  612.44
Agent:                      epsilon_greedy_agent
Average cumulative reward:  873.68
Agent:                      decaying_epsilon_greedy_agent
Average cumulative reward:  885.34
Agent:                      optimistic_greedy_agent
Average cumulative reward:  952.66
Agent:                      ucb_agent
Average cumulative reward:  975.46
Agent:                      thompson_agent
Average cumulative reward:  953.28
Agent:                      optimal_agent
Average cumulative reward:  1002.78
```

Slika 20. Prikaz prosječnih kumulativnih nagrada

Osim numeričkih rješenja, rezultati će se prikazati i pomoću grafova. Kod je prikazan na slici [slika 21.].

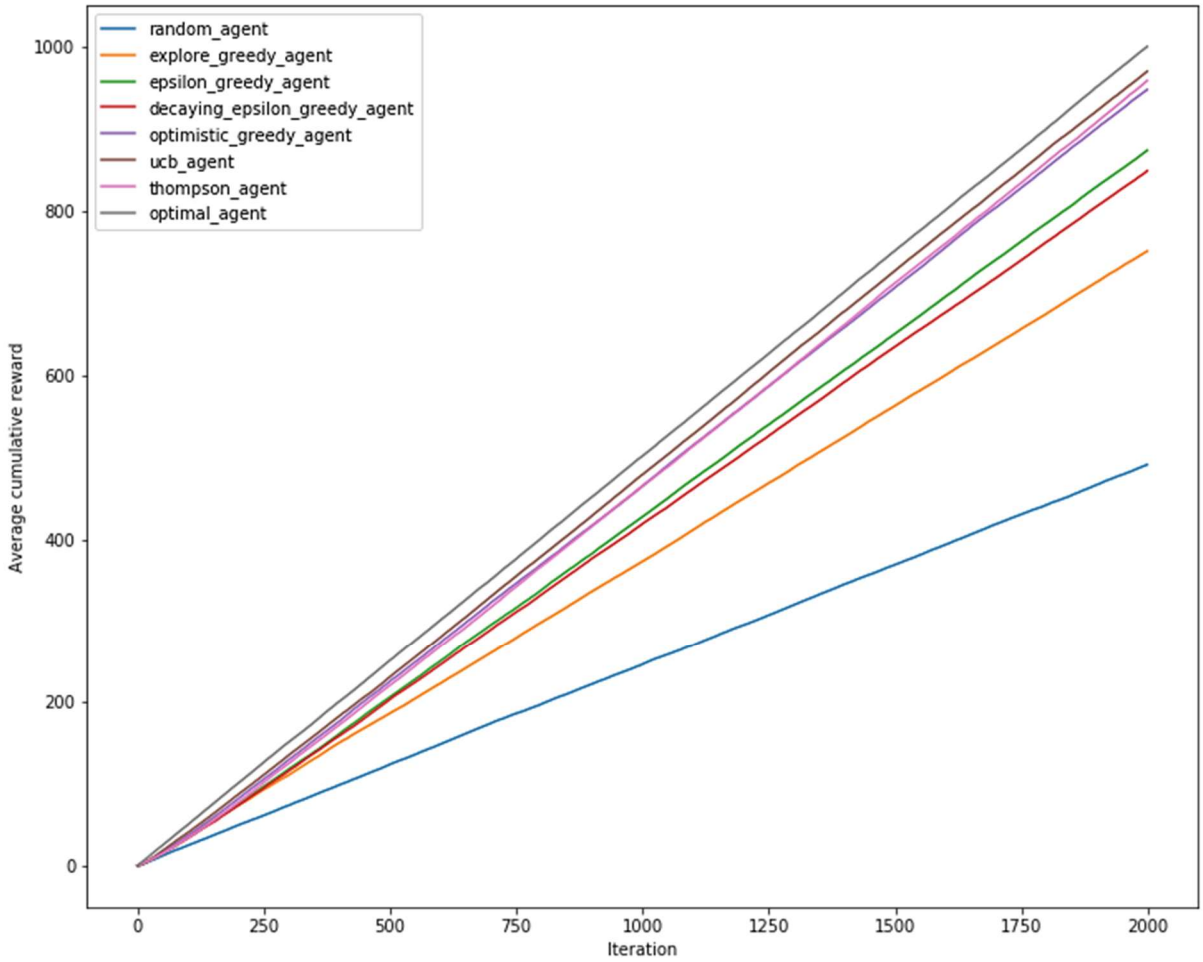
```
In [4]: # create a plot of the results
        plt.figure(figsize=(12,10))

        for agent in methods.keys():
            mean_reward_per_iteration = np.mean(results[agent],axis=0)
            mean_cum_reward_per_it = np.cumsum(mean_reward_per_iteration)
            plt.plot(mean_cum_reward_per_it, label=agent)

        plt.ylabel('Average cumulative reward')
        plt.xlabel('Iteration')
        plt.legend()
        plt.show()
```

Slika 21. Kod za prikaz grafova

Grafovi prikazuju kako se kumulativna nagrada svakog agenta povećava s vremenom te omogućuju usporedbu učinka obrađenih agenata. Prikazani su na idućoj slici.



Slika 22. Prikaz grafičkih rezultata

Iz grafa se može zaključiti da je nasumični agent najlošiji, što je i očekivano s obzirom da cijelo vrijeme nasumično odabire radnje. Najbolji agent je optimalni, ali njegovo rješenje nije realno jer agentu u stvarnosti nikad nije dana prava vrijednost nagrade. Svi ostali agenti nalaze se između prethodno spomenuta dva agenta. Iz slike se vidi da je agent s gornjom granicom pouzdanosti najbliži idealnoj vrijednosti. Optimistični epsilon agent s raspadom i Thompsonov agent također prikazuju dobre rezultate.

5. ZAKLJUČAK

U ovom radu je prikazan razvoj algoritama u sklopu rješavanja problema višerukog razbojnika. Svaki od algoritama ima svoj način kojim regulira odnos između iskorištavanja i istraživanja. Nakon primjene algoritama dobiva se rješenje u obliku grafičkog prikaza. Iz dobivenog grafa se zaključuje da je princip agenta s gornjom granicom pouzdanosti najbolji, iako ni optimistični epsilon agent s raspadom i Thompsonov agent ne zaostaju puno.

Ovim problemom se opisala problematika koja se pojavljuje u svakodnevnom životu te ima širok spektar primjene. Algoritmi višerukih razbojnika se upotrebaljavaju u financijama, oglašavanju, zdravstvu te sudjeluju u poboljšanju strojnog učenja. Postoje mnoge prednosti ovog problema, a neke od njih su kontinuirano učenje, učinkovito rješavanje problema i skalabilnost. Algoritmi su relativno jednostavni, razumljivi te se lako vizualiziraju, što može pomoći u razumijevanju i tumačenju.

Algoritam poput ovog je samo jedan od primjera koliko je tehnologija kroz godine napredovala. Robotsko oponašanje ljudskih radnji je aktualan problem u današnjem svijetu te će, razvojem umjetne inteligencije i strojnog učenja, postati naša budućnost.

LITERATURA

[1] https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained dostupno dana 11.2.2023.

[2] https://www.geeksforgeeks.org/ml-machine-learning/ dostupno dana 11.2.2023

[3] https://www.javatpoint.com/types-of-machine-learning dostupno dana 12.2.2023.

[4] https://www.mathworks.com/discovery/reinforcement-learning.html dostupno dana 12.2.2023.

[5] https://www.javatpoint.com/reinforcement-learning dostupno dana 12.2.2023.

[6] https://www.aionlinecourse.com/tutorial/machine-learning/upper-confidence-bound-%28ucb%29 dostupno dana 14.2.2023.

[7] https://www.javatpoint.com/exploitation-and-exploration-in-machine-learning dostupno dana 14.2.2023.

[8] https://pub.towardsai.net/openai-wants-to-improve-exploration-in-reinforcement-learning-models-by-using-noise-541ee4dba86a dostupno dana 14.2.2023.

[9] https://www.kaggle.com/code/marlesson/what-is-multi-armed-bandits dostupno dana 15.2.2023.

[10] Richard S. Sutton i Andrew G. Barto, Reinforcement learning: An introduction

[11] https://www.teradata.com/Glossary/What-is-Python dostupno dana 15.2.2023.

[12] https://www.coursera.org/articles/what-is-python-used-for-a-beginners-guide-to-using-python dostupno dana 15.2.2023.

[13] https://www.javatpoint.com/numpy-tutorial dostupno dana 16.2.2023.

[14] https://docs.python.org/3/library/random.html dostupno dana 16.2.2023.

[15] https://www.geeksforgeeks.org/greedy-algorithms-general-structure-and-applications/ dostupno dana 17.2.2023.

[16] https://www.w3schools.com/python/python_dictionaries.asp dostupno dana 17.2.2023.

[17] https://www.dominodatalab.com/data-science-dictionary/jupyter-notebook dostupno dana 17.2.2023.

[18] https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292 dostupno dana 18.2.2023.

PRILOG

1. Kod u Python-u

```
import random
import numpy as np


class Bandit:
    def __init__(self, payoff_probs):
        self.actions = range(len(payoff_probs))
        self.pay_offs = payoff_probs

    def sample(self, action):
        selector = random.random()
        return 1 if selector <= self.pay_offs[action] else 0


def explore_greedy_agent(bandit, iterations, initial_rounds=10, random_agent=False):

    pay_offs = dict()
    best_action = -1
    actions = []
    rewards = []

    if random_agent:
        initial_rounds = iterations

    for t in range(iterations):
        # for the initial rounds pick a random action
        if t < initial_rounds:
            a = random.choice(bandit.actions)
            r = bandit.sample(a)

            # update rewards
            if a in pay_offs:
                pay_offs[a].append(r)
            else:
                pay_offs[a] = [r]
        #  otherwise choose the best action so far
        else:
            if (best_action == -1):
                # check for the action with the best average payoff
                mean_dict = {}
                for key,val in pay_offs.items():
                    mean_dict[key] = np.mean(val)
                best_action = max(mean_dict, key=mean_dict.get)
            a = best_action

            r = bandit.sample(a)
```

```
        actions.append(a)
        rewards.append(r)

    return actions, rewards


def random_agent(bandit, iterations):

    return explore_greedy_agent(bandit, iterations, random_agent=True)


def epsilon_greedy_agent(bandit, iterations, epsilon=0.2, initial_rounds=10, decay=1, optimistic=False):

    pay_offs = dict()
    actions = []
    rewards = []

    if optimistic:
        pay_offs = {a:[1] for a in bandit.actions}

    for t in range(iterations):
        # sometimes randomly pick an action to explore
        if random.random() < epsilon or t < initial_rounds:
            a = random.choice(bandit.actions)
        # otherwise choose the best one thus far
        else:
            new_dict = {}
            for key,val in pay_offs.items():
                new_dict[key] = np.mean(val)
            a = max(new_dict, key=new_dict.get)

        r = bandit.sample(a)

        # update rewards
        if a in pay_offs:
            pay_offs[a].append(r)
        else:
            pay_offs[a] = [r]

        epsilon *= decay

        actions.append(a)
        rewards.append(r)

    return actions, rewards
```

```
def decaying_epsilon_greedy_agent(bandit, iterations, epsilon=0.2, initial_rounds=1, decay=0.99):

    return epsilon_greedy_agent(bandit, iterations, epsilon=epsilon, initial_rounds=initial_rounds, decay=decay)


def optimistic_greedy_agent(bandit, iterations, epsilon=0.2, initial_rounds=1, decay=0.99):

    return epsilon_greedy_agent(bandit, iterations, epsilon=epsilon, initial_rounds=initial_rounds, decay=decay, optimistic=True)


def optimal_agent(bandit, iterations):

    a = np.argmax(bandit.pay_offs)
    actions = []
    rewards = []

    for _ in range(iterations):
        r = bandit.sample(a)
        actions.append(a)
        rewards.append(r)

    return actions, rewards


def ucb_agent(bandit, iterations, c=0.3):

    pay_offs = dict()
    no_of_times_action_chosen = {a:0 for a in bandit.actions}
    q_values = {a:0 for a in bandit.actions}
    actions = []
    rewards = []


    for a in bandit.actions:
        for _ in range(2):
            r = bandit.sample(a)
            if a in pay_offs:
                pay_offs[a].append(r)
            else:
                pay_offs[a] = [r]
            actions.append(a)
            rewards.append(r)
            no_of_times_action_chosen[a] += 1
```

```
            q_values[a] = sum(pay_offs[a])/no_of_times_action_chosen[a]

    for t in range(2*len(bandit.actions), iterations):
        values = [q_values[a] + c*np.sqrt(np.log(t)/no_of_times_action_chosen[a]) for a in bandit.actions]
        a = np.argmax(values)
        r = bandit.sample(a)
        pay_offs[a].append(r)
        actions.append(a)
        rewards.append(r)

        # update q values and no_of_times_per_action
        no_of_times_action_chosen[a] += 1
        q_values[a] = sum(pay_offs[a])/no_of_times_action_chosen[a]

    return actions, rewards


def thompson_agent(bandit, iterations):

    pay_offs = dict()
    actions = []
    rewards = []

    alphas, betas = {a:1 for a in bandit.actions}, {a:1 for a in bandit.actions}

    for t in range(iterations):
        a = np.argmax([np.random.beta(alphas[a], betas[a]) for a in bandit.actions])
        r = bandit.sample(a)
        if a in pay_offs:
            pay_offs[a].append(r)
        else:
            pay_offs[a] = [r]

        actions.append(a)
        rewards.append(r)

        alphas[a] += r
        betas[a] += (1-r)

    return actions, rewards
```

2. Kod u Jupyter Notebook-u

In [1]:
```
from multiarmed import Bandit
from multiarmed import *
import matplotlib.pyplot as plt

pay_offs = [0.25, 0.3, 0.5, 0.1, 0.3, 0.25, 0]
bandit = Bandit(pay_offs)

methods = {'random_agent':random_agent, 'explore_greedy_agent':explore_greedy_agent,
           'epsilon_greedy_agent':epsilon_greedy_agent, 'decaying_epsilon_greedy_agent':decaying_epsilon_greedy_agent,
           'optimistic_greedy_agent':optimistic_greedy_agent, 'ucb_agent':ucb_agent,
           'thompson_agent':thompson_agent, 'optimal_agent':optimal_agent}

results = dict()

number_of_iterations = 2000
number_of_trials = 50

for m in methods.keys():
    method = methods[m]
    all_rewards = []
    total_rewards = []
    cumulative_reward = []

    for trial in range(number_of_trials):
        total_reward = 0

        actions, rewards = method(bandit, number_of_iterations)
        all_rewards.append(rewards)
        total_reward += sum(rewards)
        cumulative_reward.append(total_reward)

    results[m] = all_rewards
    print('Agent:                ', m)
    print('Average cumulative reward: ', np.mean(cumulative_reward))
```

In [4]:
```
# create a plot of the results
plt.figure(figsize=(12,10))

for agent in methods.keys():
    mean_reward_per_iteration = np.mean(results[agent],axis=0)
    mean_cum_reward_per_it = np.cumsum(mean_reward_per_iteration)
    plt.plot(mean_cum_reward_per_it, label=agent)

plt.ylabel('Average cumulative reward')
plt.xlabel('Iteration')
plt.legend()
plt.show()
```