Редакция Многорукие бандиты

Введение

- Многорукий бандит
- Проблематика задачи
- Как определить какая ручка лучше?

Практическая часть

- Жадная стратегия или е-жадная стратегия
- Upper confidence bound (UCB)
- Байесовский подход (сэмплирование Томпсона)
- Класс бандитов
- Тестовая среда
- Проверка работы

Примечание

- Отложенность обратной связи
- Новый контент

Библиотеки

Введение

Многорукий бандит - это алгоритм для корректировки выбора действия или ручки между конкурирующими (альтернативными) вариантами, чтобы максимизировать ожидаемую выгоду.

Простыми словами, задача **многоруких бандитов** заключается в том, чтобы найти самый выгодный вариант действия с минимальными потерями.

Представим, что были сделаны 2, 3 или 10 ручек по выбору чего-либо (в **практической части** будет рассмотрен пример с выбором 0 или 1).
Задача выбрать наиболее выгодную ручку (алгоритм), которая делает некий выбор с минимальными потерями. Можно воспользоваться А/Б тестами (мощный маркетинговый инструмент для повышения эффективности работы вашего интернет-ресурса), но как правило, результаты А/Б тестов нельзя принимать до завершения эксперимента. Они могут поменяться чуть более, чем полностью. С другой стороны **многорукие бандиты** подбирают лучшую ручку динамически и достаточно быстро, если оценка результата происходит сразу.

Проблематика задачи

1. Основная проблема: какую ручку выбрать в каждый момент времени.
2. Также ещё одной проблемой "бандитских" алгоритмов является то, что они online, т.е им необходимо оценивать результат сразу по завершении действия. При отложенной оценке появляются дополнительные проблемы.
3. Баланс Exploration/Exploration (исследование нового контента). При проявлении нового контента (условий/ручки) эти новые ручки абсолютно не имеют среднего вознаграждения и бандиты начинают их вставлять везде, где только можно. Иначе не исследовать новый вариант действия, что вступает в противоречие с пунктом 1.

Как определить какая ручка лучше?

1. Жадная стратегия или е-жадная стратегия
2. Upper confidence bound (UCB)
3. Байесовский подход (сэмплирование Томпсона)

Практическая часть

Общие элементы всех стратегий:

1. Определять упущенное вознаграждение за действие
2. Выбор ручки(дейсвия)
3. Каким-то способом сбрасывать стратегию

```
class Strategy:

    def __init__(self, n_arms: int):
        self.n_arms = n_arms                        # количество ручек для "дерганья"
        self.n_iters = 0                            # количество доступных итераций
        self.arms_states = np.zeros(n_arms)         # пространство состояний ручек
        self.arms_actions = np.zeros(n_arms)# пространство действий для ручек

    def flush(self):
        '''метод для обновления стратегии'''
        self.n_iters = 0
        self.arms_states = np.zeros(self.n_arms)
        self.arms_actions = np.zeros(self.n_arms)

    def update_reward(self, arm: int, reward: int):
        ''' функция для обновления стратегии, где учитываются действия и получаемые награды'''
        self.n_iters += 1
        self.arms_states[arm] += reward
        self.arms_actions[arm] += 1

    def choose_arm(self):
        '''непосредственно выбор "ручки бандита", с условием обязательной реализации в классе конкретной стратегии'''
        raise NotImplementedError
```

Жадная стратегия или e-жадная стратегия

Жадные стратегии основываются на одном простом принципе - всегда выбираем ручку, которая в среднем даёт наибольшую награду, самый лучший результат в нашем понимании. Но если так делать, можно застрять на одной ручке и игнорировать все остальные. При изменении среды это выйдет боком. Для этого и ес ть **e-жадная стратегия**. У нее есть единственный параметр — "e", определяющий вероятность, с которой мы выбираем не самую лучшую ручку, а случайную, таким образом исследуя нашу среду. Так что добавляем случайность.

В целом стратегии этого класса различаются в том, как мы исследуем среду, чтобы определить эту самую ручку.

Класс имеет единственный параметр: eps - вероятность выбора случайной для исследования среды. Со временем можно уменьшать эту вероятность (eps-decreasing).

```
class EpsGreedy(Strategy):

    def __init__(self, n_arms: int, eps: float = 0.1):
        super().__init__(n_arms)
        self.eps = eps

    def choose_arm(self):
        if random.random() < self.eps:
            return random.randint(0, self.n_arms - 1) # choose random arm
        else:
            return np.argmax(self.arms_states / self.arms_actions) # choose the best arm
```

Upper confidence bound (UCB)

Другие алгоритмы при принятии решения используют данные о средней награде. Проблема в том, что если действие даёт награду с какой-то вероятностью, то данные от наблюдений получаются шумные и мы можем неправильно определять самое выгодное действие. Алгоритм верхнего доверительного интервала (**up per confidence bound или UCB**) - семейство алгоритмов, которые пытаются решить эту проблему, используя при выборе данные не только о средней награде, но и о том, насколько можно доверять значениям награды.

Т.е. строится оптимистическое предположение о том, насколько хорош желаемый результат для каждого действия. На базе этого предположения строится верхний доверительный интервал, выбирается действие с наибольшей предполагаемой наградой в рамках интервала. Если наше предположение оказалось неверным, то наша оценка уменьшается и мы переключаемся на другое действие.

np.sqrt(2 * np.log(self.n_iters) / self.arms_actions) - эта штука призвана определить верхнюю границу доверительного интервала.

```
class UCB1(Strategy):

    def choose_arm(self):
        if self.n_iters < self.n_arms:
            return self.n_iters
        else:
            return np.argmax(self.ucb())


    def ucb(self):
        ucb = self.arms_states / self.arms_actions # mean x_j
        ucb += np.sqrt(2 * np.log(self.n_iters) / self.arms_actions) # confidence part
        return ucb
```

Байесовский подход (сэмплирование Томпсона)

В этой стратегии мы ставим каждой ручке в соответствие некоторое случайное распределение и на каждом шаге сэмплируем из этого распределения числа, выбирая ручку согласно максимуму. На основе обратной связи обновляем параметры распределения так, чтобы лучшим ручкам соответствовало распределение с большим средним, и его дисперсия уменьшалась с количеством действий.

Простыми словами, **сэмплирование** постепенно улучшает модель вероятности вознаграждения для каждого действия, а сами действия выбираются на основе образцов из распределения. Поэтому можно получить ожидание среднего значения вознаграждения, а также вычислить достоверность этого ожидания.

Алгоритм:

Есть подвыборка средних вознаграждений по каждой ручке для действия a: $R_1(a),...R_k(a)$

$R_j(a)Pr(R_j(a)r_1^a,r_2^a,...,r_n^a)$, где n — количество вознаграждений, которое мы получили при a $r_i^aPr(r_a\theta)$

1. Рассчитаем среднее по подвыборке: $R\hat{}(a)=sum(R_j(a))/k$, где k — количество сэмплов
2. Найдём такое a, где R^(a) максимально: $a=argmax_a(R\hat{}(a))$
3. Выполним a и получим r′
4. Обновим Pr(R(a)) на основе r′
5. Повторим для горизонта наблюдений

Чем больше n, тем уже полученное распределение.

Чем больше k, тем более точная оценка среднего R^(a).

```
class Thompson(Strategy):
    def __init__(self, n_arms: int):
        super().__init__(n_arms)
        self.alphas = np.ones(self.n_arms)
        self.betas = np.ones(self.n_arms)

    def choose_arm(self):
        arm = np.argmax([np.random.beta(self.alphas[i], self.betas[i]) for i in range(self.n_arms)])
        return arm

    def update_reward(self, arm: int, reward: int):
        super().update_reward(arm, reward)
        self.alphas[arm] += reward
        self.betas[arm] += 1 - reward
```

Класс бандитов

Основная задача класса - принять в себя среду и стратегию. В этом классе также содержится метод `action()`. Он выбирает ручку бандита согласно стратегии, и дальше, исходя из результата (награды), мы обновляем нашу стратегию

```
class Bandit:

    def __init__(self, env: BernoulliEnv, strategy: Strategy):
        self.env = env
        self.strategy = strategy

    def action(self):
        arm = self.strategy.choose_arm()
        reward = self.env.pull_arm(arm)
        self.strategy.update_reward(arm, reward)
```

Тестовая среда

В целом, в тестовой среде можно задать все, что угодно, границ нет. В данном случае распределение Бернулли 1 или 0 ((дискретное распределение вероятностей, моделирующее случайный эксперимент произвольной природы, при заранее известной вероятности успеха или неудачи)

```
class BernoulliEnv:

    def __init__(self, arms_proba: list):
        self.arms_proba = arms_proba

    @property
    def n_arms(self):
        return len(self.arms_proba)

    def pull_arm(self, arm_id: int):
        if random.random() < self.arms_proba[arm_id]:
            return 1
        else:
            return 0
```

Проверка работы

```
def calculate_regret(env: BernoulliEnv, strategy: Strategy, n_iters=2000):
    strategy.flush()
    bandit = Bandit(env, strategy)
    regrets = []
    for i in range(n_iters):
        reward = bandit.strategy.arms_actions.dot(env.arms_proba)
        optimal_reward = np.max(env.arms_proba) * i
        regret = optimal_reward - reward
        regrets.append(regret)
        bandit.action()

    return regrets
```

Задаем среду, создаем самих бандитов

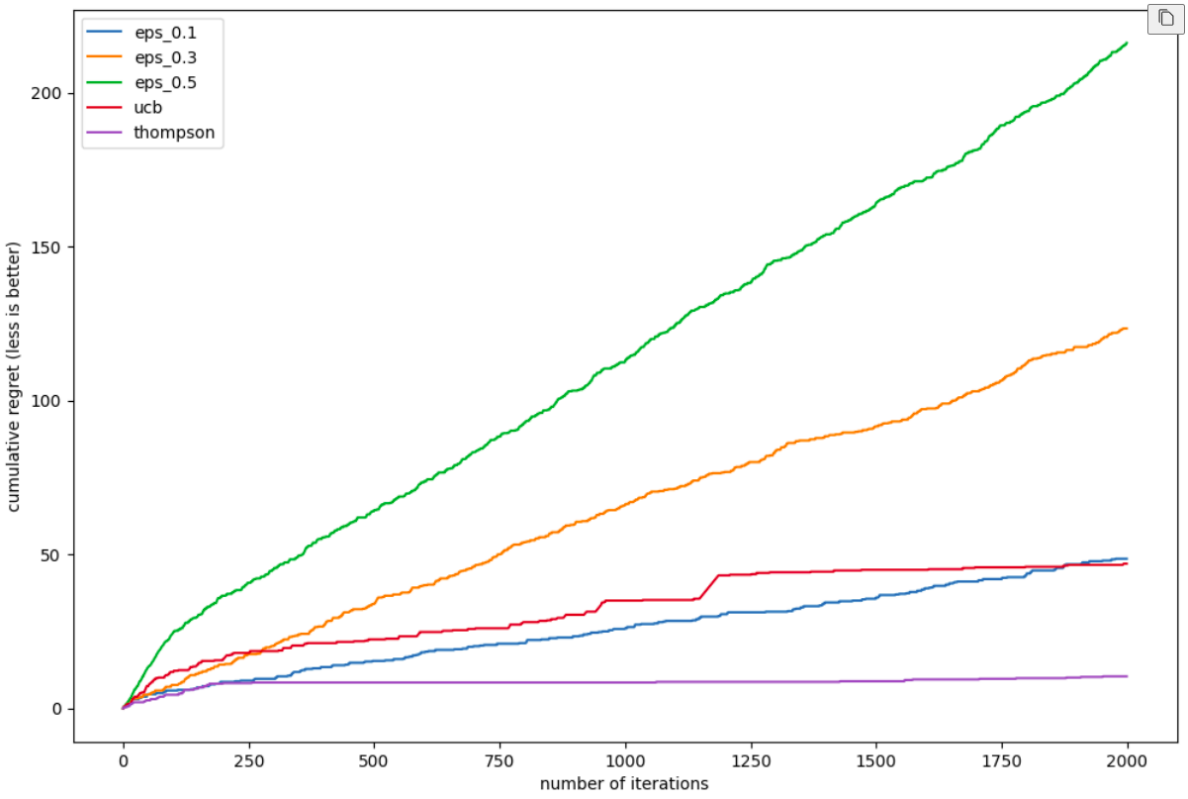
```
be = BernoulliEnv([0.3, 0.5, 0.7])
eps_1 = EpsGreedy(be.n_arms, 0.1)
eps_2 = EpsGreedy(be.n_arms, 0.3)
eps_3 = EpsGreedy(be.n_arms, 0.5)
ucb = UCB1(be.n_arms)
tompson = Thompson(be.n_arms)
```

Посчитаем упущенное вознаграждение, в форме "бандитов сожаления"

```
eps_regrets = calculate_regret(be, eps_1)
eps_2_regrets = calculate_regret(be, eps_2)
eps_3_regrets = calculate_regret(be, eps_3)
ucb_regrets = calculate_regret(be, ucb)
tompson_regrets = calculate_regret(be,tompson)
```

Результат

```
plt.figure(figsize=(12, 8))
plt.plot(eps_regrets, label='eps_0.1')
plt.plot(eps_2_regrets, label='eps_0.3')
plt.plot(eps_3_regrets, label='eps_0.5')
plt.plot(ucb_regrets, label='ucb')
plt.plot(tompson_regrets, label='thompson')
plt.legend()
plt.xlabel('number of iterations')
plt.ylabel('cumulative regret (less is better)')
```



Примечание

В нашем конкретном случае - сэмплирование Томпсона является фаворитом среди остальных стратегий, но надо помнить про две проблемы:

Отложенность обратной связи

Предполагается, что вы сразу же получаете от среды отклик на своё действие и можете тут же использовать обновленный опыт. Отклик среды и есть та обратная связь, которая позволяет быстро подстраиваться.
Отложенная обратная связь препятствует использованию UCB. В этом алгоритме награда (reward) вычисляется строго исходя из наблюдаемых значений активации. А поскольку отклик приходит с опозданием - на величину дельты этого опоздания параметры системы не меняются. Поэтому сильно страдает exploration (исследование нового контента).

Новый контент

Как правило, база постоянно пополняется новыми данными - редко, когда все статично. Для e-жадной стратегии (epsilon-greedy) совершенно безразлично сколько новых данных он получил. Понемногу он будет отдавать предпочтение новому и через какое-то время мы узнаем реальные вознаграждения. У стратегий UCB и сэмплировании Томпсона поведение отличается. Они постараются показывать сначала новое, а вознаграждение (reward) стабилизируется с течением некоторого времени. Если у нас честное онлайн обучение, то исследование нового не займет много времени.

К сожалению, совокупность **отложенной обратной связи** и **нового контента** ломает уже стратегию сэмплирования Томпсона.

В итоге самая неэффективная в идеальном мире стратегия дает наиболее стабильные результаты.

Библиотеки

```
import random

import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import pyplot as plt
```