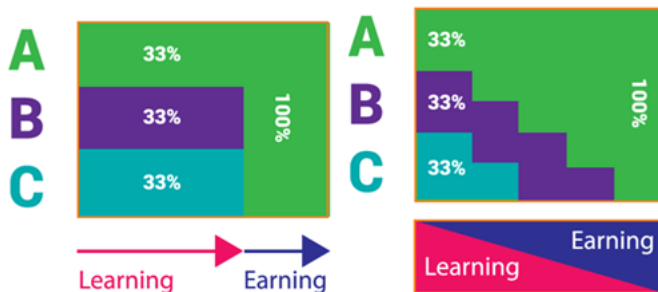


✓ A/B Testing from Scratch: Multi-armed Bandits

Frequentist and Bayesian A/B tests require you to divide your traffic into arbitrary groups for a period of time, then perform statistical tests based on the results. By definition, this forces us to divert out traffic to suboptimal variations during the test period, resulting in lower overall conversion rates. On the other hand, multi-armed bandit approach (MAB) dynamically adjusts the percentage of traffic shown to each variation according to how they have performed so far during the test, resulting in smaller loss in conversion rates.



Source: [Automizy](#) via [Multi-Arm Bandits: a potential alternative to A/B tests](#)

```
import numpy as np
import pandas as pd

#widgets
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from IPython.display import display

#plots
import matplotlib.pyplot as plt
from plotnine import *

#stats
import scipy as sp
import statsmodels as sm

import warnings
warnings.filterwarnings('ignore')
```

✓ Arms, Variations, Ads, or Anything

We treat serving a variation of content, be it product listings, recommended products, search results, online ads, or whatever we want to experiment on as *pulling an arm*. The arm will record an impression and, at an arbitrary amount of delay time, an action such as a click or add-to-cart based on that impression. In our example, we define our arm as a Bernoulli trial with the true probability of conversion ($\frac{\text{actions}}{\text{impressions}}$) as `true_p`.

```
class Arm:
    def __init__(self, true_p):
        self.true_p = true_p
        self.reset()

    def reset(self):
        self.impressions = 0
        self.actions = 0

    def get_state(self):
        return self.impressions, self.actions

    def get_rate(self):
        return self.actions / self.impressions if self.impressions > 0 else 0.

    def pull(self):
        self.impressions += 1
```

```

res = 1 if np.random.random() < self.true_p else 0
self.actions+=res
return res

```

```

a = Arm(0.1)
for i in range(100): a.pull()
a.get_state()

```

```
(100, 8)
```

Environment

We simulate an environment is an arbitrary number of arms with a set of predefined true probability (`true_p`) and average number of rewards (`avg_rewards`) per time period (`t`). This environment mimics most content serving APIs which display each variation at the ratio (`ps`) as defined by experimenters.

```

class MusketeerEnv:
    def __init__(self, true_ps, avg_impressions):
        self.true_ps = true_ps
        self.avg_impressions = avg_impressions
        self.nb_arms = len(true_ps)
        self.reset()

    def reset(self):
        self.t = -1
        self.ds=[]
        self.arms = [Arm(p) for p in self.true_ps]
        return self.get_state()

    def get_state(self):
        return [self.arms[i].get_state() for i in range(self.nb_arms)]

    def get_rates(self):
        return [self.arms[i].get_rate() for i in range(self.nb_arms)]

    # sample the actual number of impressions from a triangular function
    def get_impressions(self):
        return int(np.random.triangular(self.avg_impressions/2,
                                         self.avg_impressions,
                                         self.avg_impressions*1.5))

    # randomly choose arm based on a given probabiliy `ps`
    def step(self, ps):
        self.t += 1
        impressions = self.get_impressions()
        for i in np.random.choice(a=self.nb_arms,size=impressions,p=ps):
            self.arms[i].pull()
        self.record()
        return self.get_state()

    # for logging
    def record(self):
        d = {'t':self.t, 'max_rate':0, 'opt_impressions':0}

        for i in range(self.nb_arms):
            d[f'impressions_{i}'],d[f'actions_{i}'] = self.arms[i].get_state()
            d[f'rate_{i}'] = self.arms[i].get_rate()

            if d[f'rate_{i}'] > d['max_rate']:
                d['max_rate'] = d[f'rate_{i}']
                d['opt_impressions'] = d[f'impressions_{i}']

        d['total_impressions'] = sum([self.arms[i].impressions for i in range(self.nb_arms)])
        d['opt_impressions_rate'] = d['opt_impressions'] / d['total_impressions']
        d['total_actions'] = sum([self.arms[i].actions for i in range(self.nb_arms)])
        d['total_rate'] = d['total_actions'] / d['total_impressions']

```

```

d['regret_rate'] = d['max_rate'] - d['total_rate']
d['regret'] = d['regret_rate'] * d['total_impressions']
self.ds.append(d)

# for printing
def show_df(self):
    df = pd.DataFrame(self.ds)
    cols = ['t'] + [f'rate_{i}' for i in range(self.nb_arms)] + \
        [f'impressions_{i}' for i in range(self.nb_arms)] + \
        [f'actions_{i}' for i in range(self.nb_arms)] + \
        ['total_impressions', 'total_actions', 'total_rate'] + \
        ['opt_impressions', 'opt_impressions_rate'] + \
        ['regret_rate', 'regret']
    df = df[cols]
    return df

```

For instance, in a traditional A/B test with a default variation, new variation **A** and new variation **B**. We may divide 60% traffic to the default variation and 20% each to **A** and **B**. After 1,000 time steps, we will get the following results.

```

env = MusketeerEnv(true_ps = [0.1,0.12,0.13], avg_impressions=400)
for i in range(1000):
    env.step([0.6,0.2,0.2])
env.get_rates()

```

```
[0.09992498187411204, 0.11929434362836779, 0.1304435331706458]
```

In order to evaluate an MAB agent, we use 3 main metrics:

1. **opt_impressions_rate**: cumulative percentage of impressions we have given to the optimal arm at that timestep; this shows us how often we have picked the "best" arm
2. **regret_rate**: cumulative conversion rate of the best arm at that timestep minus cumulative conversion rate of all impressions; this shows us the difference in conversion rate we have lost by not picking the "best" arm
3. **regret**: cumulative actions if we had chosen the "best" arm minus actual cumulative conversions; this shows us how much actions we have lost by not picking the "best" arm

✓ Agent

An MAB agent solves the explore-vs-exploit dilemma. Exploitation means we choose what we know as the best choice at the current timestep, sometimes called being *greedy*; on the other hand, exploration means we try pulling other arms in order to know more about the environment.

Exploiting 100% of the time is a bad idea. For instance; let us assume there are two arms **A** and **B** with true probabilities 0.1 and 0.9 and it happens that when we pull **A** it returns a conversion whereas when we pull **B** it does not. If our policy is to always exploit, we would end up pulling only **A** which has much lower return rate than **B**. This is when you do not have any experiment set up for your content at all.

In contrast, if we always explore, we would end up pulling both arms randomly with expected return rates of $0.9 * 0.5 + 0.1 * 0.5 = 0.5$ instead of much higher if we could find out **B** is the better arm. This is close to what happens in a traditional A/B test during the test period.

Some common policies for distributing impressions to each arm are:

1. **Equal weights**: all arms have the same amount of traffic or a fixed amount.
2. **Randomize**: randomly assign traffic to all arms.
3. **Epsilon-greedy**: Assign a majority of traffic to the "best" arm at that time step, and the rest randomized among all arms; the degree of random traffic can be decayed by a parameter **gamma** as time goes on.
4. **Softmax or Boltzmann exploration**: Assigns traffic equal to the softmax activation of their current return rates; regulated by temperature parameter **tau** (lower **tau** means less exploration) that can also be decayed by **gamma** over time.

$$P(A_i) = \frac{e^{rate_i/\tau}}{\sum e^{rate_i/\tau}}$$

5. **Upper Confidence Bound**: by utilizing Hoeffding's Inequality, we can have a deterministic policy based on number of times the arms are pulled so far and impressions of each arm:

$$A = \operatorname{argmax}(rate_i + \sqrt{\frac{2 \log t}{impressions_i}})$$

6. **Deterministic Thompson Sampling:** based on a posterior distribution (in our case a Beta distribution) for each arm, sample that number of rates. Choose the arm with the highest sampled rate.
7. **Stochastic Thompson Sampling:** Instead of sampling only once, perform a Monte Carlo simulation for an arbitrary number of times, the traffic to each arm is divided by the percentage of times that arm is the best arm in the simulation.

```

from numpy._core.defchararray import equal
class BanditAgent:
    def __init__(self):
        pass

    #baselines
    def equal_weights(self,state):
        p_actions = np.array([1/len(state) for i in range(len(state))])
        return p_actions

    # TODO:1 : write a function that give the probability of choosing arm randomly
    def randomize(self,state):
        prob = np.array([np.random.rand() for i in range(len(state))])
        p_actions = prob / np.sum(prob)
        return p_actions

    # TODO:2 : write a function that give the probability of choosing arm based on epsilon greedy policy
    def eps_greedy(self, state, t, start_eps=0.3, end_eps=0.01, gamma=0.99):
        eps = start_eps*(gamma**t)
        if eps < end_eps:
            eps = end_eps
        p = np.random.rand()
        if p >= eps:
            p_actions = np.zeros(len(state))
            p_arm = np.zeros(len(state))
            for i in range(len(state)):
                ims,act = state[i]
                if ims > 0:
                    p_arm[i] = act/ims
                else:
                    p_arm[i] = 0
            arg_max = np.argmax(p_arm)
            best_arm = p_arm == p_arm[arg_max]
            p_actions[best_arm] = 1/sum(best_arm)
        else:
            p_actions = self.equal_weights(state)
        return p_actions

    # TODO:3 : write a function that give the probability of choosing arm based on softmax greedy policy
    def softmax(self, state, t, start_tau=1e-1, end_tau=1e-4, gamma=0.9):
        # do annealing on lecture
        tau = start_tau*(gamma**t)
        if tau < end_tau:
            tau = end_tau
        p_arm = np.zeros(len(state))
        for i in range(len(state)):
            ims,act = state[i]
            if ims > 0:
                p_arm[i] = act/ims
            else:
                p_arm[i] = 0
        m_val = np.max(p_arm)
        # Prevent overflow by remove prob with max value
        p_arm = p_arm - m_val
        p_actions = np.exp(p_arm/tau) / np.sum(np.exp(p_arm/tau))
        # becareful overflow on np.sum be very low value and make p_actions be infinity
        return p_actions

    # TODO:4 : write a function that give the probability of choosing arm based on UCB policy
    def ucb(self, state, t):
        p_actions = np.zeros(len(state))
        p_arm = np.zeros(len(state))

```

```

for i in range(len(state)):
    ims,act = state[i]
    if ims > 0:
        p = act/ims
        p_arm[i] = p + (np.sqrt(2*np.log(t)/ims) if ims > 0 else 0)
    else:
        p_arm[i] = 1e100 # guarantee choose
arg_max = np.argmax(p_arm)
best_arm = p_arm == p_arm[arg_max]
p_actions[best_arm] = 1/sum(best_arm)
return p_actions

```

Simulation Results

We simulate 4 campaigns with true probabilities of 12%, 13%, 15% and, 16% respectively. Our number of overall impressions is 400 on average.

```

env = MusketeerEnv(true_ps = [0.12,0.13,0.15,0.16], avg_impressions=400)
a = BanditAgent()
for i in range(20):
    p = a.equal_weights(env.get_state())
    env.step(p)
    t=i
a.equal_weights(env.get_state()), a.randomize(env.get_state()), a.eps_greedy(env.get_state(),t), \
a.softmax(env.get_state(),t), a.ucb(env.get_state(),t)

```

```

(array([0.25, 0.25, 0.25, 0.25]),
array([0.16258693, 0.35192041, 0.20424062, 0.28125204]),
array([0., 0., 1., 0.]),
array([0.02020389, 0.06288074, 0.4982668 , 0.41864857]),
array([0., 0., 1., 0.]))

```

```

N_policy = 5
envs = [MusketeerEnv(true_ps = [0.12,0.13,0.15,0.16], avg_impressions=400) for i in range(N_policy)]
a = BanditAgent()
for t in range(250):
    states = [env.get_state() for env in envs]
    actions = [a.equal_weights(states[0]), a.randomize(states[1]),
               a.eps_greedy(states[2],t), a.softmax(states[3],t),
               a.ucb(states[4],t)]

    for i in range(N_policy): envs[i].step(actions[i])

dfs = [env.show_df() for env in envs]
policies = ['equal_weights', 'randomize', 'eps_greedy', 'softmax', 'ucb']
for i in range(N_policy): dfs[i]['policy'] = policies[i]
df = pd.concat(dfs)[['policy', 't', 'opt_impressions_rate', 'regret_rate', 'regret', 'total_rate']]

```

```

df_m = df.melt(id_vars=['policy', 't'])
df_m

```

	policy	t	variable	value	
0	equal_weights	0	opt_impressions_rate	0.224138	
1	equal_weights	1	opt_impressions_rate	0.241843	
2	equal_weights	2	opt_impressions_rate	0.258278	
3	equal_weights	3	opt_impressions_rate	0.253567	
4	equal_weights	4	opt_impressions_rate	0.254645	
...	
4995	ucb	245	total_rate	0.153976	
4996	ucb	246	total_rate	0.154035	
4997	ucb	247	total_rate	0.154171	
4998	ucb	248	total_rate	0.154266	
4999	ucb	249	total_rate	0.154294	

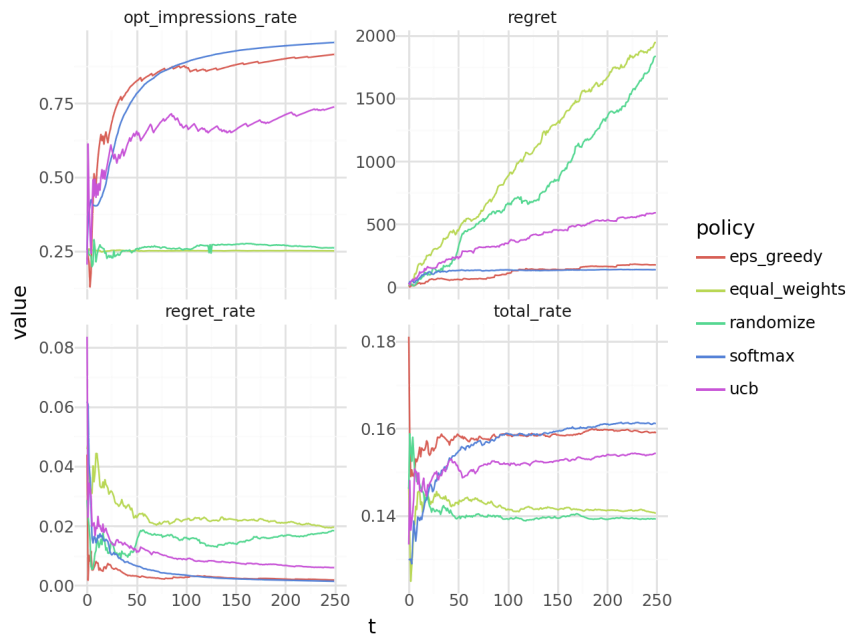
5000 rows × 4 columns

ขั้นตอนถัดไป:

สร้างโค้ดด้วย df_m

New interactive sheet

```
g = (ggplot(df_m, aes(x='t', y='value', color='policy', group='policy')) +
     geom_line() + theme_minimal() + facet_wrap('~variable', scales='free_y'))
g
```



TODO:5 Compare the result. Which policy has the best performance ?

Submit a pdf with screenshots of TODO1-5.

TODO:5 ตอน จากกราฟข้างต้นสรุปได้ว่า performance ของ softmax ดีที่สุดเนื่องจาก total rate มากสุดเมื่อเวลาผ่านไป 250 ครั้ง แต่โดยปกติ Ucb มักจะมากที่สุด แต่เหตุการณ์นี้อาจเกิดได้จากหลายปัจจัยเช่น 250 ครั้งอาจจะไม่พอทำให้ค่ามันยังไม่แม่นยำ ถ้าเพิ่มเป็น 1000 จะแม่นยำมากขึ้น

References

Here are some useful resources reviewed for this notebook.

- [tldr Bayesian A/B test](#)
- [Bayesian A/B Testing: a step-by-step guide](#)
- [Bayesian Coin Flips](#)
- [Multi-Arm Bandits: a potential alternative to A/B tests](#)
- [Multi Armed Bandits and Exploration Strategies](#)
- [MAB Google](#)

