Homework Stats 3

Week 5

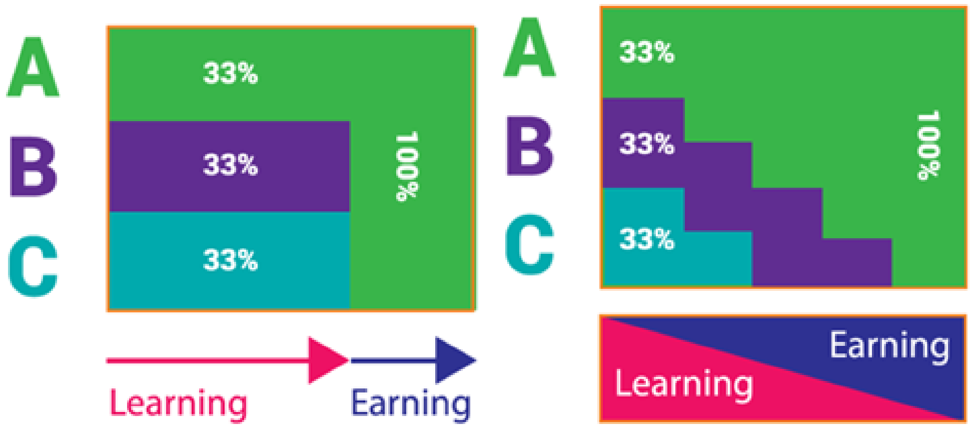
6733172621 Patthadon Phengpinij

Collaborators. ChatGPT (for LaTeX styling and grammar checking)

1 MAB: Multi-Armed Bandit

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-barmed bandit appraoch (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.

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`.

```python
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
```

The `Arm` class simulates an arm of a multi-armed bandit. It has the following methods:

- `__init__(self, true_p)`:

  Initializes the arm with a true probability of conversion `true_p` and resets the arm's state.
- `reset(self)`:

  Resets the arm's state by setting the number of impressions and actions to zero.
- `get_state(self)`:

  Returns the current state of the arm as a tuple containing the number of impressions and actions.
- `get_rate(self)`:

  Calculates and returns the conversion rate of the arm, which is the ratio of actions to impressions. If there are no impressions, it returns 0.
- `pull(self)`:

  Simulates pulling the arm by incrementing the number of impressions and determining whether an action occurs based on the true probability. It updates the number of actions accordingly and returns the result of the pull (1 for action, 0 for no action).

For example,

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

a.get_state()
```

After pulling the arm 100 times, we can check its state using `get_state()`, which will return the number of impressions and actions recorded by the arm.

The output will be similar to `(100, 13)`, indicating that there were 100 impressions and 13 actions (this is result at `random_seed`) = 42.

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.

```python
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 * 0.5,
                self.avg_impressions,
                self.avg_impressions * 1.5
            )
        )

    # ramdomly 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": 0.0,
            "opt_impressions": 0.0
        }

        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
```

The `MusketeerEnv` class simulates a multi-armed bandit environment with multiple arms, each having its own true probability of conversion. It provides methods to reset the environment, get the current state of the arms, simulate pulling the arms based on a given probability distribution, and record the results of each time step. The environment also keeps track of various metrics such as total impressions, total actions, conversion rates, and regret.

The main methods of the `MusketeerEnv` class are:

- `__init__(self, true_ps, avg_impressions)`:

  Initializes the environment with a list of true probabilities for each arm and the average number of impressions per time step. It also resets the environment.

- `reset(self)`:

  Resets the environment by setting the time step to -1, clearing the recorded data, and creating new arms based on the provided true probabilities.

- `get_state(self)`:

  Returns the current state of all arms in the environment as a list of tuples, where each tuple contains the number of impressions and actions for each arm.

- `get_rates(self)`:

  Returns the conversion rates of all arms in the environment as a list.

- `get_impressions(self)`:

  Samples the actual number of impressions for the current time step from a triangular distribution based on the average number of impressions.

- `step(self, ps)`:

  Simulates a time step in the environment by incrementing the time step counter, sampling the number of impressions, and randomly choosing arms to pull based on the provided probability distribution `ps`. It then records the results and returns the current state of the arms.

- `record(self)`:

  Records the results of the current time step, including impressions, actions, conversion rates, and various metrics such as optimal impressions rate and regret.

- `show_df(self)`:

  Returns a pandas DataFrame containing the recorded data for all time steps, organized in a specific order of columns.

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.

Using the following code:

```python
N = 1000
env = MusketeerEnv(true_ps = [0.1, 0.12, 0.13],
    avg_impressions=400)

for i in range(N):
    env.step([0.6, 0.2, 0.2])

print(env.get_rates())
```

We will get the output similar to:

```
[0.09988851447635333, 0.11837382400725431, 0.12914850187737822]
```

Indicating that the estimated conversion rates for the (1) default variation, (2) variation `A`, and (3) variation `B` are approximately 9.99%, 11.84%, and 12.91% respectively.

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 $0.9 \times 0.5 + 0.1 \times 0.5 = 0.5$ 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.

$$P(A_i) = \frac{e^{\text{rate}_i/\tau}}{\sum e^{\text{rate}_i/\tau}}$$

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:

$$A = \arg\max \left(\text{rate}_i + \sqrt{\frac{2\log t}{\text{impressions}_i}} \right)$$

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.

TO SUBMIT

Problem 1. From the following `BanditAgent` class:

```python
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
```

Implement the following policies:

1.1 **TODO1:** write a `randomize` function that give the probability of choosing arm randomly

Solution. Because each arm has an equal chance of being selected, the probability of choosing each arm is uniform across all arms.

```python
def randomize(self, state):
    p_actions = np.random.rand(len(state))
    p_actions = p_actions / p_actions.sum()
    return p_actions
```

Explanation: The `randomize` function generates a random probability distribution for selecting arms. Using `np.random.rand(len(state))`, it creates an array of random values for each arm. To ensure that these values represent valid probabilities, the function normalizes the array by dividing each element by the sum of all elements, resulting in a probability distribution where the sum of probabilities equals 1.

1.2 **TODO2:** write a `eps_greedy` function that give the probability of choosing arm based on epsilon greedy policy

Solution. Because the epsilon-greedy policy balances exploration and exploitation, the function calculates the probability of choosing each arm based on the current estimated rates and a decaying exploration factor.

```python
def eps_greedy(self, state, t, start_eps=0.3, end_eps
    =0.01, gamma=0.99):
    eps = end_eps + (start_eps - end_eps) * np.exp(-gamma
     * t)

    n_arms = len(state)
    rates = np.array([ (s[1] / s[0] if s[0] > 0 else 0)
    for s in state ])
    best_arm = np.argmax(rates)
    p_actions = np.ones(n_arms) * (eps / n_arms)
    p_actions[best_arm] += (1 - eps)
    return p_actions
```

Explanation: The eps_greedy function implements the epsilon-greedy policy for selecting arms in a multi-armed bandit scenario. It first calculates the exploration probability eps, which decays exponentially over time based on the parameters start_eps, end_eps, and gamma.

$$\text{eps}(t) = \text{end\_eps} + (\text{start\_eps} - \text{end\_eps}) \times e^{-\text{gamma} \times t}$$

Next, it computes the estimated conversion rates for each arm using the current state, identifying the arm with the highest rate as the best_arm. The function then constructs a probability distribution p_actions where each arm is assigned a base probability of eps / n_arms for exploration. The best_arm receives an additional probability of 1 - eps for exploitation. Finally, the function returns the probability distribution for selecting each arm.

1.3 **TODO3:** write a softmax function that give the probability of choosing arm based on softmax greedy policy

Solution. Because the softmax policy assigns probabilities based on the estimated rates of each arm, the function calculates the softmax probabilities using a temperature parameter that decays over time.

```python
def softmax(self, state, t, start_tau=1e-1, end_tau=1e-4, gamma=0.9):
    tau = end_tau + (start_tau - end_tau) * np.exp(-gamma * t)

    rates = np.array([ (s[1] / s[0] if s[0] > 0 else 0) for s in state ])
    max_scaled = np.max(rates / tau)
    exp_rewards = np.exp((rates / tau) - max_scaled)

    p_actions = exp_rewards / exp_rewards.sum()
    return p_actions
```

Explanation: The softmax function implements the softmax policy for selecting arms in a multi-armed bandit scenario. It first calculates the temperature parameter tau, which decays exponentially over time based on the parameters start_tau, end_tau, and gamma.

$$\text{tau}(t) = \text{end\_tau} + (\text{start\_tau} - \text{end\_tau}) \times e^{-\text{gamma} \times t}$$

Next, it computes the estimated conversion rates for each arm using the current state. To ensure numerical stability during the exponentiation step, the function subtracts the maximum scaled rate from each rate divided by tau. It then calculates the exponentiated rewards using the softmax formula.

$$\text{exp\_rewards}_i = e^{(\text{rate}_i/\text{tau}) - \max(\text{rates}/\text{tau})}$$

Finally, the function normalizes the exponentiated rewards to obtain a probability distribution p_actions for selecting each arm and returns it.

1.4 **TODO4:** write a `ucb` function that give the probability of choosing arm based on UCB policy

Solution.

```python
def ucb(self, state, t):
    rates = np.array([ (s[1] / s[0] if s[0] > 0 else 0)
    for s in state ])
    impressions = np.array([ s[0] for s in state ])

    ucbs = rates + np.array([ np.sqrt(2 * np.log(t + 1) /
     imp) if imp > 0 else float("inf") for imp in
    impressions ])
    best_arm = np.argmax(ucbs)

    p_actions = np.zeros(len(state))
    p_actions[best_arm] = 1.0
    return p_actions
```

Explanation: The `ucb` function implements the Upper Confidence Bound (UCB) policy for selecting arms in a multi-armed bandit scenario. It first computes the estimated conversion rates for each arm using the current state. Next, it retrieves the number of impressions for each arm. The function then calculates the UCB values for each arm using the formula:

$$\text{UCB}_i = \text{rate}_i + \sqrt{\frac{2\log(t+1)}{\text{impressions}_i}}$$

If an arm has not been pulled yet (i.e., impressions are zero), its UCB value is set to infinity to ensure it gets selected. The function identifies the arm with the highest UCB value as the `best_arm`. Finally, it constructs a probability distribution `p_actions` where only the `best_arm` has a probability of 1.0, while all other arms have a probability of 0.0, and returns this distribution.

After implementing all the required functions, the complete `BanditAgent` class is as follows:

```python
class BanditAgent:
    def __init__(self): pass

    def equal_weights(self, state): ...
        return p_actions

    def randomize(self, state): ...
        return p_actions

    def eps_greedy(self, state, ...): ...
        return p_actions

    def softmax(self, state, ...): ...
        return p_actions

    def ucb(self, state, ...): ...
        return p_actions
```

We can now use this `BanditAgent` class to simulate and compare the performance of different policies in a multi-armed bandit environment.

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.

```python
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

results_df = pd.DataFrame(
    (
        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)
    ),
    index=[
        "equal_weights",
        "randomize",
        "eps_greedy",
        "softmax",
        "ucb"
    ]
)
```

The results of each policy after 20 time steps are as follows:

Policy	Arm 0	Arm 1	Arm 2	Arm 3
equal_weights	0.2500	0.2500	0.2500	0.2500
randomize	0.5476	0.2092	0.0779	0.1652
eps_greedy	0.0025	0.0025	0.0025	0.9925
softmax	1.48×10^{-149}	1.96×10^{-137}	1.70×10^{-5}	0.999983
ucb	0.0000	0.0000	0.0000	1.0000

After 20 time steps, we can see that both `eps_greedy`, `softmax`, and `ucb` policies have converged to choosing Arm 3 almost all the time, which has the highest true probability of 16%. While the `randomize` policy shows a more varied distribution of choices among the arms, while the `equal_weights` policy maintains an equal distribution across all arms.

To further analyze the performance of each policy over time, we can visualize the metrics such as opt_impressions_rate, regret_rate, regret and total_rate after running for 250 times. The following code generates the plots:

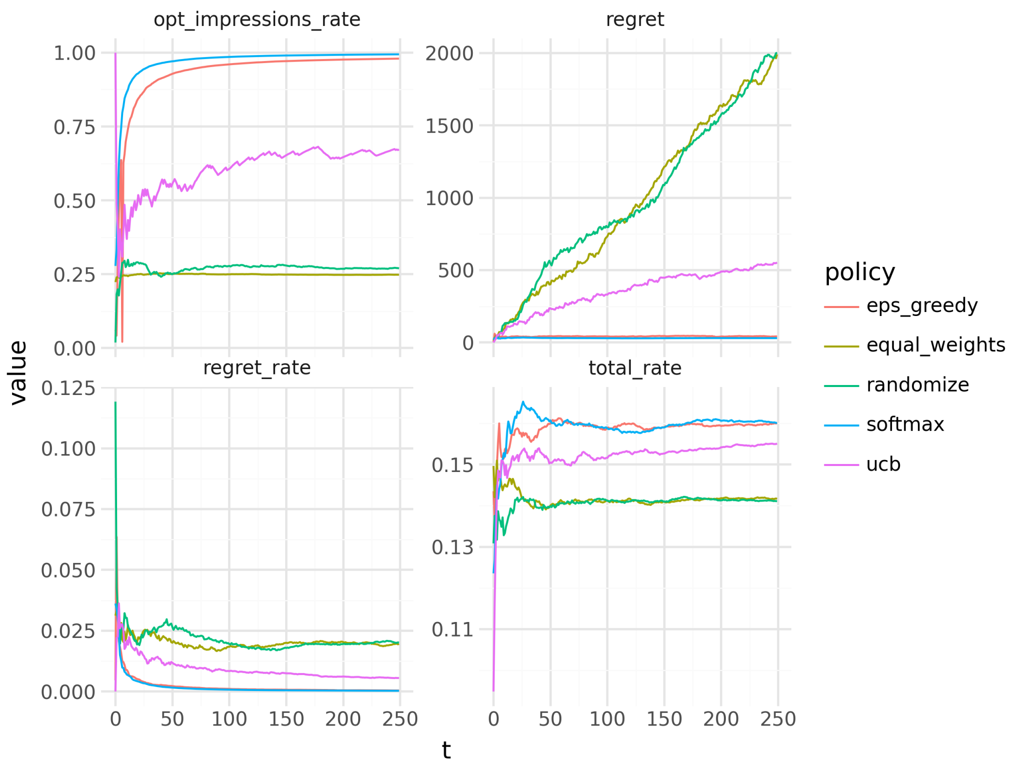
```python
N_policy = 5

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"]]

g = (
    ggplot(df_m, aes(x="t", y="value", color="policy", group="
    policy")) +
    geom_line() +
    theme_minimal() +
    facet_wrap("~variable", scales="free_y")
)

display(g)
```

The code initializes multiple instances of the MusketeerEnv class, each representing a different policy. It then runs a simulation for 250 time steps, where at each step, it retrieves the current state of each environment and determines the actions to take based on the respective policies.

The following table is the head and tail of the combined DataFrame used for plotting:

policy	t	opt_impressions_rate	regret_rate	regret	total_rate
equal_weights	0	0.223278	0.031207	13.138298	0.149644
equal_weights	1	0.238987	0.033045	30.004608	0.142070
equal_weights	2	0.240838	0.034275	45.826087	0.145849
equal_weights	3	0.239195	0.024766	41.829208	0.150977
...	...	...	...	...	...
ucb	246	0.670520	0.005602	549.024163	0.154991
ucb	247	0.672082	0.005560	547.561962	0.154987
ucb	248	0.670326	0.005550	547.983229	0.154998
ucb	249	0.672070	0.005568	552.722370	0.155125

The resulting plots illustrate how each policy performs over time in terms of the selected metrics.



TO SUBMIT

Problem 2. TODO5: Compare the result. Which policy has the best performance?
Solution.

First, lets print out the final value of `"opt_impressions_rate"`, `"regret_rate"`, `"regret"`, and `"total_rate"`

```
df[df["t"]==df["t"].max()]
```

To select the best performance algorithm, we should choose the algorithm that has the highest `"total_rate"` and the lowest `"regret"` among all algorithms.

The following is the result of the simulation.

Policy	`opt_impressions_rate`	`regret_rate`	`regret`	`total_rate`
`equal_weights`	0.250529	0.021550	2150.471960	0.142090
`randomize`	0.255510	0.021394	2131.409491	0.140533
`eps_greedy`	0.977498	0.000487	48.635314	0.159635
`softmax`	0.987997	0.000152	15.510418	0.161609
`ucb`	0.667973	0.005684	561.241847	0.153273

Since, the `softmax` algorithm has the highest `"total_rate"` of `0.161609` and the lowest `"regret"` of `15.510418`, we can conclude that the `softmax` algorithm has the best performance among all algorithms.

References

Here are some useful resources reviewed for this exercise:

- tl;dr 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