11. A/B 테스팅과 '여러 팔 강도' 문제

11.1. A/B 테스팅

최근 인터넷을 이용한 상거래가 활발해지면서 'A/B 테스팅'이라는 아이디어가 인기를 얻게 되었다. A/B 테스팅은 메뉴, 문구, 광고 등을 고객마다 A안과 B안으로 다르게 보여주고 가장 반응이 좋은 안을 선택하는 방법이다. 꼭 2가지 안만이 아니라 다양한 안을 보여주는 경우도 포함한다.

A/B 테스팅은 과학계에서 사용하는 무작위 대조군 시험(randomized controlled trial: RCT)과 동일한 것이다. 예를 들어 의학에서 약의 효과를 검증할 때는 환자들을 무작위로 나누고 일부 환자들에게는 진짜 약을 주고, 다른 환자들에게는 가짜 약을 준다.

기초 통계학에서 배우는 t-검정, 카이제곱 검정 등은 이런 문제에서 집단간 비교를 위한 기법들이다.

그런데 이러한 A/B 테스팅에는 몇 가지 문제가 있다.

첫째, 탐색(exploration)의 문제다. 다양한 대안을 테스트하는 과정은 그 자체로 비용을 발생시킨다.

어느 회사의 쇼핑몰 사이트에서 주문 버튼에 "지금 주문하세요"라는 문구(A안)와 "구입하기"라는 문구(B안)를 무작위로 50:50으로 내보내고 하루 동안 비교해보았다고 하자. 하루 동안의 A/B테스트를 해보니 A안을 보여준 고객들이 구매한 액수는 100억이었고, B안을 보여준 고객들이 구매한 액수는 50억이었다.

그런데 만약 이 회사 홈페이지에는 A안으로 되어있었다면, 이 A/B 테스트를 하기 위해 사실상 50억의 비용을 치른 셈이다. 문구 하나를 고르는 것치고 지나치게 많은 비용이 들었다.

물론 B안으로 되어있었다면 50억을 추가로 벌었다고 생각할 수도 있다. 그런데 테스트를 하루동안이나 하지 않고, 좀 더 짧게 한 나절 정도만 해본 후 빠르게 A안으로 통일했다면 몇 십억을 더 벌 수 있지 않았을까? 이 경우에도 역시 기회비용이 발생한다.

둘째, 활용(exploitation)의 문제다. 반대로 테스트를 하지 않아도 많은 기회를 놓치게 된다.

앞서의 회사에서 A/B 테스트를 짧게 하고 그 결과를 활용하면 더 많은 수익을 얻을 수 있다. 단순히 산술적으로만 생각하면 테스트를 짧게 하면 짧게 할 수록 그 결과를 더 많이 활용할 수있으므로 더 많은 수익이 가능하다.

그런데 통계라는 것은 표본이 작아지면 작아질 수록 그 결과가 신뢰성이 떨어진다. 테스트가 짧아지면 A안과 B안 중에 잘못된 안을 고를 수도 있다.

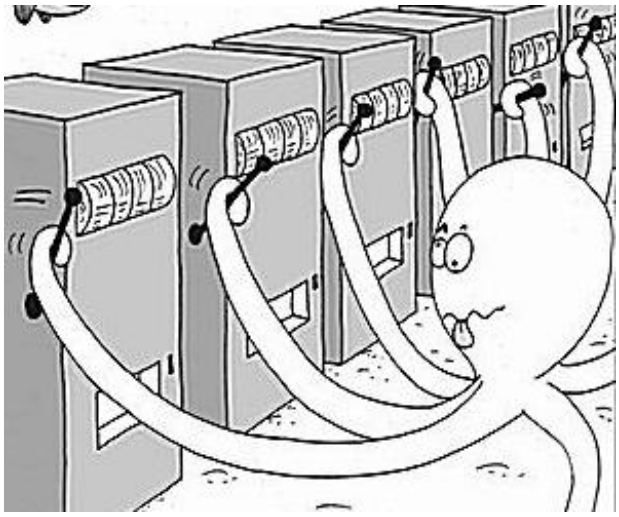
게다가 주중에는 A안이 더 효과적이지만 주말에는 B안이 더 효과적이라든가 하면 테스트를 하루만 하는 것으로는 부족하고 1주일은 해야될 수도 있다. 여름과 겨울에 결과가 다르다든가 아니면 당장은 A안이 효과적이지만 몇 년에 걸쳐 서서히 B안이 더 우세해진다면 어떻게 할 것인가?

이렇게 테스트에는 **탐색-활용 교환**(exploration-exploitation tradeoff)가 존재한다. 테스트를 많이 하면 많이 하는데로, 안하면 안하는데로 문제인 것이다. 이러한 문제를 체계화한 것이 "여러 팔 강도" 문제다.

11.2. "여러 팔 강도" 문제

영어에서 슬롯 머신을 속어로 "한 팔(one-armed) 강도(bandit)"라고 한다. 모양이 팔이 하나 달린 것처럼 생겼고, 결국 고객의 돈을 털어가기 때문이다. 만약 카지노에 여러 대의 슬롯 머신이 있다고 해보자. 카지노에서는 슬롯머신마다 돈을 따고 잃을 확률을 다르게 설정해둔다. 몇 명은 돈을 따게

해줘야 카지노에 계속 '호구'들이 오기 때문이다. 고객들은 돈을 따게 해줄 슬롯 머신을 어떻게 해야 찾을 수 있을까? 이 문제를 슬롯 머신이 여러 대 있으므로 "여러 팔(multi-armed) 강도(bandit)", 줄여서 MAB 문제라고 한다.



MAB문제는 쉽게 생각하면 모든 슬롯머신을 여러 번 당겨보고 수익률을 비교해보면 될 것이다. 그런데 슬롯머신은 당겨보는데도 돈이 들어간다. 슬롯 머신으로 돈을 벌려면 슬롯 머신의 확률을 정확히 알아야 하는데, 슬롯 머신의 확률을 정확히 알자면 슬롯 머신을 많이 당겨봐야 하니 돈이 많이 들어간다. 앞서 말한 탐색-활용 교환이 여기에도 똑같이 있다는 것을 알 수 있다.

이제 MAB 문제의 몇 가지 용어를 알아보자.

- **행동**(action): MAB에서 할 수 있는 하나의 선택. (예: A/B 테스팅에서 A안, B안)
- **보상**(reward): 한 번의 행동에 따른 수치화된 결과 (예: 한 고객에게 안을 보여줬을 때 구매액)
- **가치**(value): 행동의 기대 보상, 또는 행동에 따르는 보상의 평균.

MAB에서는 모든 행동이 순서대로 발생한다고 가정할 것이다. 그 순서에 따라 시점 t의 행동을 A_t라 하고, 그 행동의 보상은 R_t로 표기한다.

행동 a의 가치는 $q_*(a)$, 시점 t에 추정된 가치는 $Q_t(a)$라고 쓴다.

11.3. 탐욕 알고리즘

MAB를 푸는 가장 간단한 방법은 탐욕 알고리즘이다. 우선 현재 시점 t까지 행동 a를 사용한 경우 중에서 보상의 표본 평균으로 행동의 가치 $Q_t(a)$를 추정한다. 만약 지금까지 그 행동을 한 번도 선택하지 않았다면 행동의 가치는 미리 설정한 기본값으로 추정한다. 그 다음 $Q_t(a)$가 가장 높은 행동 a를 선택하는 것이다. 수식으로 써보면 아래와 같다.

$$A_t = \arg\max_a Q_t(a)$$

R로 모의실험을 해보자.

```
actions = c('A', 'B')  # 행동들
rewards = c(A = 0, B = 0)  # 지금까지 보상 합계
counts = c(A = 0, B = 0)  # 행동을 한 횟수
values = c(A = 0, B = 0)  # 추정 가치

# 모의실험을 위한 행동의 보상 분포 (현실에서는 알 수 없음)
mean = c(A = 1, B = 2)  # 보상의 모평균(실제 가치)
sd = c(A = 0.5, B = 1)  # 보상의 모표준편차

# 100회 모의 실험
```

```
for(i in 1:100){
      # 가치가 가장 큰 행동을 고른다
      best = names(which.max(values))
    print(best)

    # 보상을 받는다
      R = rnorm(1, mean[best], sd[best])

    # 지금까지 합계에 더한다
      rewards[best] = rewards[best] + R
    # 행동을 한 횟수에 1을 더한다
      counts[best] = counts[best] + 1
    # 가치를 계산한다
      values[best] = rewards[best] / counts[best]
}
counts
values
```

위의 모의 실험에서 A안의 실제 가치는 1이고 B안의 실제 가치는 2이다. 따라서 B안을 선택하게 되는 결과가 나와야 겠지만 A안을 대부분 하게 된다. 시작할 때는 둘 다 기본값이므로 A안이 순서상 먼저 선택된다. 그리고 A안은 가치가 작지만 분산이 작아서 보통 +의 보상을 받게 된다. B안은 추정가치가 0이므로 계속 A안이 선택된다. A안의 데이터가 자꾸 쌓이면 A안의 추정 가치는 실제 가치인 1로 수렴하게 된다. 결국에는 B안은 영원히 선택하지 않고 A안만 계속 선택하게 된다.

여기서 탐욕 알고리즘의 한 가지 문제를 알 수 있다. 탐색보다 활용을 너무 많이 한다는 것이다.

11.3.1. 입실론 탐욕 알고리즘

탐욕 알고리즘에서 탐색을 촉진하는 한 가지 변형은 입실론-탐욕(ϵ-greedy) 알고리즘이다. 그리스 문자 입실론(ϵ)은 작은 수치를 의미하는 표현으로 자주 쓰인다. 입실론 탐욕 알고리즘은 대부분의 경우는 현재 최선의 행동을 하되, 가끔은 무작위로 행동하는 알고리즘이다. 이렇게 하면 원래의 탐욕 알고리즘보다 좀 더 탐색을 많이 하게 된다. 역시 모의실험을 해보도록 하자.

```
actions = c('A', 'B')
rewards = c(A = 0, B = 0)
counts = c(A = 0, B = 0)
values = c(A = 0, B = 0)
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)

epsilon = 0.3  # 무작위 탐색의 확률


for(i in 1:100){

      if(epsilon < runif(1)){  # 활용
        best = names(which.max(values))
      }
      else { # 무작위 탐색
        best = sample(actions, 1)
      }

      print(best)
    R = rnorm(1, mean[best], sd[best])
    rewards[best] = rewards[best] + R
    counts[best] = counts[best] + 1
    values[best] = rewards[best] / counts[best]
}
counts
values
```

입실론 그리디의 경우 초반에는 똑같이 A를 더 많이 선택하지만 결국에는 B를 더 많이 선택하게 되는 것을 볼 수 있다.

입실론 그리디는 간단히 탐색-활용의 균형을 찾을 수 있는 방법이지만 여전히 한계가 있다. 먼저 입실론 값을 지나치게 작게 주면 탐색을 충분히 하지 않기 때문에 차선의 행동에서 빠져나오는데 시간이 오래 걸린다. 그렇다고 입실론 값을 지나치게 크게 주면 최선의 행동을 확실히 아는 상황에서도 불필요하게 탐색을 많이 하는 문제가 생긴다.

11.3.2. 다른 가치 추정 방법

여기서 잠깐 가치를 추정하는 다른 방법을 알아보자. 우리가 지금까지 써온 방법은 보상을 모두 더한 후 횟수로 나누는 방법이다.

$$Q_n = \frac{R_1 + R_2 + \cdots + + R_{n-1} + R_n}{n}$$

그럼 평균의 식을 다음과 같이 바꿔써보자.

$$\begin{aligned}
Q_n &= \frac{R_1 + R_2 + \cdots + R_{n-1} + R_n}{n} \\
&= \frac{R_1 + R_2 + \cdots + + R_{n-1}}{n} + \frac{R_n}{n} \\
&= \frac{n-1}{n}\frac{R_1 + R_2 + \cdots + + R_{n-1}}{n-1} + \frac{R_n}{n} \\
&= \frac{n-1}{n} Q_{n-1} + \frac{R_n}{n} \\
&= \frac{n}{n} Q_{n-1} - \frac{1}{n} Q_{n-1} + \frac{R_n}{n} \\
&= Q_{n-1} - \frac{1}{n} Q_{n-1} + \frac{R_n}{n} \\
&= Q_{n-1} + \frac{1}{n}(R_n - Q_{n-1})
\end{aligned}$$

보상의 평균을 위와 같이 구하면 보상의 합계를 구하지 않더라도 이전의 추정 가치 Q_{n-1}과 현재의 보상 R_n만 있으면 현재의 추정 가치 Q_n을 구할 수 있다. 이 방법으로 다시 모의실험을 해보자.

```
actions = c('A', 'B')
counts = list(A = 0, B = 0)
values = list(A = 0, B = 0)
mean = list(A = 1, B = 2)
sd = list(A = 0.5, B = 1)

epsilon = 0.3


for(i in 1:100){

      if(epsilon < runif(1)){
        best = names(which.max(values))
      }
```

```
      else {
        best = sample(actions, 1)
      }
      print(best)

    R = rnorm(1, mean[best], sd[best])
    Q = values[best] # 기존의 가치

    counts[best] = counts[best] + 1
    values[best] = Q + (R - Q) / counts[best] # 가치 업데이트
}
counts
values
```

11.3.3. 지수이동평균

위와 같이 가치를 추정하는 방법은 합계를 구하지 않아도 된다는 점 외에는 별 장점이 없어보인다. 그런데 여기서 아래 식을

$$Q_n = Q_{n-1} + \frac{1}{n}(R_n - Q_{n-1})$$

다음과 같이 바꿔보자.

$$Q_n = Q_{n-1} + \alpha(R_n - Q_{n-1})$$

이렇게 바꾸면 기존의 가치를 $(1-\alpha)$의 비율로 줄이고, 새로운 보상을 α의 비율로 가치에 추가하게 된다. 이와 같은 평균 계산 방법을 지수이동평균(exponential moving average)라고 한다.

지수이동평균은 일정한 시간이 지나면 과거의 보상이 추정 가치에서 차지하는 비율이 거의 없어지는 것이 특징이다. 극단적으로 α가 1이면 직전의 보상만이 가치 추정에 반영된다.

계속해서 변화하는 상황일 경우 과거의 보상은 더이상 의미가 없기 때문에 지수이동평균을 사용하는 것이 좋다.

```
actions = c('A', 'B')
counts = c(A = 0, B = 0)
values = c(A = 0, B = 0)
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)
epsilon = 0.3

alpha = 0.1  # 가중치

for(i in 1:100){

      if(epsilon < runif(1)){
        best = names(which.max(values))
      }
      else {
        best = sample(actions, 1)
      }
      print(best)
    counts[best] = counts[best] + 1

    R = rnorm(1, mean[best], sd[best])
    Q = values[best]

    values[best] = Q + alpha * (R - Q) # 가치 업데이트
}
counts
values
```

11.3.4. 낙관적 초기화

지수이동평균의 또 다른 장점은 낙관적 초기화(optimistic initial)를 통해 탐색을 촉진할 수 있다는 것이다. 낙관적 초기화는 간단히 말해 초기값을 매우 크게 주는 것이다. 초기값이 크기 때문에 초반에 선택이 적게 된 행동일 수록 선택될 가능성이 높아진다. 따라서 탐색이 촉진된다. 그리고 후반으로 가면서 초기값이 가치에서 차지하는 비중이 점점 사라지므로 실제 가치에 수렴하므로 탐색보다 활용을 많이 하게 된다.

```
actions = c('A', 'B')
counts = c(A = 0, B = 0)
values = c(A = 10, B = 10)  # 낙관적 초기화
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)

epsilon = 0.01 # 입실론이 작아도 초반에는 탐색을 많이 한다

alpha = 0.1

for(i in 1:100){
      if(epsilon < runif(1)){
        best = names(which.max(values))
      }
      else {
        best = sample(actions, 1)
      }
    counts[best] = counts[best] + 1
      print(best)

    R = rnorm(1, mean[best], sd[best])
    Q = values[best]

    values[best] = Q + alpha * (R - Q)
}
counts
values
```

11.4. UCB 알고리즘

입실론 탐욕 알고리즘의 한 가지 단점은 탐색을 할 때 무작위로만 한다는 것이다. 만약 10개의 행동이 있고 이 중에 2개 중 하나가 최선인 상황이라고 해도 입실론 탐욕 알고리즘은 나머지 8개도 모두 공평하게 탐색한다. 이 경우 과도하게 탐색을 많이 하는 문제가 있다.

이를 해결하기 위해 널리 쓰이는 방법 중 하나는 UCB(Upper-Confidence-Bound) 알고리즘이다. 이 알고리즘의 핵심 아이디어는 추정가치에서 일종의 신뢰구간(오차범위)을 구해서 그 구간의 위쪽이 높은 행동을 선택하는 것이다. 수식으로 쓰면 아래와 같다.

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

t는 현재 시점, $N_t(a)$는 현재 시점까지 행동 a를 한 횟수이다. 식의 형태를 보면 신뢰구간의 식 $\mu + c\sqrt{\frac{\sigma^2}{n}}$ 에서 분산 σ^2 대신 $\ln t$를 사용한 것만 다르다.

c는 신뢰구간의 폭을 제어하는데 c를 크게 하면 탐색을 많이 하게 되고 작게 하면 활용을 많이하게 된다. 입실론 탐욕 알고리즘의 입실론과 역할이 비슷한다.

UCB 알고리즘은 추정 가치 Q_t와 이제까지 행동을 한 횟수 $N_t(a)$ 두 가지를 모두 고려하여 탐색을 하므로 좀 더 효율적인 탐색을 할 수 있게 해준다.

```
actions = c('A', 'B')
counts = c(A = 0, B = 0)
values = c(A = 0, B = 0)
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)
control = 2

for(t in 2:101){
    # UCB 알고리즘으로 행동을 선택한다
       best = names(which.max(values + control * sqrt(log(t) / counts)))

    counts[best] = counts[best] + 1
    print(best)

    R = rnorm(1, mean[best], sd[best])
    Q = values[best]

    values[best] = Q + alpha * (R - Q)
}
counts
values
```

11.5. 톰슨 샘플링

톰슨 샘플링(Thompson sampling)은 최근 가장 좋은 성능을 보여주고 있는 알고리즘이다. 구글 어낼리틱스에서 A/B 테스팅 지원 기능에서도 이 알고리즘을 사용한다.

톰슨 샘플링은 가치를 직접 추정하는 대신 가치의 분포를 추정한다.그리고 행동을 선택할 때는 이 분포로부터 가치를 무작위 추출해서 가장 가치가 높은 행동을 선택한다. 그리고 받은 보상으로부터 베이즈 정리를 이용해서 가치의 분포를 업데이트한다.

톰슨 샘플링은 아이디어도 간단하고 성능도 좋지만, 가치의 분포를 추정하고 베이즈 정리를 업데이트하기 위해서는 많은 수학적, 통계학적 지식이 필요하다.

```
actions = c('A', 'B')
counts = c(A = 0, B = 0)
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)
probs = c(A = .5, B = .5)  # A안과 B안에서 가치가 2일 확률

for(t in 1:100){
    # A와 B의 가치를 무작위 추출한다
      values = c(
        ifelse(probs['A'] < runif(1), 1, 2),
        ifelse(probs['B'] < runif(1), 1, 2))

      best = names(which.max(values))
    counts[best] = counts[best] + 1
    print(best)

      R = rnorm(1, mean[best], sd[best])

    # P(R|가치=1) * P(가치=1)
      p1 = dnorm(R, 1) * (1 - probs[best])

    # P(R|가치=2) * P(가치=2)
      p2 = dnorm(R, 2) * probs[best]

    # 베이즈 정리: P(가치=2|R) = P(R|가치=2) * P(가치=2) / P(R)
    probs[best] = p2 / (p1 + p2)
```

```
}
counts
probs
```

11.6. 소프트맥스 알고리즘

소프트맥스 알고리즘은 가치를 추정하지 않는 대신 각 행동의 선호도(preference)를 구한다. 선호도는 $H_t(a)$와 같이 표기한다. 그리고 각 행동들의 선호도를 소프트맥스 함수에 넣어 확률적으로 행동을 선택한다.

$$P(A_t = a) = \pi_t(a) = \frac{e^{H_t(a)}}{\sum_i e^{H_t(a_i)}}$$

이때 기대 보상을 극대화하도록 행동의 선호도에 경사하강법을 적용한다. 그래서 '경사 강도(gradient bandit) 알고리즘'이라고도 한다.

```
actions = c('A', 'B')
counts = c(A = 0, B = 0)
mean = c(A = 1, B = 2)
sd = c(A = 0.5, B = 1)

pref = c(A = 0, B = 0)  # A와 B의 선호도
mean_rewards = 0  # 지금까지 평균 보상
learning_rate = 0.1 # 학습률

for(t in 1:100){
    # 소프트맥스로 확률을 구한다
    e = exp(pref)
    p = e / sum(e)

    # 행동을 확률적으로 선택한다
    chosen = sample(actions, 1, prob=p)

    counts[chosen] = counts[chosen] + 1
    print(best)

      R = rnorm(1, mean[chosen], sd[chosen])

    # 평균 보상을 빼서 +이면 잘한 것이므로 선호도를 높인다
    advantage = R - mean_rewards

    # 경사하강법으로 선택된 행동의 선호도를 업데이트한다
    # 직관적으로 이해한다면 advantage가 크고 확률이 낮을 수록 선호도를 더 높인다
    pref[chosen] = pref[chosen] + learning_rate * advantage * (1 - p[chosen])

    # 경사하강법으로 선택되지 않은 행동의 선호도도 업데이트 한다
    nc = setdiff(actions, chosen)  # not chosen
    pref[nc] = pref[nc] - learning_rate * advantage * p[nc]
}
counts
pref
```

11.6.1. 모의담금질

영화에서 대장간을 보면 철을 뜨겁게 달구고 때린 다음 식히는 것을 볼 수 있다. 이것을 '담금질'이라고 한다. 온도를 높이면 원자들이 더 자유롭게 움직이기 때문에 이것을 이용해서 우리가 원하는 단단한 상태로 만들고서 온도를 낮추는 것이다.

최적화 문제에서도 이와 유사한 방법으로 접근하는 것을 모의담금질(simulated annealing)이라고 한다. 소프트맥스 함수에 온도에 해당하는 τ를 포함시켜보자. 만약 τ가 크다면 모든 행동의 확률이 비

슷비슷하게 된다.

$$P(A_t = a) = \pi_t(a) = \frac{e^{H_t(a)/\tau}}{\sum_i e^{H_t(a_i)/\tau}}$$

그래서 초반에는 온도 τ 높여서 탐색을 촉진하고, 후반에는 온도를 낮춰서 활용을 좀 더 하도록 만들 수도 있다.

사실은 소프트맥스 함수 자체가 원래 물리학에서 물체의 온도와 입자의 상태를 다루는 볼츠만 분포(Boltzmann distribution)에서 가져온 것이다.

11.7. MAB와 지도학습

11.7.1. MAB와 지도학습의 차이

지도학습과 MAB의 차이를 표로 간단히 정리하면 아래와 같다.

	지도학습	MAB
정답	알고 있음	모름
출력	예측	행동
목표	행동과 예측의 차이를 최소화	행동에 대한 보상을 최대화
데이터	학습 전에 주어짐	행동에 따라 달라짐

11.7.2. MAB에 지도학습 적용

지금까지 다룬 MAB는 행동들의 가치가 고정되어 있었다. 그러나 실제로는 맥락에 따라 행동의 가치가 변할 수 있다. 예를 들어 고객에게 상품을 추천하는 것을 MAB로 본다면 각각의 상품이 행동이 되고, 고객이 구매하면 보상을 받게 된다. 그런데 고객의 특성에 따라 구매가능성도 달라질 수 있다. 즉, 행동의 가치가 고청된 것이 아니라 고객의 특성이라는 맥락에 따라 달라지는 것이다.

이렇게 되면 단순히 기존의 보상을 평균내서 행동의 가치를 구하던 것과 달리 고객의 특성으로 행동의 가치를 예측하도록 지도학습을 적용할 수 있다. 이렇게 MAB에 지도학습을 적용한 것을 '맥락 있는 강도(contextual bandit)' 문제라고 한다.

11.7.3. 지도학습에 MAB 적용

반대로 지도학습에 MAB를 적용할 수도 있다. 지도학습을 할 때는 모형, 알고리즘, 하이퍼 파라미터 등등 매우 많은 선택의 갈림길에 서게 된다. 일반적으로 권하는 것은 가능한 다양한 시도를 해보고 가장 좋은 성능을 보여주는 것을 선택하라는 것이다. 그런데 대부분의 경우 시간과 비용의 제약이 있다. 그리고 별로 좋은 성능을 보여주지 않는 방법이라면 굳이 시도해봤자 시간을 낭비하게 된다. 지도학습에서 모형이나 하이퍼파라미터 선택을 행동으로 보고, 학습된 모형의 성능을 보상으로 본다면 여기에 MAB를 적용할 수 있다.