12. 몬테 카를로 법과 응용

12.1. 몬테 카를로 법

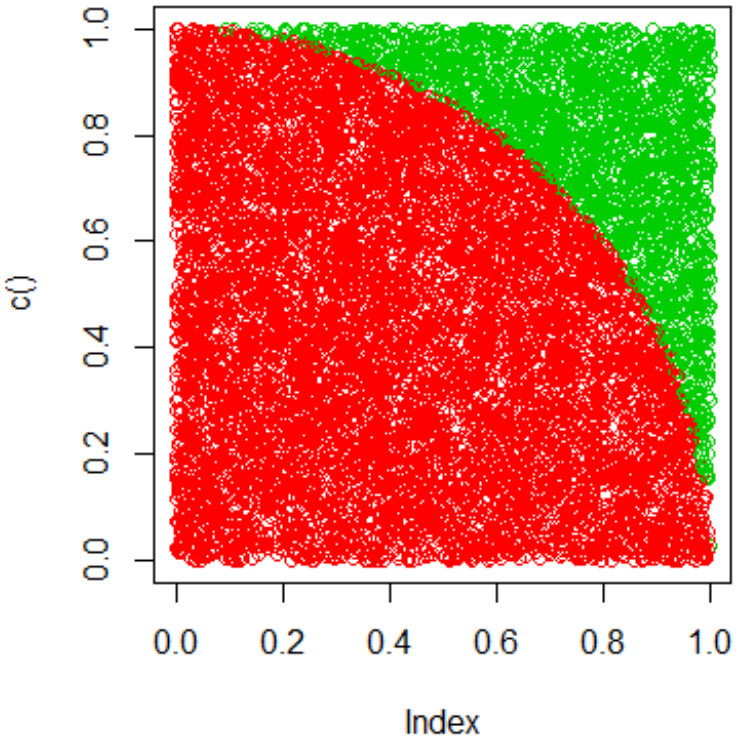
기계학습이나 통계에서는 정해진 공식으로 풀 수 없는 종류의 문제들이 많이 있다. 몬테 카를로 법(Monte Carlo methods: 이하 MC)은 무작위 표본 추출을 반복해서 답을 구하는 방식이다. 몬테 카를로는 모나코 공국에 있는 카지노 이름이다.

간단한 예시로 원의 넓이를 구해보자. 원의 넓이는 πr^2으로 구할 수 있다. 이때 원주율 π를 계산하기는 꽤 까다롭다. 세종 24년(1442년)에 나온 수학책 "칠정산"에서도 원주율은 단순히 3으로 해놓고 있다.

MC를 사용하면 원주율을 몰라도 원의 넓이를 구할 수 있다. 일단 반지름이 1인 원을 생각해보자. 이 원에 외접하는 정사각형을 그리고, 이 정사각형 위로 모래를 뿌렸다고 해보자. 만약 모래를 매우 고르게 뿌렸다면 원 안에 들어간 모래의 비율은 정사각형의 넓이 대비 원의 넓이의 비율과 매우 비슷할 것이다.

아래 코드는 이 원리를 구현한 것이다. 중심이 $(0, 0)$이고 반지름이 1인 원을 그린다. 그리고 $(0, 0)$에서 $(1, 1)$까지 정사각형을 하나 그린다. 그러면 이 정사각형의 넓이는 1이고, 정사각형 안에 포함된 원의 넓이는 $\pi/4$가 된다. 다음으로 x와 y를 각각 0~1 범위에서 무작위 추출한다. 추출된 좌표에서 원의 중심까지 거리가 반지름보다 짧으면 원의 안쪽, 반지름보다 멀면 바깥쪽에 있는 것이다. 그러면 이 경우의 수를 세면 간단히 원의 넓이를 구할 수 있다. 또한 이렇게 구한 원의 넓이를 이용하면 원주율을 역산해낼 수도 있다.

```
total = 100000  # 반복 횟수
radius = 1  # 원의 반지름

x = runif(total)  # x좌표
y = runif(total)  # y좌표
dist = x^2 + y^2  # 원점에서 거리
n = sum(dist < radius)  # 거리 < 반지름인 점 (원의 내부)

n / total # MC로 추정한 원의 넓이
pi / 4    # 실제 원의 넓이

# 시각화
plot(c(), c(), xlim=c(0, 1), ylim=c(0, 1))
points(x[1:10000], y[1:10000], col=ifelse(dist < radius, 2, 3))
```



12.1.1. MC를 이용한 누적 확률 추정

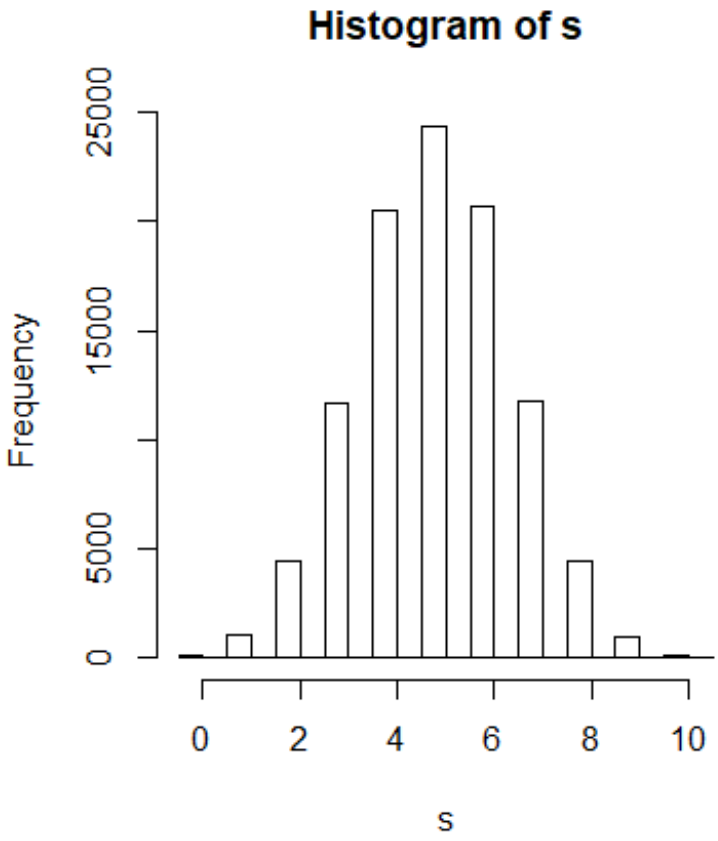
기계학습이나 통계에서 MC는 확률의 합을 구할 때 자주 사용한다. 역시 간단한 예로 이항분포에서 누적확률분포(cumulative probability distribution)를 구하는 경우를 생각해보자.

이항분포는 연속된 n번의 독립시행에서 각 시행이 확률 p를 가질 때의 이산확률 분포이다. 예를 들어 앞면이 나올 확률이 60%인 동전을 10번 던졌을 때, 그 중에 앞면이 x번 나올 확률 같은 것이 이항 분포다.

이항분포의 누적확률분포는 0에서 x까지의 확률을 모두 더한 것을 말한다. 즉, $x = 2$의 누적확률은 $P(x = 0) + P(x = 1) + P(x = 2)$이 된다.

이항분포를 계산하는 법과 그 누적확률분포를 계산하는 방법도 모두 알려져 있지만 실제로 계산하기는 번거롭다. 특히 n이 커지면 더 그렇다. MC를 이용하면 이 문제를 간단히 풀 수 있다. 무작위로 여러 번 해보면 된다.

```
total = 100000
n = 10
p = 0.5

trial = runif(total * n) < p  # 난수를 만들어 p보다 낮은지 확인
m = matrix(trial, ncol = 10)  # 10 x total의 행렬로 만든다
s = rowSums(m)                # 행단위로 합을 구한다

sum(s <= 6) / total  # 6 이하인 사례의 비율
pbinom(6, 10, 0.5)   # 이항 분포에서 6이하의 누적 확률

hist(s) # 히스토그램
```



12.1.2. MCMC를 이용한 누적 확률 추정

이제 좀 더 복잡한 경우를 생각해보자. 이항 분포의 경우에는 0~1까지 범위에서 숫자 하나를 무작위로 뽑고, 이것을 p와 비교하여 작으면 1, 크면 0으로 바꾸는 방식으로 실험을 했다. 즉, 균등 분포에서 간단한 변환을 통해서 이항 분포를 따르는 난수를 무작위로 추출한 것이다.

그런데 다른 분포들에서는 이렇게 변환을 하려면 분위함수(quantile function)을 알고 있어야 한다. 분위함수는 누적확률분포의 역함수이다. x까지의 누적확률이 p면, 분위함수에 p를 넣으면 x가 되는 것이다. 그러면 균등분포에서 p를 뽑고 이를 분위함수에 넣으면 해당 분포를 따르는 난수를 만들 수 있다.

누적확률분포를 만들려면 해당 분포를 따르는 난수가 필요한데, 그런 난수를 만들려면 누적확률분포를 알아야하니 닭과 달걀의 문제인 셈이다.

이런 경우에 적용할 수 있는 방법이 마코프 체인 몬테 카를로 법(Markov Chain Monte Carlo methods: 이하 MCMC)이다. 앞서 예제는 모두 독립 시행으로 표본을 추출했다면, MCMC는 직전 시행에 따라 다음 시행이 달라지는 방법으로 표본을 추출한다. MCMC의 재미있는 점은 이런 시행을 충분히 오래하면 마치 독립 시행으로 추출한 것과 같은 표본을 얻을 수 있다는 점이다.

MCMC는 구체적인 방법에 따라 메트로폴리스-헤이스팅스 알고리즘(Metropolis-Hastings algorithm)과 깁스 샘플링(Gibbs Sampling) 등 여러 가지 알고리즘이 있다.

메트로폴리스-헤이스팅스 알고리즘의 개요는 다음과 같다.

1. 기존 값으로 임의로 정한다
2. 기존 값을 기준으로 제안 분포(proposal distribution)에서 후보 값을 정한다
3. 후보 값의 확률의 기존 값의 확률에 대한 비율(α)을 구한다
4. 0~1까지 난수를 뽑아 α가 크면 후보 값이 새로운 기존 값이 된다.
5. (2)~(4)를 반복한다.

위와 같은 시행을 반복하면 개별 시행은 3~4단계 때문에 직전 시행에 종속적이 된다. 그렇지만 이를 오래 반복하면 확률이 높은 사건이 낮은 사건보다 더 많이 일어나게 되서 결과적으로는 독립시행을 한 것과 비슷한 결과를 얻을 수 있게 된다.

위에서 제안 분포는 무엇이든 될 수 있는데 균등분포를 쓸 수도, 정규분포를 쓸 수도 있다.

이번에는 정규분포의 누적확률분포를 구해보자. 이 분포도 계산 공식이 있기는 하지만 매우 어려운 적분을 풀어야 한다.

$$\int_{-\inf}^{t} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx$$

MCMC로는 상당히 간단하게 구할 수 있다.

```
x = 0  # 시작값을 정한다
n = 0
total = 100000  # 총 샘플링 횟수
xs = rep(0, total)

for(i in 1:total){
  # 기존 값 대비 ±0.5 범위에서 후보 값을 정한다
  delta = runif(1) - 0.5
  x.new = x + delta

  # 확률분포에서 기존 값보다 후보 값이 얼마나 더 확률이 높은지(alpha) 구한다
  alpha = dnorm(x.new) / dnorm(x)

  # 기존값에 그대로 머무르거나, 후보 값으로 넘어간다
  # alpha가 클 수록 후보 값으로 넘어갈 가능성이 커진다
  x = ifelse(runif(1) < alpha,  # 0~1까지 난수를 alpha와 비교해서
             x.new,             # alpha가 크면 후보 값으로 넘어간다
             x)                 # alpha가 작으면 기존 값에 머무른다

  xs[i] = x
}

sum(xs < -2) / total  # MCMC로 구한 누적확률
pnorm(-2)  # 실제 누적확률

hist(xs) # 히스토그램
```

12.2. 부트스트랩

지금까지는 확률분포 자체는 알고 있다는 가정 아래 MC를 사용했다. 그럼 확률분포도 모르는 상황에서는 어떻게 할까? 이럴 때 사용할 수 있는 방법을 부트스트랩(bootstrap)이라고 한다. 부트스트랩은 실제 데이터를 반복 복원 추출해서 분포를 추정하는 방법이다.

12.2.1. 신뢰구간 추정

예를 들어 `iris` 데이터에서 평균의 신뢰구간을 구하고 싶다고 해보자. 부트스트랩은 `iris` 데이터에서 반복 복원 추출을 통해 여러 개의 샘플을 만들고 각 샘플들의 평균을 구해서 신뢰구간을 구한다.

구체적인 방법에 따라 신뢰구간은 여러 가지가 나오지만 t 분포를 이용해 구한 신뢰구간과 비슷하다는 것을 알 수 있다. 표본평균이 t 분포를 따르기 때문이다.

부트스트랩은 신뢰구간을 구하는 경우 이외에도 다양하게 활용할 수 있다는 것이다. 이를 앙상블에 활용한 것이 바로 배깅(bagging = bootstrap aggregation)이다. 즉, 부트스트랩으로 만든 각 표본으

로 모형들을 학습시키고 이 모형들의 예측을 모으는 방법이다.

```
data(iris)
x = iris$Petal.Length

# 부트스트랩을 이용해 95% 신뢰구간 구하기
library(boot)
b = boot(x, function(d, i){ mean(d[i]) }, R = 2000)
quantile(b$t, c(.025, .975))  # 2.5% ~ 97.5%

# t 분포를 이용해 95% 신뢰구간 구하기
n = length(x)
q = c(qt(.025, df=n), qt(.975, df=n))
mean(x) + q * sd(x) / sqrt(n)
```

12.2.2. 평균 차이 검정

이번에는 두 집단의 평균 차이를 검정해보자. 통계학에서 일반적으로 하는 방식은 통계적 가설 검정을 사용한다. 두 집단이 실제로는 같은 분포에서 나왔다고 가정하고(귀무가설), 관찰된 평균의 차이가 단순히 표집 오차 때문에 나올 확률은 매우 낮다는 사실을 보여줌으로써(귀무가설 기각) 두 집단 사이에 통계적으로 유의미한 차이가 있다는 것을 보여주는 것이다.

부트스트랩으로도 똑같이 가설 검정을 할 수 있다. 두 집단의 데이터를 뒤섞은 다음, 무작위로 두 집단으로 나눠서 평균의 차이를 구해보는 것을 여러 번 반복하는 것이다. 이렇게 해서 관찰된 정도의 평균 차이가 얼마나 나타날 수 있는지 확인해본다.

```
# iris데이터에서 두 품종의 꽃받침 길이를 뽑는다
data(iris)
y = iris$Petal.Length
x = iris$Species

versicolor = y[x == 'versicolor']
virginica = y[x == 'virginica']

# 두 집단의 평균 차이
mean(versicolor)
mean(virginica)
diff = mean(versicolor) - mean(virginica)

# t 검정
t.test(versicolor, virginica)

# 부트스트랩을 이용한 비교
library(boot)
n1 = length(versicolor)
n2 = length(virginica)
b = boot(c(versicolor, virginica),
     function(x, i){
       g1 = i[1:n1]               # 집단 1
       g2 = i[(n1+1):(n1+n2)]     # 집단 2
       mean(x[g1]) - mean(x[g2]) # 평균 차이
     }, R = 1000)
b
quantile(b$t, c(.025, .975))  # 평균 차이의 95%가 어떤 범위인지 확인
```

12.3. 트리 탐색

하나의 선택이 다음의 선택을 제약하는 경우가 있다. 문장을 만들 때 앞의 단어를 정하면 그 뒤에 이어서 나올 수 있는 단어는 한정이 된다. 게임에서도 초반에 잘못하면 후반에는 복구를 할 수 없기도 한다. 이런 식으로 선택에 선택이 이어지는 문제를 트리 탐색(tree search)의 형태로 푼다.

트리 탐색의 어려움은 가지를 한 번 쳐 나갈 때마다 경우의 수가 급격히 증가해서 모든 경우의 수를

다 살펴볼 수 없다는 것이다. 한 번에 10개의 가지만 쳐도 10단계만 지나가면 10억가지 경우의 수가 생긴다.

12.3.1. 바이그램을 이용한 문장 생성

실습에 사용할 데이터는 제인 오스틴의 소설 텍스트이다. 텍스트에서 한 단어 뒤에 다음 단어가 나올 확률을 구한 것을 바이그램(bigram)이라고 한다.

```
install.packages(c('dplyr', 'tidytext', 'janeaustenr'))

library(dplyr)
library(tidytext)
library(tidyr)
library(janeaustenr)

austen_bigrams <- austen_books() %>%
  unnest_tokens(bigram, text, token = "ngrams", n = 2) %>%
  count(bigram, sort = TRUE) %>%
  separate(bigram, c("word1", "word2"), sep = " ")
```

바이그램을 이용하면 간단한 텍스트 생성 모형을 만들 수 있다. 예를 들어 한 문장에 10단어가 있다고 하면, 이 문장의 확률은 첫째 단어와 둘째 단어의 확률, 둘째 단어와 셋째 단어의 확률 등등을 순서대로 곱한 것과 같게 된다.

12.3.2. 탐욕 알고리즘

바이그램 문장 생성은 일종의 트리 탐색이다. 한 단어 뒤에 나올 수 있는 단어가 여러 가지 있고, 그 단어 뒤에 나올 수 있는 단어도 여러 가지가 있기 때문이다. 가장 쉽게 사용할 수 있는 것은 탐욕 알고리즘이다.

트리 탐색에서 탐욕 알고리즘은 이후의 일은 생각하지않고 각 단계에서 최선의 선택을 한다.

```
start <- 'she'  # she로 시작하는 문장을 만든다

word <- start
words <- c()

for(i in 1:10){
  words <- c(words, word)

  # 현재 단어 다음에 나올 확률이 가장 높은 단어를 고른다
  rel <- austen_bigrams %>%
    filter(word1 == word) %>%
    filter(n == max(n))
  word <- rel$word2
}

# 문장을 합친다
paste(words, collapse=' ')
```

12.3.3. 빔 탐색

탐욕 알고리즘은 매우 간단하고 속도도 빠르지만 지나치게 근시안적이다. 이를 개선한 것이 빔 탐색(beam search)이다. 빔 탐색은 챗봇, 번역기 등에서 문장을 생성할 때 가장 널리 쓰이는 방법 중에 하나다.

빔 탐색은 정해진 갯수의 후보를 검토한다. 이 후보들의 다음 단어를 고르고 나면 이들을 순서대로 정렬해서 정해진 갯수만큼만 남긴다.

```
# 빔 탐색 함수
```

```
beam.expand <- function(word, CLP, sentence){
      # word: 탐색할 단어
      # CLP: 누적 로그 확률(cumulative log probability)
    # sentence: 앞에서 만들어진 문장

    # 후보 단어들을 찾는다
      d <- austen_bigrams %>%
        filter(word1 == word)

    # 사용빈도의 합을 구한다
      total = sum(d$n)

    # 가장 많이 쓰인 3단어를 고른다
    d <- d %>%
        arrange(desc(n)) %>%
        slice(1:3)

    # 로그 확률과 누적 로그 확률을 계산한다
    d$LP <- log(d$n / total)
    d$CLP <- CLP + d$LP

    # 문장 뒤에 단어를 추가한다
    d$sentence <- paste(sentence, d$word2)

    # 데이터 프레임을 넘긴다
    d
}
```

이제 위의 함수를 활용하여 전체 탐색은 아래와 같이 진행된다.

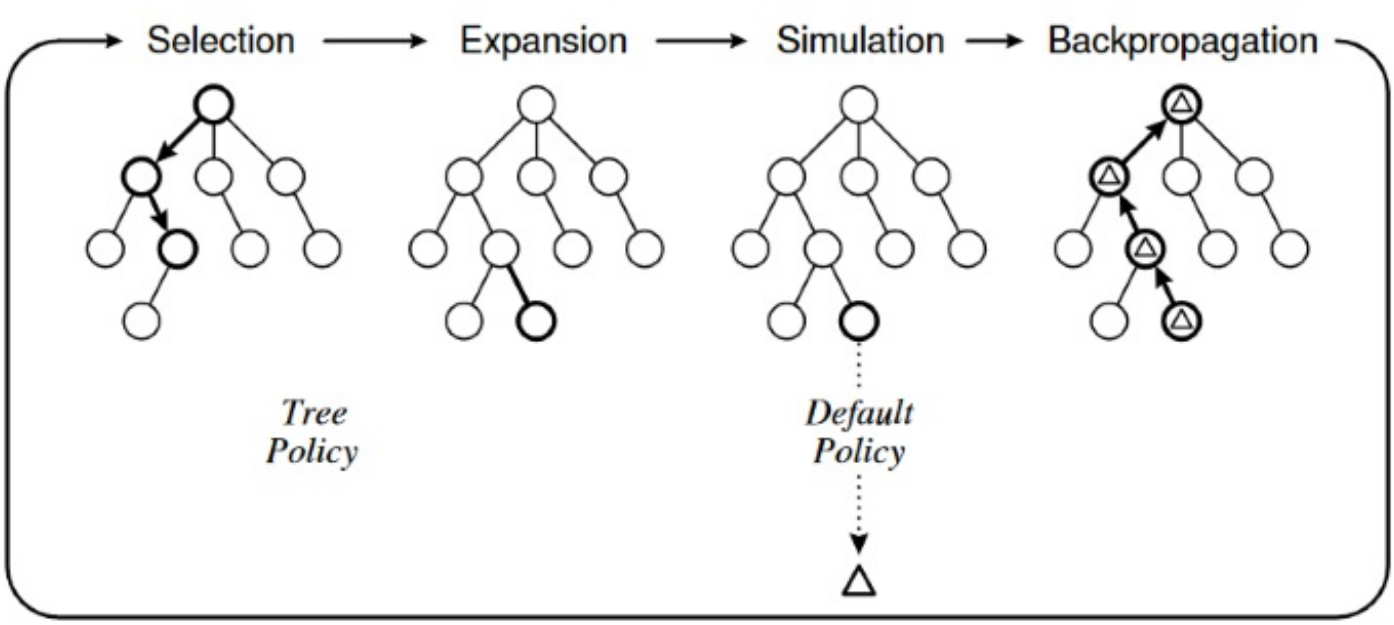
```
# 빔 탐색
CLP = 0

# 시작 단어로 시작한다
start = 'she'
d <- beam.expand(start, 0, start)

for(j in 1:8){
      df = data.frame()

    # 3개의 후보 단어의 후보를 뽑아낸다
      for(i in 1:3){
        word <- d$word2[i]
        CLP <- d$CLP[i]
        sentence <- d$sentence[i]
        branch = beam.expand(word, CLP, sentence)
        df = rbind(df, branch)
      }

    # 9개의 후보 중 CLP가 가장 높은 3개의 후보로 좁힌다
      d <- df %>% arrange(CLP) %>% slice(1:3)
}

# 가장 확률 높은 문장
d$sentence[1]
```

12.3.4. 몬테카를로 트리탐색

대부분의 경우 문장생성은 빔 탐색 정도로도 충분하다. 하지만 빔 탐색도 여전히 근시안적이라는 문제가 있다. 바둑 같은 게임에서는 수 십 수 앞을 내다봐야할 경우도 있다. 이럴 때 사용할 수 있는 것이 몬테카를로 트리탐색(Monte Carlo Tree Search: 이하 MCTS) 이다. 알파고가 사용한 알고리즘이 이것이다.

탐욕 알고리즘이나 빔 탐색은 현재까지 가장 좋은 것을 선택하지만 MCTS는 앞으로 가장 좋을 것을 선택한다. 트리 탐색에서는 앞으로 일어날 모든 경우를 따져보는 것은 불가능하므로 MCTS에서

는 일부의 경우만 표본 추출해서 그 추정치로 삼는다.



MCTS 알고리즘은 다음과 같이 진행된다.

(1) 선택: 시작점에서부터 이미 가치를 추정한 행동들을 하나씩 고른다.

(2) 확장: 행동을 골라나가다가 아직 가치를 추정하지 않은 행동이 나오면 그 중 하나 또는 여럿을 고른다.

(3) 모의실험: 도착점까지 행동들을 여러 번 모의실험 해서 행동의 가치를 추정한다.

(4) 역전파: 추정된 행동의 가치를 바탕으로 이전 행동들의 가치를 조정한다.

여기서 (1)-(2) 단계에서 행동을 선택/확장하는 방법을 트리 정책(tree policy), (3) 단계에서 모의실험을 하는 방법을 기본 정책(default policy)라고 한다.

트리 정책은 이미 가치를 추정한 행동들 중에 하나를 고르기 때문에 MAB에서 행동을 고르는 것과 비슷하다. 그래서 MAB의 UCB를 변형한 UCT 알고리즘을 많이 사용한다. 아래 식에서 n_p는 부모 노드의 검토 횟수, n_j는 자식 노드의 검토 횟수이다.

$$\text{UCT} = \bar{X}_j + c\sqrt{\frac{\ln n_p}{n_j}}$$

```
uct.policy <- function(tree){
      node = 0
      for(i in 2:10){
        # 자식(다음에 할 행동)을 찾는다.
           children <- tree %>%
              filter(parent == node)

         # 자식이 없으면 중단
        if(nrow(children) == 0){
              break
        }

        # 자식 중에 가치의 상한이 가장 높은 것을 고른다
        selected <- children %>%
              mutate(uct = value + control * sqrt(log(1)/visit)) %>%
```

```
              arrange(desc(uct)) %>%
              slice(1)

        node <- selected$id
      }
      node
}
```

확장의 코드는 간단하다.

```
mcts.expand <- function(word, CLP, parent){
  tree <- beam.expand(word, 0, '')
  tree = tree[c('word1', 'word2', 'CLP')]
  tree$value = 0
  tree$visit = 1
  tree$parent = parent
  tree
}
```

기본 정책은 자식 중에 확률이 가장 높은 것 중 하나를 무작위로 고를 것이다.

```
random.policy <- function(tree, node){
      depth <- tree$depth[node]

    for(i in depth:10){
        word <- tree$word2[node]
        CLP <- tree$CLP[node]

        # 트리를 확장한다
         branch <- mcts.expand(word, CLP, parent = node)
        branch$id <- nrow(tree) + 1:3
        branch$depth <- i
        tree <- rbind(tree, branch)

        # 다음 행동을 무작위로 고른다
        node <- sample(branch$id, 1)
      }
      tree
}
```

MCTS는 모의실험을 통해 각 행동의 가치를 추정하기 때문에 해당 문제 영역에 대한 **영역 특수적 지식**(domain-specific knowledge)이 필요 없다. 즉, 바둑에 대한 전문지식이 없더라도 MCTS를 이용하면 바둑을 잘 두는 인공지능을 만들 수 있다. 물론 영역 지식을 활용하면 기본 정책을 개선하여 성능을 더 향상시킬 수도 있다.

다음으로 역전파를 구현하자.

```
backprop <- function(tree){
    # 마지막 3개의 결과의 평균을 구한다
     last = tail(as.numeric(rownames(tree)), 3)

      tree$value[last] = tree$CLP[last]
      R = sum(tree$value[last]) / 3

      parent <- tail(tree$parent, 1)

    # 부모로 거슬러올라가면서 가치를 반영한다
      while(parent != 0){
        n = tree$visit[parent]
        tree$visit[parent] <- n + 1
        Q = tree$value[parent]
        tree$value[parent] = Q + (R - Q) / n
        parent <- tree$parent[parent]
      }
      tree
```

```
}
```

전체 과정은 아래와 같이 진행된다. MCTS의 또 다른 장점은 모의실험 결과가 즉시 역전파되어 가치 추정에 반영되기 때문에 탐색 시간이 허용하는 범위까지 모의실험을 충분히 할 수 있다는 것이다. 즉, 탐색 시간이 길어지면 길어지는대로 모의실험을 많이 해서 성능이 향상되지만, 탐색 시간이 짧아지더라도 최소한의 탐색시간만 주어지면 탐색에 실패하는 경우는 없다.

```
start <- 'she'
tree <- mcts.expand(start, 0, parent = 0)
tree$id <- 1:3
tree$depth <- 1
control = 2

# 모의실험을 10회 반복한다
for(i in 1:10){
    node <- uct.policy(tree)
    tree <- random.policy(tree, node)
    tree <- backprop(tree)
}

# 가장 좋은 결과를 고른다
best_end = tree %>%
  filter(depth == 10) %>%
  filter(value == max(value))

# 거슬러 올라가며 부모를 찾는다
nodes = c()
node <- as.numeric(best_end['id'])
tree$value[node]

while(node != 0){
    nodes = c(node, nodes)
    node <- tree$parent[node]
}

# 부모에서부터 끝까지 단어를 합쳐 문장을 만든다
paste(c(start, tree$word2[nodes]), collapse=' ')
```

12.3.5. MCTS와 MAB

MCTS는 MAB와 몇 가지 비슷한 점들이 있다. MCTS도 가치가 가장 높은 행동(action)을 고르는 문제다. MCTS와 MAB의 차이점은 아래 표와 같다.

	MAB	MCTS
행동들 사이의 관계	관련 없음	앞 행동에 따라 뒷 행동이 달라짐
보상이 주어지는 시점	행동 직후	일련의 행동을 마친 후에
보상을 알 수 있는 방법	실제로 행동을 해봐야 알 수 있음	모의실험으로 알 수 있음

12.3.6. MCTS와 알파고

2016년 구글 딥마인드가 공개한 알파고는 한국의 이세돌 9단을 꺾었다. 그 후로 지금까지도 한국에서 '알파고'는 인공지능의 대명사로 쓰이고 있다.

알파고는 다음과 같은 과정을 통해 만들어졌다. 1단계는 대량의 기보로부터 트리 정책에 사용할 정책망과 기본 정책에 사용할 롤아웃망을 지도학습시킨다. 이 두 가지는 모두 주어진 상황에서 '다음 수'를 예측한다. 모의실험의 속도를 높이기 위해 롤아웃망은 단순하게 만든다.

2단계는 정책망 대 정책망으로 바둑을 여러 번 두게 한다. 이 게임의 결과를 바탕으로 정책망을 "다음 수의 정확한 예측" 대신 "게임의 승리"를 최대화하도록 경사하강법으로 학습시킨다. 이미 이 상

태에서도 기존의 바둑 프로그램을 상대로 80% 이상 승리를 거두게 되었다.

3단계는 추가 훈련된 정책망으로 둔 바둑을 바탕으로 주어진 상황에 대해 승패를 예측하는 가치망을 학습시킨다.

실제 시합에서는 롤아웃망, 정책망, 가치망을 바탕으로 MCTS를 수행한다. 우선 트리 정책은 추정된 가치에 정책망이 예측한 확률을 검토 횟수로 나눈 것을 더하여 최대값인 것을 선택한다. 즉, 가치가 높고 다음에 둘 가능성이 높은 수를 선택하지만 검토를 많이 했으면 추정된 가치만으로 선택을 하는 것이다.

다음으로 가치 추정은 롤아웃망을 이용해 모의실험을 한 결과와 가치망의 예측을 더해서 반으로 나누는 방법을 쓴다.

2017년 공개된 알파고 제로는 기존의 알파고를 더욱 단순하게 만들었다. 롤아웃망과 기본 정책, 그리고 인간의 기보를 지도학습시키는 단계를 모두 없애버렸. 정책망과 가치망도 하나의 신경망으로 통합했다.

알파고 제로는 MCTS에서 트리 정책은 알파고와 유사하다. 대신 모의실험을 하지 않고 신경망이 예측한 가치를 그대로 사용한다. 알파고 제로의 성능은 이세돌과 맞붙었던 알파고를 100번 붙어 100번 이기는 수준이다. 그 대신 알파고 제로를 학습시키기 위해서는 현재로서 구글만 보유하고 있는 고성능의 계산장치가 대량으로 필요하다.