随机对照试验的贝叶斯方法

作者： *罗子俊*

前言

随机对照试验（Randomized Controlled Trial，RCT）是通过实验来建立因果关系的黄金标准，在新药临床试验，社会科学，以及经济学等领域的实验中都有广泛的应用。在商业特别是电子商务应用中，RCT被称为A/B/N测试。 RCT和A/B/N测试的主要思想非常简单：将许多个体随机分组，然后比较和评估结果，以找出哪种治疗方案的效果更好/最好。RCT中通常包括一个接受“安慰剂”的对照组。需要注意的是，安慰剂也应该被视为一种治疗方案，因为接受安慰剂的个体并不是“什么都没有得到”。安慰剂没有治疗效果，因为它不是为了治疗疾病或疾病而设计的。但是，安慰剂可能会对接受它的个体产生心理作用从而对病情产生积极的影响。认为RCT中接受安慰剂的对照组“效果为零”是错误的。

在本文的接下来的大部分内容当中，我会用A/B/N测试作为例子，因为我想避免RCT的许多细节。我们会在文章的最后再回到RCT这个题目上。我用“A/B/N”这个说法来包括测试多个版本的情形。如果你只比较两个版本，那么就只是A/B测试了。

当我在2022年面试数据科学家职位时，其中一个面试问题是：我们将在客户网站上运行A/B测试，有什么办法可以确定我们的实验需要运行多久？那时候，我只知道如何根据统计学中的假设检验来计算最小样本量，因此我用的就是这个思路来回答。但是，我在回答的被迫停下来了，因为我忽然发现了一个以前没有仔细想过的问题：在进行实验之前，我怎么知道标准差的值？标准差是计算样本大小所需的数值之一。我的面试从那开始走下坡，最后我没有得到那份工作。但是面试官很好心地告诉我，让我看看“功效分析（Power Analysis）”。

以下是我后来所了解到的功效分析，其实和假设检验是一回事。假如你有一个电子商务网站，可以使用两种不同的配色方案。为了了解哪种配色方案更能促进销售量，你可以随机分配访客到网站的两种配色中的一个。在这个过程当中，你要记录每个访客所看到的版本以及他们的消费量。对于 $i\in(A,B)$，我们定义以下数值：

- $\bar{x}_i$：方案 i 的访客预期消费量；
- n_i：方案 i 的访客数量；
- s_i：方案 i 的访客花费消费的标准差。

我们可以通过以下的公式计算“功效（Power）”： $$t=\frac{\bar{x}_A-\bar{x}_B}{s_p\sqrt{\tfrac{1}{n_A}+\tfrac{1}{n_B}}}$$

其中 $s_p=\sqrt{\frac{(n_A-1)s_A^2+(n_B-1)s_B^2}{n_A+n_B-2}}$ 是合并样本标准差。 "功效" t 遵循自由度为 n_A+n_B-2 的 t-分布。

假设 A/B 测试中两个方案的标准差相等（ $s_A=s_B$ ），那么我们可以将它表示为 s。为了简单起见，假设 $n_A=n_B$。从上面的功效分析公式中我们可以解出 n_i： $$N=\frac{4t^2s^2}{(\bar{x}_A-\bar{x}_B)^2}$$ 其中 N 是总样本量（ n_A+n_B ）。很容易看出，如果

- 两个版本之间的预期差异较小；
- 你想要更好的显著性水平，例如 1% 而不是 5%；
- 标准差较大；

那么你将需要更大的样本量。

但是问题在于：你在实验之前并不知道 $\bar{x}_i$ 和 s_i 的值。对于 $\bar{x}_i$，问题倒不是很大，因为你可以指定期望值的差。譬如，网站当前正在运行版本 A，你知道 $\bar{x}_A=50$。而你所关心的只是版本 B 能否将预期的花费提高到 65 元。换句话说，$\bar{x}_B-\bar{x}_A=15$。但即使如此，你仍然需要知道标准差。怎么办？有一个建议是跑一个短暂的实验来估计标准差。但难道 A/B 测试本身不就是一次实验吗？

以下是经典A/B测试设计的另一个问题。我成为另一家公司的数据科学家后，有机会参与一个A/B测试。问题在于，根据功效分析，这个实验需要至少运行3个月。但我们没有那么多时间。在实验开始1个月后，我们的模型（版本B）的表现优于现有模型（版本A）。我们可以以此宣称我们的模型更好吗？根据经典的A/B测试设计，答案是显示是“不行”，因为我们不应该对结果进行“窥视（peeking）”，因为短期内的显著差异可能是随机因素造成的。

可是在新药临床试验中，这种“不准窥视”的规则可能会引发严重的问题。如果一种药物在前500个患者中证明了其有效性，但根据功效分析，需要在5万名患者身上进行测试，你会怎么做？继续给可能从药物中受益的患者使用安慰剂难道不是一种不道德的行为吗？

这两个问题一直困扰着我，直到我后来了解到A/B/N测试中可以使用贝叶斯方法。下面是A/B/N测试中的贝叶斯方法的工作原理。与预先确定样本量不同，A/B测试是在实时中部署的。如果我们延续之前使用的网站例子，一个访客在第一次访问时会被随机分配到网站的其中一个版本。在实践中，最好确保每个版本最初都得到一些访问者，譬如在最开始的50个访客上，每个版本都有相等的概率被分配到。但从那里开始，收到更高消费额的版本应该获得更多的访客。但这并不意味着另一个版本被抛弃。我们面临的是探索-利用权衡（Explore-Exploit Tradeoff）。

探索-利用权衡

简单来说，探索-利用权衡显示了一个悖论：为了找到最好的版本，我们需要探索。这意味着，花费越多时间去尝试不同版本，探索的结果必然会变得更好。然而，为了最大化总体回报，一旦找到最佳版本，我们希望能一直用它。这意味着，花费越多时间去尝试不同版本，开发的结果必然会变得更糟，因为有且只有一个最佳版本。

如何处理探索-利用权衡构成了算法之间的核心差异。一些算法，如“贪婪算法”家族的变体，更擅长利用。它们的缺点是很容易“陷入”次优版本。这一点，我们讨论“贪婪算法（Epsilon Greedy）”时会再讲到。其他算法，如“乐观初始值（Optimistic Initial Values”，更擅长探索。

如果你阅读这篇文章的原因是因为你认为它可能对你的研究项目有帮助，那么你肯定不会对探索-利用平衡的问题感到陌生。我记得有一次我跟研究生时期的一位教授聊天，我问他，是否应该放弃那些我认为不会被发表在好期刊上的项目。他回答说：但你怎么知道呢？他说的是有道理：这位教授从未发表过他的博士论文中的任何一章。他的成功的发表都是在他探索了新的研究领域之后才出现的。然而，大约在他博士毕业15年后，一篇与他在博士论文中研究的主题非常相关的论文在一个顶级期刊上发表了。回想起来，他可能过度探索了。这可能是他建议我更多地“利用”的原因。

贪婪算法（Epsilon Greedy）

我们会从贪婪算法讲起。在每个算法的讨论中，我打算涉及一下内容：

- 理论和直觉
- 伪代码
- Python代码

我是从Udemy的一门名为Bayesian Machine Learning in Python: A/B Testing的课程中学习到这些算法的。虽然课程但中也有提供Python代码，我在文章里所展示的代码是我根据算法理论写出来后再根据课程所提供代码进行修改，以使得代码使用起来更有效率。

贪婪算法都有一个简单得逻辑：选择历史预期回报最高的版本。为了简单期间，我们可以考虑一个电子商务网站。这个网站有5种不同的设计，但是只销售一个产品：一个69.99元的可随身携带的乐器。如果我们在这个网站上进行A/B/N测试，每个访客只会有两个可能得结果：购买产品，或者不购买产品。

虽然不是必需的，但在开始时我们可以把所有5个算法都尝试一遍。譬如，我们以同等的概率将前50个访问者分配到每个设计中。从第51个访客开始，算法找出预期回报最高的版本，并将访客分配到这个版本。以下是一个简单的贪婪算法的伪代码：

```
for i in [1, 50]:
    choose each bandit randomly
loop:
    j = argmax(expected bandit win rates)
    x = T/F from playing bandit j
    bandit[j].update_mean(x)
```

在这里，我用了"土匪（bandit）"而不是"版本（version）"，因为我们正在处理的问题在概率论和机器学习中被称为"多臂老虎机问题（Multi-Armed Bandits）"。这个类比来自于在赌场中选择老虎机，因为一个单独的老虎机被称为"单臂土匪（Sing-Armed Bandit）"。

在上面的伪代码中，i 代表访客，j 代表网站版本（或老虎机），如果访客购买了产品，则x 取值为1，否则为0。此外，update_mean() 是一个函数，它会获取 x 的最新值并更新赌徒机器 j 的期望回报。我们可以用以下的公式来更新老虎机 j 在被玩了 n 次后的期望回报： $$\bar{x}*n=|frac{|bar{x}*{n-1}\times(n-1)+x_n}{n}$$

这个公式可以在常数时间下计算均值。这意味着无论n的值是什么，它计算均值只需要用到三个数值：$\bar{x}_{n-1}$，x_n，以及 n。而我们常用的计算均值的公式 $$\bar{x}*n=|frac{|sum{i=1}^n{x_i}}{n}$$ 在计算均值时所需要的数值会随着n的增加而增加。

很明显，上面的贪婪算法存在一个问题：一旦找到了一个具有足够高回报的老虎机，它就很少会切换。换句话说，它几乎不会进行探索。Epsilon Greedy 提供了一个简单的修正方案：

```
for i in [1, 50]:
    choose each bandit randomly
loop:
    p = random number in [0, 1]
    if p < epsilon:
        j = choose a bandit at random
    else:
        j = argmax(expected bandit win rates)
    x = T/F from playing bandit j
    bandit[j].update_mean(x)
```

在Epsilon Greedy中，每次有一个新的访客，系统会抽取一个随机数。如果这个随机数小于事先设定的阈值epsilon，系统会随机选取一个老虎机。这个老虎机有可能跟argmax所选取的最优老虎机一样。我们可以多写几行代码来排除这个情况。

接下来我们看看如果在Python当中实现Epsilon Greedy算法。需要注意的是，代码当中会出现一些"*only in demonstration*"的注释。这些注释意味着相关的代码是用于构造概率的真实值的。在实际应用当中，你并不会知道这些真实值。

```
import numpy as np
import random

# set the number of bandits
N_bandits = 5
# set the number of trials/visitors
# only in demonstration
N = 100000
# set the number of trials to try all bandits
N_start = 50

class BayesianAB:
  def __init__(
      self,
      number_of_bandits: int = 2,
  ):
    self.prob_true = [0] * number_of_bandits # only in demonstration
    self.prob_win = [0] * number_of_bandits
    self.history = []
    self.count = [0] * number_of_bandits
    self.a = [1] * number_of_bandits
    self.b = [1] * number_of_bandits

    # set the last bandit to have a win rate of 0.75 and the rest lower
    # only in demonstration
    self.prob_true[-1] = 0.75
    for i in range(0, number_of_bandits-1):
      self.prob_true[i] = round(0.75 - random.uniform(0.05, 0.65), 2)

  # Receives a random value of 0 or 1
  # only in demonstration
  def pull(
      self,
      i,
  ) -> bool:
    return random.random() < self.prob_true[i]

  # Updates the mean
  def update(
      self,
      i,
      k,
  ):
    outcome = self.pull(i)
```

```
    # may use a constant discount rate to discount past
    self.prob_win[i] = (self.prob_win[i] * k + outcome) / (k+1)
    self.history.append(self.prob_win.copy())
    self.count[i] += 1

  ####################
  # epsilon greedy
  def epsilon_greedy(
      self,
      epsilon: float, # decay epsilon?
  ) -> list:

    self.history.append(self.prob_win.copy())

    for k in range(0, N_start):
        i = random.randrange(0, len(self.prob_win))
        self.update(i, k)

    for k in range(N_start, N):
      # find index of the largest value in prob_win
      i = np.argmax(self.prob_win)

      if random.random() < epsilon:
        j = random.randrange(0, len(self.prob_win))
        # If the randomly picked bandit is the same as one from argmax,
pick a different one
        while j == i:
          j = random.randrange(0, len(self.prob_win))
        else:
          i = j

      self.update(i, k)

    return self.history
```

我们可以看看这个代码的细节。首先，我们导入两个库：numpy和random。我们会用到这两个库里面的函数，譬如numpy当中的argmax以及random当中的randrange：

```
import numpy as np
import random
```

然后，我们设定三个全局参数：

```
# set the number of bandits
N_bandits = 5
# set the number of trials/visitors
N = 100000
# set the number of trials to try all bandits
N_start = 50
```

在实际应用中，N_bandits的值将取决于你使用中所测试的版本数，而访客数，N，则是未知的。

在这个代码中，我们将会构造一个名为的BayesianAB的类（class）。这篇文章里的所有算法，都会放在这个类的下面。在类的最开始，我们初始化以下数值：

```
class BayesianAB:
  def __init__(
      self,
      number_of_bandits: int = 2,
  ):
    self.prob_true = [0] * number_of_bandits # only in demonstration
    self.prob_win = [0] * number_of_bandits
    self.history = []
    self.count = [0] * number_of_bandits
    self.a = [1] * number_of_bandits
    self.b = [1] * number_of_bandits
```

BayesianAB 默认有两个老虎机。我们预分配六个列表来储存算法所需要用到的值：

- prob_true 用来储存每个老虎机概率的真实值。在实际应用当中，这些真实值是未知的；
- prob_win 用来储存每个老虎机概率的计算值。这个值会在每一轮当中更新；
- history 用来储存每一轮所产生的 prob_win 值。这些值既可以用于计算下一轮的均值，也可以用来在最后评估算法的效果；
- count 用来储存每个老虎机所被使用的次数；
- a and b 是 Thompson Sampling 或 Bayesian Bandit 所需要用到的数值。这个算法会在后面讨论。

下面几行代码构建老虎机概率的真实值：

```
    # set the last bandit to have a win rate of 0.75 and the rest lower
    # only in demonstration
    self.prob_true[-1] = 0.75
    for i in range(0, number_of_bandits-1):
      self.prob_true[i] = round(0.75 - random.uniform(0.05, 0.65), 2)
```

其中，最后一个老虎机有最高的胜率，在这里设置为.75。剩下的老虎机胜率在.1和.7之间。我们也可以直接指定几个胜率，但是我更喜欢我在这里用的方法，因为这样我就可以允许概率值的数目随着N_bandits的值而改变。

接下来，我们定义两个函数。除了Thompson Sampling这个算法以外，其他的算法都会用到这两个函数：

```
  # Returns a random value of 0 or 1
  # only in demonstration
  def pull(
      self,
      i,
```

```
    ) -> bool:
      return random.random() < self.prob_true[i]

    # Updates the mean
    def update(
        self,
        i,
        k,
    ):
      outcome = self.pull(i)
      # may use a constant discount rate to discount past
      self.prob_win[i] = (self.prob_win[i] * k + outcome) / (k+1)
      self.history.append(self.prob_win.copy())
      self.count[i] += 1
```