Machine Learning for Economics and Business

Zijun Luo

2023-06-25

Contents

Preface

The primary purpose of these articles, which I started to write in May 2023, is to review, summarize, and organize what I have learned in Machine Learning as a PhD economist. I have taken several Machine Learning related courses on Coursera and Udemy. In 2022, I also worked, briefly, as a data scientist in an AdTech start-up.

In my mind, I pretend that these articles are written for those who have some quantitative training but is interested in what Machine Learning can offer, particularly graduate students who have taken at least one econometrics course. My emphases are those algorithms that can provide an alternative, sometimes more useful and rigorous, approach to known problems in economics, business, and social sciences. I also provide Python scripts that I have written to implement these algorithms.

Chapter 1

Randomized Controlled Trial, A/B/N Testing, and Multi-Armed Bandit Algorithms

1.1 Introduction

Randomized Controlled Trial (RCT) is the gold standard for establishing causality in experimental methods. It is used widely in clinical trials for new drugs or field experiments in social sciences and economics. In business, especially e-commerce, a related concept is A/B/N Testing. The main idea of RCT and A/B/N test is straightforward: individuals are randomly divided into groups to receive different treatments. Afterwards treatments, outcomes are being valuated and compared in order to find out which treatment works better/best. In RCT, a control group, where individuals receive a "placebo", is usually included. Note that placebo should be considered as a type of treatment too and individuals who receive a placebo are not getting "nothing". A placebo is something that has no therapeutic effect, i.e., it is not designed to cure a disease or an illness. But it can nevertheless positively impact the well-being of individuals who have received it, if due to nothing but psychological effects. As a result, it would be rather wrong to expect "no effect" from the the controlled group that receives the placebo in an RCT.

In the rest of this article, I will be using A/B/N test as the example because I want to stay away from the nitty-gritty details of RCT. I am using "A/B/N" to include tests with more than 2 versions. If you are only comparing two versions, it is an A/B test.

When I was interviewing for a data scientist job in 2022, the following was one of the interview questions: We are going to run an A/B test on a client's website. How long do we need to run the experiment for? Back then I knew about how to find minimum sample size based on hypothesis testing in Statistics, so I framed my answer that way. But I stopped in the middle while answering the question. Something I did not think seriously enough about popped into my head: how would I know the standard deviation, one of the required values to carry out the calculation for minimum sample size, before we even run the experiment? My interview went downhill from there. Needless to say, I did not get the job. However, the interviewer was nice enough to tell me that I should look into "power analysis".

I did. Suppose you have built an e-commerce website with two possible color pallettes, and you want to understand which color pallette would induce more purchases. You can randomly assign a visitor to the two versions of the website, and after a while, you will have a dataset with two columns: for each visitor, you recorded the version that the visitor was assigned to and the purchases that the visitor made. For $i \in (A, B)$, let's define the following values: * $\bar{x}_i$: expected dollars spent by visitors of version i; * n_i: number of visitors who saw version i; * s_i: standard deviation of dollars spent by visitors who saw version i.

We can now calculate the "power" as

$$t = \frac{\bar{x}_A - \bar{x}_B}{s_p \sqrt{\frac{1}{n_A} + \frac{1}{n_B}}}$$

where $s_p = \sqrt{\frac{(n_A-1)s_A^2 + (n_B-1)s_B^2}{n_A+n_B-2}}$ is the pooled standard deviation. The "power", t, follows a t-distribution with $n_A + n_B - 2$ degrees of freedom.

Suppose $s_A = s_B$ for the two versions in your A/B test, we can denote the two standard deviations as s. Also suppose, for simplicity, you want $n_A = n_B$. You can solve for n_i from the above power analysis formula and obtain:

$$N = \frac{4t^2 s^2}{(\bar{x}_A - \bar{x}_B)^2}$$

where N is the total sample size ($n_A + n_B$). It is easy to see that you will need a larger sample size if * the expected difference between the two versions are smaller; * you want a better significance level, e.g., 1% instead of 5%; * the standard deviation is bigger, i.e., dollars spent are more dispersed among individuals;

But here is the problem: you do not know the values of $\bar{x}_i$ and s_i before the experiment. For $\bar{x}_i$, it may be less of an issue. Instead of the expected values, all you really need is the *expected difference*, which can be specified. For example, suppose your website is currently running Version A, and all you care about is that Version B can increase expected purchase amount by 15. In other words, $\bar{x}_B - \bar{x}_A = 15$. But you still need to know the standard deviations. How? Some

suggest that you can run a short trial to estimate the standard deviation. But then, isn't the A/B test already a trial itself?

Here is another problem about classic A/B test design. After I became a data scientist, at another company, we actually ran an A/B test. The problem is that, according to the aforementioned power analysis, the experiment needed to be ran for at least 3 months, but we did not have that much time. After 1 month, our model (Version B) outperformed the existed model (Version A). Could we have declared our model to be the better one? According to classic A/B test design, the answer is "No" because we should not be "peeking" as the difference can be driven by random factors happened only in the first month of the experiment.

Now think about clinical trials for a new drug, where the "no peeking" rule can raise serious concerns. If a drug has proved its effectiveness in the first 500 patients, yet the power analysis tells you that you need to test it on 50,000 patients, what would you do? Isn't it unethical to continue to give a placebo to individuals who may benefit from the actual drug?

These two problems have bothered me for a while, until I learned about the approaches I will cover in this article. Here is a brief overview of how these algorithms work. Instead of having a predetermined sample size, the A/B test is deployed in real-time. Continued with our example of a website with two color pallettes, a visitor is randomly assigned to a version of the website on the first visit. An algorithm will then pick a version for a visitor. In general, the version that has received higher purchase values should get more visitors. But how do we know which one gives the higher payoff? Here, we face the Explore-Exploit Tradeoff.

1.2 The Explore-Exploit Tradeoff

In a nutshell, the explore-exploit tradeoff shows the following paradox: in order to find the best version, you need to explore, which means that the outcome of exploration necessarily improves the more you different versions. However, to maximize total payoff, you want to stick with the best version once (you think) you have found it, which is to exploit. This means that the outcome of exploitation necessarily deteriorates the more you different versions since there is one and only one best version.

How to handle the explore-exploit tradeoff constitutes the core difference among algorithms. Some algorithms, such as variants of the "greedy" family, really focuses on exploitation. The disadvantage is that such algorithm can easily "saddle" into the second-best as long as the second-best is "good enough", as I will show later when we discuss the `Epsilon Greedy` algorithm. Others, such as `Upper Confidence Bound`, put more emphasis on exploration, at least initially.

If you are reading this article because you think it may help you with your

research project(s), you are not a stranger to the explore-exploit tradeoff. I remember a conversation I had with a professor not long after I graduated. I asked him if I should have given up on projects that I do not think that would end up in good journals. His answer was: but how do you know before you try? He had a point: my professor never published his PhD dissertation. He was successful after he has explored a new area of research. However, about 15 years after his dissertation, a paper on a closely related topic was published in a top journal. In retrospect, he may have explored more than the optimum, which was probably why he suggested me to exploit more.

1.3 Epsilon Greedy

We will begin our in-depth discussion of algorithms with `Epsilon Greedy`. For each algorithm, I aim to provide: * intuition and description of theory * pseudocode * `Python` code

Algorithms in the `Greedy` family applies a simple logic: choose the version that gives the best *observed* expected payoff. For simplicity, and for the rest of this article, let's consider an e-commerce website that has 5 different designs but sells a single product: an EveryDay-Carry (EDC) musical instrument for 69.99 dollars. If we run an A/B/N test on the web designs, only 2 outcomes are possible from each visitor: buy or not buy.

Here is the pseudocode for a simple `Greedy` algorithm:

```
loop:
    j = argmax(expected bandit win rates)
    x = reward (1 or 0) from playing bandit j
    bandit[j].update_mean(x)
```

I used **bandit** instead of **version** here, and will be using these two terms interchangeably, because the problem we are working on is commonly known as the `Multi-Armed Bandits` problem in probability theory and reinforcement learning. The analogy comes from choosing among multiple slot machines in a casino since a single slot machine is referred to as a "one-armed bandit".

Let's take a closer look at the pseudocode. In this pseudocode, i indexes visitor, j indexes the website version (or bandit), and x is 1 when the visitor buys and 0 otherwise. Furthermore, `update_mean()` is a function that takes the new value of `x` and updates the expected payoff for bandit `j`. To update the expected payoff after bandit `j` was played for the n_{th} time, we have

$$\bar{x}_n = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

This calculates the mean in *constant time and memory*, i.e., it requires only 3 values to calculate the mean, $\bar{x}_n$, regardless of the value of n: $\bar{x}_{n-1}$, x_n, and n, whereas the number of values required to calculate the mean with the formula

$$\bar{x}_n = \frac{\sum_{i=1}^{n} x_i}{n}$$

increases with n.

While not necessary, it can sometimes be a good idea to try all versions in the beginning. For example, for the first 50 visitors, we can send them to each design with equal probability. From that point on, the algorithm finds the version that gives the best expected payoff, and play that version.

It should be obvious that the simple `Greedy` algorithm has a problem: once it finds a bandit with a *high enough* payoff, it rarely switches. In other words, it almost never explores. The `Epsilon Greedy` algorithm provides a simple fix:

```
loop:
    p = random number in [0, 1]
    if p < epsilon:
        j = choose a bandit at random
    else:
        j = argmax(expected bandit win rates)
    x = reward (1 or 0) from playing bandit j
    bandit[j].update_mean(x)
```

As the pseudocode shows, a random value is drawn when a new visitor has arrived. If the random value is smaller than the threshold, `epsilon`, set before the start of the experiment, then a random bandit is picked. Note that this randomly picked bandit can be the same as the one picked by `argmax`. To exclude such case only requires a few more lines of code. However, this is not a requirement of the `Epsilon Greedy` algorithm and the benefit of doing so is not obvious.

Let's now move onto the actual implementation of `Epsilon Greedy` in `Python`. Note that the script includes lines with the comment "*only in demonstration*". These are codes to generate the *true* win rate of different bandits, which you do not know when running a real-world experiment. This also means you can not use the scripts here in real-world problems without making necessary changes.

```
import numpy as np
import random

# set the number of bandits
N_bandits = 5
# set the number of trials
# only in demonstration
N = 100000


class BayesianAB:
```

```
    def __init__(
            self,
            number_of_bandits: int = 2,
            number_of_trials: int = 100000,
            p_max: float = .75,
            p_diff: float = .05,
            p_min: float = .1
    ):
        if p_min > p_max - p_diff:
            raise ValueError("Condition p_min < p_max - p_diff not satisfied. Exit...")

        self.prob_true = [0] * number_of_bandits  # only in demonstration
        self.prob_win = [0] * number_of_bandits
        self.history = []
        self.history_bandit = []  # for Monte Carlo
        self.count = [0] * number_of_bandits  # only in demonstration
        # preference and pi are for gradient_bandit only
        self.pref = [0] * number_of_bandits
        self.pi = [1 / number_of_bandits] * number_of_bandits
        # a and b are for bayesian_bandits only
        self.alpha = [1] * number_of_bandits
        self.beta = [1] * number_of_bandits
        # number of trials/visitors
        self.N = number_of_trials

        # set the last bandit to have a win rate of 0.75 and the rest lower
        # only in demonstration
        self.prob_true[-1] = p_max
        for i in range(0, number_of_bandits - 1):
            self.prob_true[i] = round(p_max - random.uniform(p_diff, p_max - p_min), 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,
    ) -> None:
```

```
        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.history_bandit.append(i)  # for Monte Carlo
        self.count[i] += 1

    ####################
    # epsilon greedy
    def epsilon_greedy(
            self,
            epsilon: float = 0.5,
    ):

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

        for k in range(1, self.N):
            if random.random() < epsilon:
                i = random.randrange(0, len(self.prob_win))
            else:
                i = np.argmax(self.prob_win)

            self.update(i, k)

        return self.history
```

Let's break it down. First, we import two libraries: `numpy` and `random`. We will be using functions from these libraries such as `argmax()` from `numpy` and `randrange()` from `random`:

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

We then set three global parameters:

```
# set the number of bandits
N_bandits = 5
# set the number of trials/visitors
N = 100000
```

In practice, the value of `N_bandits` depends on the number of versions your experiment is set out to test, and the number of visitors, `N`, is unknown.

In this script, we are creating a class named `BayesianAB`, and put all the algorithms we cover in this article under `BayesianAB`. We initiate the class with the following values:

```
class BayesianAB:
    def __init__(
            self,
            number_of_bandits: int = 2,
            number_of_trials: int = 100000,
            p_max: float = .75,
            p_diff: float = .05,
            p_min: float = .8
    ):
        if p_min > p_max - p_diff:
            raise ValueError("Condition p_min < p_max - p_diff not satisfied. Exit..."

        self.prob_true = [0] * number_of_bandits  # only in demonstration
        self.prob_win = [0] * number_of_bandits
        self.history = []
        self.history_bandit = []  # for Monte Carlo
        self.count = [0] * number_of_bandits  # only in demonstration
        # preference and pi are for gradient_bandit only
        self.pref = [0] * number_of_bandits
        self.pi = [1 / number_of_bandits] * number_of_bandits
        # a and b are for bayesian_bandits only
        self.alpha = [1] * number_of_bandits
        self.beta = [1] * number_of_bandits
        # number of trials/visitors
        self.N = number_of_trials
```

The `BayesianAB` class accepts 5 parameters: * `number_of_bandits` has a default value of 2; * `number_of_trials` indicates the number of rounds/visitors, which has a default value of 100,000; * `p_max` is the highest win rate; * `p_diff` is the smallest possible difference between the highest win rate and the second highest win rate; * `p_min` is the lowest possible win rate, andt the condition `p_min > p_max - p_diff` must be met.

The `BayesianAB` class pre-allocates 10 lists to store various values necessary for different tasks: * `prob_true` stores the *true* win rate of each bandit. These win rates are to be generated next. In practice, you do not know these true win rate values; * `prob_win` stores the *observed (expected)* win rate of each bandit. Values in this list are to be updated during each round of the experiment; * `history` stores the history of `prob_win` in each trial/round. This is important for both updating the mean in constant time (see above) and the evaluation of bandit/algorithm performances afterwards; * `history_bandit` stores the history of what bandit was picked in each trial/round. This is useful when we need to run the Monte Carlo simulation for testbed; * `count` stores the number of times that each bandit was chosen; * `pref` and `pi` are values for the `Gradient Bandit` algorithm; * `alpha` and `beta` are values used in `Thompson Sampling`, the last algorithm to be considered in this article; * `N` stores the number of trials

and is used by each method/algorithm in the `BayesianAB` class.

The following lines generate the *true* win rates:

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

The last bandit always has the highest win rate, `p_max`, and the rest of them are randomized between `p_min` and `p_max - p_diff`. I used this approach to allow for flexibility in specifying the number of bandits using `N_bandits` (or `number_of_bandits` inside the `BayesianAB` class).

Next, we define two functions used by almost every algorithm:

```
    # 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,
    ) -> None:
        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.history_bandit.append(i)  # for Monte Carlo
        self.count[i] += 1
```

The first function, `pull()`, returns either True or False depending on if the value of `random.random()` is less than the true win rate of bandit i. This is unnecessary in practice. Instead, a call to either the `BayesianAB` class or a specific method (such as `Epsilon Greedy`) inside `BayesianAB` should be triggered with the arrival of a new visitor, and by the end of the visit, you would know if the visitor has purchased (True) or not (False). In `Python`, `True` is given a numerical value of 1 and `False` a value of 0.

The `update()` function updates the mean. It also adds the updated expected win rate to the list `history` and increase the count of bandit i being picked by 1.

Here is the actual method inside `BayesianAB` that implements `epsilon greedy`:

```
    def epsilon_greedy(
            self,
            epsilon: float = 0.5,
    ):

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

        for k in range(1, self.N):
            if random.random() < epsilon:
                i = random.randrange(0, len(self.prob_win))
            else:
                i = np.argmax(self.prob_win)

            self.update(i, k)

        return self.history
```

It follows the logic outlined in the pseudocode. Inside the `for` loop, these steps are followed: 1. Checks if a random value is smaller than `epsilon` which can be specified when the `epsilon_greedy()` method is called. `epsilon` also has a default value of 0.5. If this is `True`, then a random bandit is selected; 2. Otherwise, select the bandit that has the highest expected win rate; 4. Update the mean for the chosen bandit by calling the `update()` function.

The `epsilon_greedy()` method returns the list `history`, which stores the complete history for run as discussed earlier.

To call `epsilon_greedy()` and examine the results, we execute the following:

```
eg = BayesianAB(N_bandits)
print(f'The true win rates: {eg.prob_true}')
eg_history = eg.epsilon_greedy()
print(f'The observed win rates: {eg.prob_win}')
print(f'Number of times each bandit was played: {eg.count}')
```

Here, we call `epsilon_greedy()` with the default value for `epsilon`. This means the algorithm will explore half of the time. We also print out the true win rates, the expected win rates, and the number of times that each bandit was played. Here is the printed output from a particular run:

```
The true win rates: [0.37, 0.55, 0.67, 0.4, 0.75]
The observed win rates: [0.2062, 0.3354, 0.6717, 0.1953, 0.5526]
Number of times each bandit was played: [10200, 9945, 60001, 9789, 10064]
```

In the above run, the best bandit was NOT the one that was selected the most. The second best bandit, with a 0.67 win rate, was picked about 60% of the time, as dictated by the value of `epsilon`. Such outcome is due to the fact that the

bandit with a 0.67 win rate did well in the beginning. Since it is close enough to 0.75, the default win rate of the best bandit, random jumps to the bandit with the 0.75 win rate were not enough to "flip" the results.

Also note that the expected win rates have not converged to the true win rates except for the "chosen" one after 100,000 visitors. However, if the number of visitors approaches infinity, which means that each version would be picked infinite times, all win rates would converge to their true values. This, in turn, means that the best bandit would eventually overtake the second-best if the experiment runs *long enough*. In other words, `Epsilon Greedy` guarantees the identification of the best bandit as n approaches infinity.

We can visualize the outcome from the experiment with the following code:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns


def plot_history(
        history: list,
        prob_true: list,
        k=N,
):
    df_history = pd.DataFrame(history[:k])

    plt.figure(figsize=(20, 5))

    # Define the color palette
    colors = sns.color_palette("Set2", len(prob_true))

    for i in range(len(prob_true)):
        sns.lineplot(x=df_history.index, y=df_history[i], color=colors[i])

    # Create custom legend using prob_true and colors
    custom_legend = [plt.Line2D([], [], color=colors[i], label=prob_true[i]) for i in range(len(p
    plt.legend(handles=custom_legend)
```

Then execute:

```
plot_history(history=eg.history, prob_true=eg.prob_true)
```

Here is the output from the above run:

We can also get the visualization for the first 100 visitors, which shows that the third bandit, the a 0.67 win rate, jumped ahead early:

```
plot_history(history=eg.history, prob_true=eg.prob_true, k=100)
```

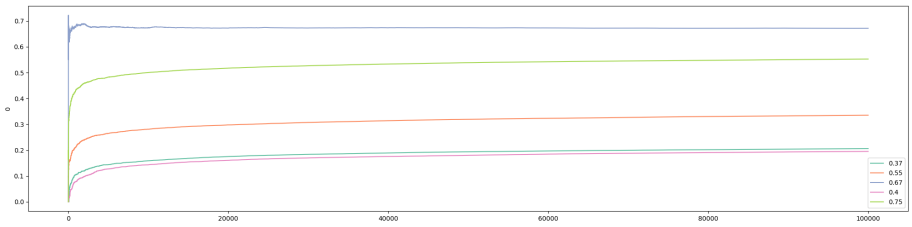

Figure 1.1: Epsilon Greedy

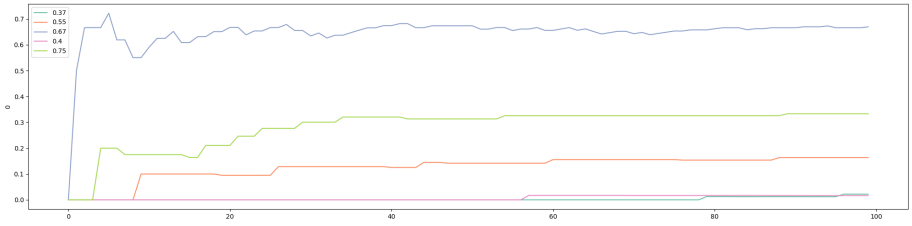

Figure 1.2: Epsilon Greedy (first 100)

1.4 Optimistic Initial Values

The `Optimistic Initial Values` algorithm is one of my favorites (the other being the `Gradient Bandit` algorithm) amongst the algorithms discussed in this article. While `Epsilon Greedy` focused on "exploit" and can end up choosing the second-best version, the `Optimistic Initial Values` algorithm puts more focus on "explore" initially, while staying `greedy`, i.e., pick the strategy that shows the highest expected win rate. The name of this algorithm informs you about what it does: at the start of the experiment, each bandit is set to have a high expected win rate, i.e., we are "optimistic" about each bandit. This ensures that each of them is played a fair number of times initially. If we compare `Epsilon Greedy` to English auctions where the values go up over time, `Optimistic Initial Value` is comparable to Dutch auctions where the values go *down* over time. Here is the pseudocode:

```
p_init = 5 # a large value as initial win rate for ALL bandits

loop:
    j = argmax(expected bandit win rates)
    x = reward (1 or 0) from playing bandit j
    bandit[j].update_mane(x)
```

Assuming you already have the code from the `Epsilon Greedy` section, you can add the following method inside the `BayesianAB` class to run the `Optimistic Initial Values` algorithm:

```
    ####################
    # optimistic initial values
    def optim_init_val(
            self,
            init_val: float = 0.99,
    ):

        self.prob_win = [init_val] * len(self.prob_win)
        self.history.append(self.prob_win.copy())

        for k in range(1, self.N):
            i = np.argmax(self.prob_win)

            self.update(i, k)

        return self.history
```

The only thing new here is the line that assigns `init_val` to `prob_win` in the beginning. We can execute the following to get results and visualization:

```
oiv = BayesianAB(N_bandits)
print(f'The true win rates: {oiv.prob_true}')
oiv_history = oiv.optim_init_val(init_val=0.99)
print(f'The observed win rates: {oiv.prob_win}')
print(f'Number of times each bandit was played: {oiv.count}')

# plot the entire experiment history
plot_history(history=oiv.history, prob_true=oiv.prob_true)
```

And following are outcomes from a typical run:

```
The true win rates: [0.6, 0.54, 0.62, 0.14, 0.75]
The observed win rates: [0.6633, 0.7493, 0.7491, 0.7493, 0.7521]
Number of times each bandit was played: [2, 168, 285, 65, 99479]
```

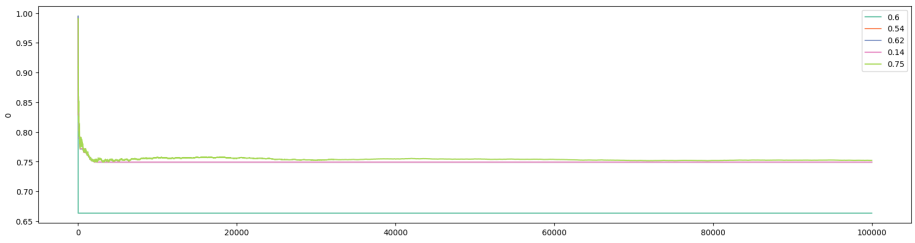

Figure 1.3: Optimistic Initial Values

From the visualization below, you can see that the best bandit jumped ahead

after about merely 15 visitors:

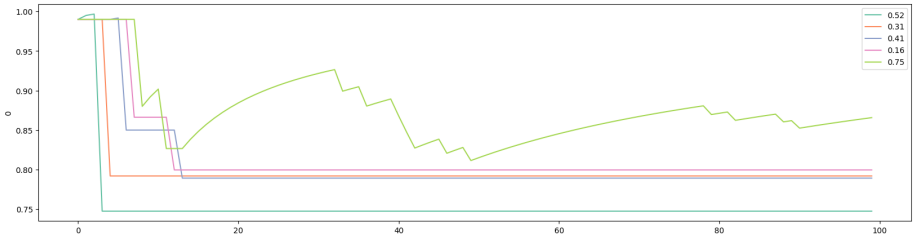

Figure 1.4: Optimistic Initial Values (first 100)

Note that I set `init_val` to 0.99 since we are comparing win rates that can not exceed 1. The larger the initial value, the more the algorithm explores initially. Because the `Optimistic Initial Values` algorithm was specifically designed to explore in the beginning, it can "fall behind" in reaching the best version, if ever, compared to other algorithms such as `Epsilon Greedy`. Note that if the best bandit is discovered early, the expected win rates of other bandits never converge to their true win rates in `Optimistic Initial Values` (but would in `Epsilon Greedy`). This is a common feature of several of the algorithms discussed in this article.

1.5 Upper Confidence Bound (UCB)

The theory of `UCB` is harder to fully grasp although its intuition and implementation are straightforward. To start, let's look back to the last two algorithms that we have discussed: `Epsilon Greedy` and `Optimistic Initial Values`. A common step in the implementation of both of these algorithms is to find the bandit that gives the best *observed* expected win rate. This is why both algorithms are said to be `greedy`. But can we do better, especially that we know these expected win rates are probabilistic? Put differently, we know that the more a certain bandit was selected, the closer its expected win rate is to its true win rate. But what about those bandits that were rarely picked?

That is where `Upper Confidence Bound` comes into play. The idea is that we should not be relying on the observed expected win rates alone in making decisions. We should give each version some "bonus points": if a version has been picked a lot, it gets a small bonus; but if a version has barely been barely chosen, it should get a larger bonus because, probabilistically, the observed expected win rate *can* be far from the true win rate if a version has not been played much.

If you are interested in the math, you can read the paper "Finite-time Analysis of the Multiarmed Bandit Problem". In the paper, the authors have outlined a function for the "bonus", which is commonly known as `UCB1`:

$$b_j = \sqrt{\frac{2 \log N}{n_j}}$$

where N is the total number of visitors at the time of computing the bonus, and n_j is the number of times that bandit j was chosen at the time of computing the bonus. Adding b to the expected win rate gives the **upper confidence bound**:

$$\text{UCB1}_j = \bar{x}_{n_j} + b_j$$

Here is the pseudocode for `UCB1`:

```
loop:
    Update UCB1 values
    j = argmax(UCB1 values)
    x = reward (1 or 0) from playing bandit j
    bandit[j].update_mean(x)
```

Adding the following method into `BayesianAB` will implement `UCB1`:

```
    ####################
    # upper confidence bound (UCB1)
    def ucb1(
            self,
            c=1,
    ):

        self.history.append(self.prob_win.copy())
        bandit_count = [0.0001] * len(self.prob_win)

        for k in range(1, self.N):
            bound = self.prob_win + c * np.sqrt(np.divide(2 * np.log(k), bandit_count))
            i = np.argmax(bound)

            self.update(i, k)

            if bandit_count[i] < 1:
                bandit_count[i] = 0
            bandit_count[i] += 1

        return self.history
```

This is very similar to what we had before. One thing to note is that I give a very small initial value (0.0001) to `bandit_count` to avoid the division of zero in the beginning of the experiment. Later, I reversed the value to 0 with the `if` statement. An alternative approach is to run the first several iterations on all versions before implementing `UCB1` thereafter.

`UCB1` has a parameter, c, which controls the degree of exploration. Other things being equal, A larger value c means a higher reward. The default value is set to 1 in the above script.

Executing the following will give us results and visualization for `UCB1`:

```
ucb = BayesianAB(N_bandits)
print(f'The true win rates: {ucb.prob_true}')
ucb_history = ucb.ucb1()
print(f'The observed win rates: {ucb.prob_win}')
print(f'Number of times each bandit was played: {ucb.count}')

# plot the entire experiment history
plot_history(history=ucb.history, prob_true=ucb.prob_true)
```

A typical run gives the following:

```
The true win rates: [0.65, 0.12, 0.63, 0.33, 0.75]
The observed win rates: [0.6505, 0.1165, 0.1928, 0.0774, 0.3794]
Number of times each bandit was played: [99470, 77, 103, 67, 282]
```

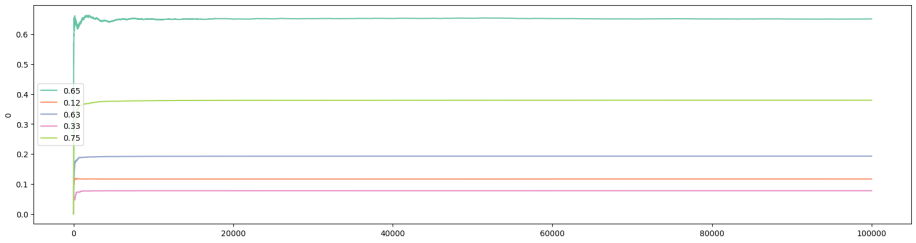

Figure 1.5: UCB1

This particular run shows that `UCB1` has also failed to identify the best version. Examining the first 100 visitors shows that the bandit with a .65 win rate jumped out early and never looked back. And I do not believe the algorithm can guarantee convergence even with infinite number of visitors:

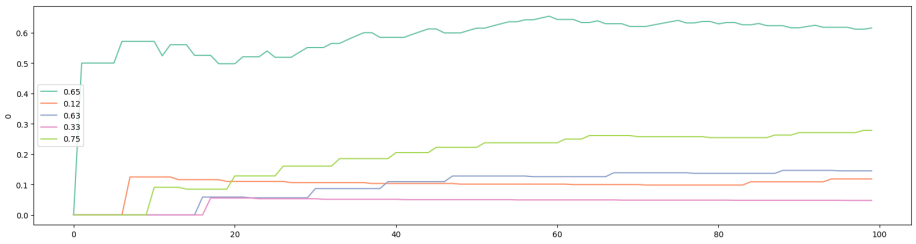

Figure 1.6: UCB1 (first 100)

1.6 Gradient Bandit Algorithm

Another algorithm that does not rely *entirely* on expected win rates is the `Gradient Bandit` algorithm. Not surprisingly, this algorithm is related to gradient ascent. With this algorithm, each bandit's probability of being chosen is determined according to a soft-max distribution:

$$\pi_n(i) = \frac{e^{H_n(i)}}{\sum_{j=1}^{J} e^{H_n(j)}}$$

where $\pi_n(i)$ is the probability of bandit i being picked for customer n, $H_n(i)$ is the *preference* for bandit i at the time customer n arrives, and J is the total number of bandits in the experiment. In the case of only two bandits, this specification is the same as the logistic or sigmoid function.

When the first customer arrives, i.e., $n = 1$, it is custom to set the *preference*, $H_1(j)$, for all j, to 0 so that every bandit has the same probability of getting picked. Suppose bandit i is picked for customer $n(\geq 1)$, then the *preference* for i is updated according to:

$$H_{n+1}(i) = H_n(i) + \alpha(x_n - \bar{x}_{n-1})(1 - \pi_n(i))$$

whereas the *preferences* for all $j \neq i$ are updated according to:

$$H_{n+1}(j) = H_n(j) - \alpha(x_n - \bar{x}_{n-1})\pi_n(j)$$

where $\alpha > 0$ is a "step-size" parameter.

The intuition of the `Gradient Bandit` algorithm is as follows. When the reward received from picking i for customer n is higher than the expected reward, the probability of picking i in the future (next round) is increased. In our simple case with only two outcomes (buy and not buy), the reward is higher than the expected reward only if customer n buys.

Let's take a look at the pseudocode:

```
H = [0] * number_of_bandits

loop:
  pi = pi(H) # Calculates the soft-max distribution
  i = random.choices(bandits, weights=pi)
  gb.update()
```

where `H.update()` updates the values of $H(i)$ (the bandit that was chosen) and $H(j)$ (bandits that were not chosen).

Here is the `Python` implementation for `Gradient Bandit`:

```
    ####################
    # gradient_bandit update
    def gb_update(
            self,
            i,
            k,
            a,
    ):

        outcome = self.pull(i)
        for z in range(len(self.pref)):
            if z == i:
                self.pref[z] = self.pref[z] + a * (outcome - self.prob_win[z]) * (1 - s
            else:
                self.pref[z] = self.pref[z] - a * (outcome - self.prob_win[z]) * self.p

        self.prob_win[i] = (self.prob_win[i] * k + outcome) / (k + 1)

        return self.pref

    # gradient bandit algorithm
    def gradient_bandit(
            self,
            a=0.2,
    ):

        self.history.append([self.pi.copy(),
                             self.pref.copy(),
                             self.prob_win.copy()])

        for k in range(1, self.N):
            self.pi = np.exp(self.pref) / sum(np.exp(self.pref))
            pick = random.choices(np.arange(len(self.pref)), weights=self.pi)
            i = pick[0]
            self.pref = self.gb_update(i, k, a)

            self.count[i] += 1
            self.history.append([self.pi.copy(),
                                 self.pref.copy(),
                                 self.prob_win.copy()])
            self.history_bandit.append(i)  # for Monte Carlo

        return self.history
```

Here are some notes on the `Python` implementation of the `Gradient Bandit`

algorithm: 1. We have already initiated `pref` and `pi` at the start of the `BayesianAB` class; 2. As discussed in the pseudocode, a new function, `gb_update()`, is necessary since we need to update the preference function for every bandit in each round; 3. The `gradient_bandit()` function takes 1 parameter: a, which is the step-size parameter. The default value of a is set to 0.2. The smaller the value of a, the more the algorithm explores; 4. `gradient_bandit()` saves `history` differently: each row in `history` is an array of 3 lists. In order to examine the performance of the `Gradient Bandit` algorithm, we not only save the expected win rates, but also preferences and the values from the soft-max distribution, $\pi(i)$; 5. The function `choices()` from the `random` library picks a value from a list based on `weights`. In this case, the weights is given by the soft-max distribution;

Because `gradient_bandit()` saves arrays in `history`, we also need to update the `plot_history()` function:

```
def plot_history(
        history: list,
        prob_true: list,
        col=2,
        k=N,
):
    if type(history[0][0]) == list:  # To accommodate gradient bandit
        df_history = pd.DataFrame([arr[col] for arr in history][:k])
    else:
        df_history = pd.DataFrame(history[:k])

    plt.figure(figsize=(20, 5))

    # Define the color palette
    colors = sns.color_palette("Set2", len(prob_true))

    for i in range(len(prob_true)):
        sns.lineplot(x=df_history.index, y=df_history[i], color=colors[i])

    # Create custom legend using prob_true and colors
    custom_legend = [plt.Line2D([], [], color=colors[i], label=prob_true[i]) for i in range(len(p
    plt.legend(handles=custom_legend)
```

The updates occurred in

```
    if type(history[0][0]) == list:  # To accommodate gradient bandit
        df_history = pd.DataFrame([arr[col] for arr in history][:k])
    else:
        df_history = pd.DataFrame(history[:k])
```

with the added parameter `col`. This is to accommodate the arrays saved in his-

tory by `gradient_bandit()`. The `if` statement checks whether the first element in `history` is a `list`. If it is, then `history` was saved from `gradient_bandit()` and we would need to extract the specific column, given by `col`, for plotting. The default value of `col` is 2, which is to plot the history of the win rates.

Executing the following will run the `Gradient Bandit` algorithm:

```
# Gradient bandit
gb = BayesianAB(N_bandits)
print(f'The true win rates: {gb.prob_true}')
gb_history = gb.gradient_bandit()
print(f'The observed win rates: {gb.prob_win}')
print(f'Number of times each bandit was played: {gb.count}')

# plot the entire experiment history
plot_history(history=gb.history, prob_true=gb.prob_true)
```

Here are results from a typical run:

```
The true win rates: [0.17, 0.56, 0.17, 0.7, 0.75]
The observed win rates: [0.2564, 0.5514, 0.0105, 0.6636, 0.7498]
Number of times each bandit was played: [35, 67, 22, 196, 99679]
```

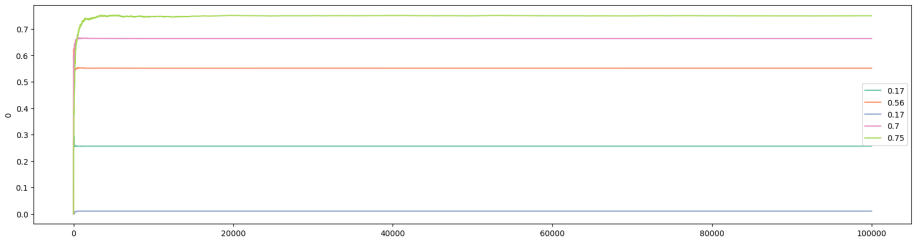

Figure 1.7: Gradient Bandit

As usual, we can examine what happened after 100 customers. Interestingly, the bandit with the highest win rate did not lead after only 100 customers:

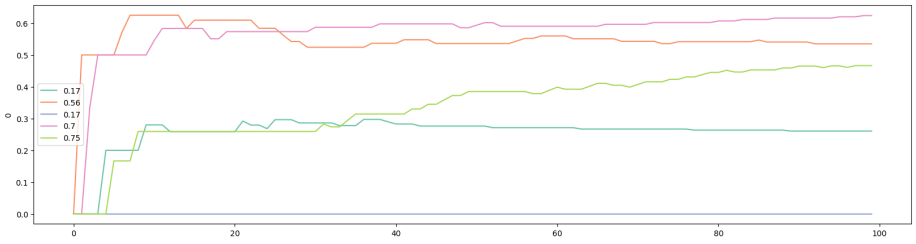

Figure 1.8: Gradient Bandit (first 100)

We can plot the evolution of the `preference` with the following:

```
# plot preference
plot_history(history=gb.history, prob_true=gb.prob_true, col=1)
```

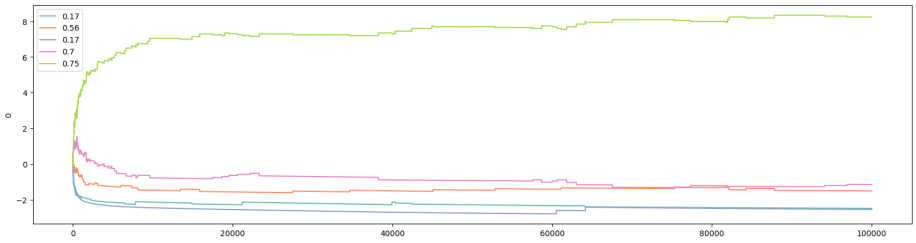

Figure 1.9: Gradient Bandit (preference)

And plot the soft-max function with the following:

```
# plot pi
plot_history(history=gb.history, prob_true=gb.prob_true, col=0)
```

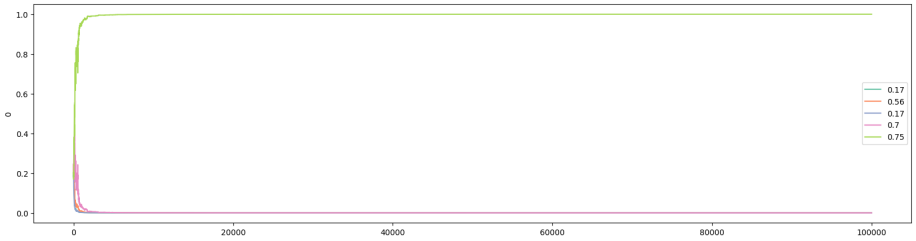

Figure 1.10: Gradient Bandit (pi)

There are several reasons why the `Gradient Bandit` algorithm is one of my favorites: 1. Economists are familiar with the idea of using `preference` to model choices; 2. Economists are familiar with `logistic` function, used in logistic regression, which is the special case of `soft-max` with only two bandits; 3. One of my research areas is conflict and contest, in which the `soft-max` function, known as "contest success function", is widely used in the literature.

1.7 Thompson Sampling (Bayesian Bandits)

`Thompson Sampling`, or `Bayesian Bandits`, takes another (big) step forward. In our discussion on `Upper Confidence Bound`, we acknowledged the fact that using only the expected win rate to represent the performance of a bandit is not accurate. To tackle this, `UCB1` adds a "bonus": the bonus is smaller for the bandits that were played more, and larger for the bandits that were played less. Then in our discussion on `Gradient Bandit`, each bandit's chance of being picked is described by a soft-max distribution.

To push these ideas further, and as the name `Thompson Sampling` has hinted, we ask if we could construct a probability distribution to describe the expected win rates of all the bandits. As it turns out, this is possible, as everything, including parameters, are considered random variables in Bayesian Statistics. For example, with Normal Distribution, we often speak about fixed values of mean and variance. But in Bayesian Statistics, the mean and variance of a Normal Distribution are two random variables and they can be described by probability distributions.

The mathematical derivation of `Thompson Sampling` requires the use of conjugate prior, which I will discuss here briefly, before returning to the `Python` implementation of `Thompson Sampling`.

1.8 Conjugate Prior

Recall the Bayes Rule:

$$p(\theta \mid X) = \frac{p(X \mid \theta)p(\theta)}{p(X)}$$

where the four parts are called, respectively * $p(\theta \mid X)$: Posterior distribution * $p(X \mid \theta)$: Likelihood function * $p(\theta)$: Prior probability distribution * $p(X)$: Evidence

In Bayesian Statistics, if the posterior distribution is in the same probability distribution family as the prior probability distribution, the prior and posterior are then called conjugate distributions, and the prior is called a **conjugate prior** for the likelihood function.

With conjugate priors, the updating in a Bayesian approach reduces to the updates of hyperparameters that are used to describe both the prior and posterior distributions, since they are the same. I will leave the details to a Statistics textbook, but for our purpose, since our example concerns of binary outcomes (buy or not buy), our likelihood function is that of Bernoulli. As it turns out, the conjugate prior for a Bernoulli likelihood function is the Beta distribution, which has two hyperparameters: α and β.

Now that we have established that Beta distribution is the conjugate prior for Bernoulli, the problem of `Thompson Sampling` is reduced to 1. sample from the Beta distribution 2. find the highest expected value

1.9 Thompson Sampling: Code

Since `Thompson Sampling` is mechanical different from the previous algorithms, we need to develop special functions and methods to implement `Thompson Sampling`. Here is a pseudocode:

```
loop:
    sampling from Beta function for bandit b
```

```
j = argmax(b.sample() for b bandits)
x = reward (1 or 0) from playing bandit j
bandit[j].bb_update(x)
```

The function that needs to be added is `bb_update()`. Here is the full `Python` implementation:

```
from scipy.stats import beta

    ####################
    # bayesian_bandits update
    def bb_update(
            self,
            a,
            b,
            i,
    ):

        outcome = self.pull(i)
        a[i] += outcome
        b[i] += 1 - outcome
        self.count[i] += 1

        return a, b

    # Bayesian bandits
    # For Bernoulli distribution, the conjugate prior is Beta distribution
    def bayesian_bandits(
            self,
            sample_size: int = 10,
    ):

        a_hist, b_hist = [], []
        a_hist.append(self.alpha.copy())
        b_hist.append(self.beta.copy())

        for k in range(1, self.N):
            sample_max = []

            for m in range(len(self.prob_true)):
                m_max = np.max(np.random.beta(self.alpha[m], self.beta[m], sample_size))
                sample_max.append(m_max.copy())

            i = np.argmax(sample_max)

            self.alpha, self.beta = self.bb_update(self.alpha, self.beta, i)
```

```
            a_hist.append(self.alpha.copy())
            b_hist.append(self.beta.copy())
            self.history_bandit.append(i)  # for Monte Carlo

        self.history = [a_hist, b_hist]
        return self.history
```

Let's walk through this script: 1. We have already initiated `alpha` and `beta` at the start of the `BayesianAB` class. They are the hyperparameters in the Beta distribution; 2. We import `beta` from `scipy.stats` since the conjugate prior for Bernoulli distribution is the Beta distribution. 3. The function `bb_update()` updates the hyperparameter values based on the outcome from the last visitor for bandit i: if the outcome was `True`, then the value of `alpha` increases by 1; otherwise, the value of `beta` increases by 1.

For the actual implementation of the `Bayesian Bandits` in the `bayesian_bandits()` method, it is largely consistent with what we have been doing in other algorithms. The main differences include: 1. Instead of storing the history of outcomes, we store the history of the values of `alpha` and `beta`; 2. In each iteration, we first find the maximum value from the sample of values of each bandit, then pick the best from this set of *maximum* values; 3. As described earlier, the updating is different. Instead of updating the mean, we update the values of `alpha` and `beta`.

Due to these changes, we also need a new function for visualizing the `history` returned by `bayesian_bandits()`:

```
def bb_plot_history(
        history: list,
        prob_true: list,
        k=-1,
):
    x = np.linspace(0, 1, 100)
    legend_str = [[]] * len(prob_true)
    plt.figure(figsize=(20, 5))

    for i in range(len(prob_true)):
        a = history[0][k][i]
        b = history[1][k][i]
        y = beta.pdf(x, a, b)
        legend_str[i] = f'{prob_true[i]}, alpha: {a}, beta: {b}'
        plt.plot(x, y)

    plt.legend(legend_str)
```

We can now run a simulate for `Thompson Sampling` by executing the following:

```
bb = BayesianAB(N_bandits)
print(f'The true win rates: {bb.prob_true}')
bb_history = bb.bayesian_bandits(sample_size=10)
print(f'The observed win rates: {np.divide(bb.history[0][-1], bb.count)}')
print(f'Number of times each bandit was played: {bb.count}')

# plot the entire experiment history
bb_plot_history(history=bb.history, prob_true=bb.prob_true)
```

Outcomes from a typical run look like the following:

```
The true win rates: [0.15, 0.67, 0.11, 0.47, 0.75]
The observed win rates: [0.1, 0.6563, 0.1667, 0.4545, 0.7500]
Number of times each bandit was played: [10, 355, 12, 44, 99578]
```

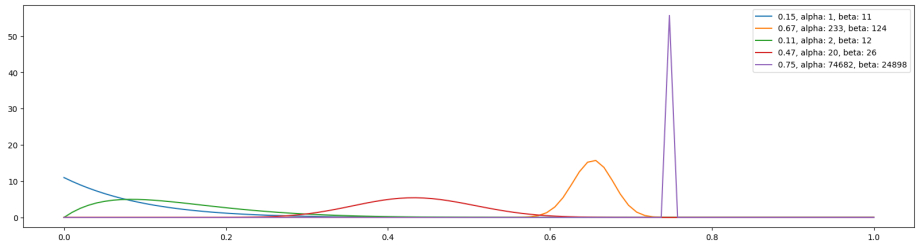

Figure 1.11: Bayesian Bandits

It is also interesting to look at what happened after only 100 visitors:

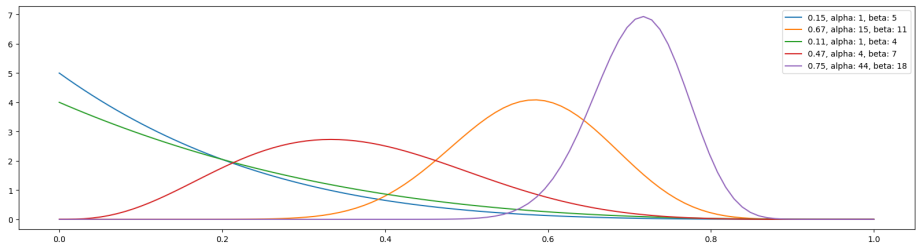

Figure 1.12: Bayesian Bandits (first 100)

Two differences between `Thompson Sampling` and the other algorithms we have discussed should be noted. First, as already mentioned, `Thompson Sampling` attempts to build a distribution for the bandits. Comparing the two visuals from 100 visitors and all visitors shows that, although the best version has jumped out early, the distribution is much tighter/narrower at the end of the experiment, indicating greater "confidence" for the estimated expected win rate. Second, and importantly, the `Thompson Sampling` algorithm has no problem

distinguishing between a bandit with a 0.67 win rate and the best version with a win rate of 0.75.

1.10 Comparing the Algorithms

It is important to compare the five algorithms in various settings. Following Sutton and Barto (2020), I conduct a 5-armed testbed. The idea of the testbed is to run the algorithms many times, say 2,000, then calculates the success rate, which is the percentage that the best bandit was picked in each round. For example, suppose we run the `Epsilon Greedy` algorithm 2,000 times with different win rates. We look at the bandit picked on the 100th visitor and found that, out of the 2,000 runs the best bandit was picked 800 times. Then at the 100th round/visitor, the success rate was 0.4. When we developed the `BayesianAB` class, we were already anticipating the implementation of the testbed. Specifically, in the following functions/methods: * `update()` * `gradient_bandit()` * `bayesian_bandits()`

there is

```
self.history_bandit.append(i)
```

which records the which bandit was picked at each round. The parameters `p_max`, `p_diff`, and `p_min` also allow the `BayesianAB` class to generate different win rates. We will now develop a script to implement the testbed where the `BayesianAB` class is imported and called:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from multiprocessing import Pool, cpu_count
from functools import partial

from bayesianab import BayesianAB

# set the number of bandits
N_bandits = 5
# set the number of visitors
N = 10001
# set the number of trials
M = 2000


def worker(algo, number_of_bandits, number_of_trials, p_max, p_diff, p_min, n):
    bayesianab_instance = BayesianAB(number_of_bandits, number_of_trials, p_max, p_dif
    getattr(bayesianab_instance, algo)()
    return bayesianab_instance.history_bandit
```

```
def monte_carlo(
        algos,
        m=500,
        n=10001,
        p_max: float = .75,
        p_diff: float = .05,
        p_min: float = .1
):
    algos_hist = {algo: [] for algo in algos}

    for algo in algos:
        print(f'Running {algo}...')
        with Pool(cpu_count()) as pool:
            func = partial(worker, algo, N_bandits, n, p_max, p_diff, p_min)
            results = list(pool.imap(func, range(m)))

        algos_hist[algo] = results

    return algos_hist


def run_monte_carlo(
        algos,
        m,
        n,
        p_values,
):
    trials = {}
    df_all = {}

    for i in range(len(p_values)):
        print(f'The p_values are {p_values[i]}')
        trials[f'p{i}'] = monte_carlo(algos,
                                      m,
                                      n,
                                      p_values[i][0],
                                      p_values[i][1],
                                      p_values[i][2],)

    for i in range(len(p_values)):
        df = pd.DataFrame()
        for j in algos:
            lst = [0] * (N - 1)
            for k in range(M):
```

```
                lst = np.array(lst) + np.array([1 if x == 4 else 0 for x in trials[f'p
            df[j] = (lst / M).tolist()

        df_all[f'p{i}'] = df.copy()

    return df_all


def plot_monte_carlo(
        df_all,
        algos,
        col,
        row,
):
    figure, axis = plt.subplots(row, col, figsize=(20, 10))
    colors = sns.color_palette("Set2", len(algos))

    m = 0  # column index
    n = 0  # row index

    for key in df_all:
        ax = axis[n, m]
        for i in range(len(algos)):
            sns.lineplot(x=df_all[key].index, y=df_all[key][algos[i]], linewidth=0.5,

        ax.set_ylabel('')
        ax.set_title(prob_list[n * 3 + m])
        ax.set_xticks([])

        if m == 2:
            # Create custom legend using prob_true and colors
            custom_legend = [plt.Line2D([], [], color=colors[i], label=algos[i]) for i
            ax.legend(handles=custom_legend, loc='upper left', fontsize=9)
            n += 1
            m = 0
        else:
            m += 1

    figure.suptitle('Comparing 5 Algorithms in 12 Different Win Rate Specifications',

    # Adjust the spacing between subplots
    plt.tight_layout()

    plt.savefig("comparison.png", dpi=300)
    # plt.show()
```

```
if __name__ == "__main__":
    algorithms = ['epsilon_greedy', 'optim_init_val', 'ucb1', 'gradient_bandit', 'bayesian_bandit
    prob_list = [[.35, .1, .1], [.35, .05, .1], [.35, .01, .1],
                 [.75, .1, .1], [.75, .05, .1], [.75, .01, .1],
                 [.75, .1, .62], [.75, .05, .62], [.75, .01, .62],
                 [.95, .1, .82], [.95, .05, .82], [.95, .01, .82],
                 ]

    results_df = run_monte_carlo(algorithms, M, N, prob_list)

    plot_monte_carlo(results_df, algorithms, 3, 4)
```

Some explanations may be instructive. First, since we will be calling the same functions many times, I decided to use parallelization, which is through the `multiprocessing` library. In the script, the `worker()` function defines the task (or worker) for parallelization. The core function in the script, `monte_carlo()`, accepts six arguments: 1. `algos` contains a list of algorithms. The algorithms should match the names given as methods in the `BayesianAB` class. In our case, we have

```
algorithms = ['epsilon_greedy', 'optim_init_val', 'ucb1', 'gradient_bandit', 'bayesian_bandits']
```

2. `m` is the number of simulations/trials to run. The default value is 500. We will actually be running 2000 simulations;
3. `n` is the number of rounds/visitors in each simulation. A default value of 10001 means it will have 10000 visitors;
4. `p_max` is the highest win rate;
5. `p_diff` is the smallest possible difference between the highest win rate and the second highest win rate;
6. `p_min` is the lowest possible win rate.

We will run simulations with 12 different combinations of `p_max`, `p_diff`, and `p_min`, given in the following list:

```
    prob_list = [[.35, .1, .1], [.35, .05, .1], [.35, .01, .1],
                 [.75, .1, .1], [.75, .05, .1], [.75, .01, .1],
                 [.75, .1, .62], [.75, .05, .62], [.75, .01, .62],
                 [.95, .1, .82], [.95, .05, .82], [.95, .01, .82],
                 ]
```

The `run_monte_carlo()` function calls the `monte_carlo()` function, then processes the results and calculate success rate in each round. The results are stored in a `dictionary` named `df_all`.

The `plot_monte_carlo()` function, as the name suggests, plots the results in a 4-by-3 grid. Each subplot corresponds to a certain combination of `[p_max, p_diff, p_min]` which is the titles of the subplots.

Here is the resulted plot with 2,000 simulations, each with 10,000 rounds/visitors:

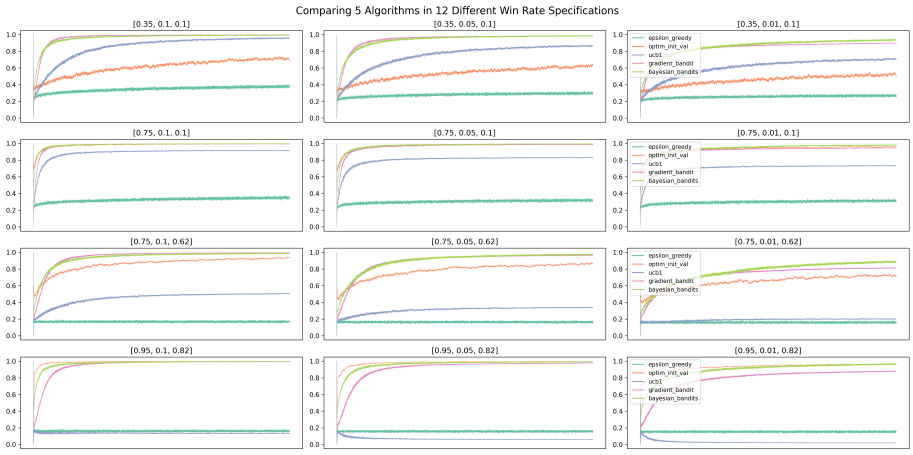

Figure 1.13: Comparison

Several results are worth mentioning: 1. `Thompson Sampling` and `Gradient Bandit` both do consistently well. `Thompson Sampling` has the best overall performance, picking the best bandit in over 90% of the simulations at the end of 10,000 rounds. It also edged out `Gradient Bandit` when `p_diff` is 0.01, which means there can be a close second-best bandit; 2. The `Epsilon Greedy` algorithm performs consistently regardless of win rate settings, but at a poor 20% success rate at picking the best bandit. This may be partly due to a relatively high value of epsilon at 0.5; 3. The algorithm that is the most sensitive to win rate settings is `UCB1`. When the win rate of the best bandit is at 0.95, `UCB1`'s performance can be puzzling. In part, it may be a the result of a relatively high `c` value, since the default is 1. Intuitively, when the best bandit has a win rate of 0.95, and especially when there exists a close second-best, the "bonus" that `UCB1` can give to a bandit is small. After all, the win rate is not to be exceeding 1. As a result, `UCB1` has a hard time distinguishing between the best bandits and others that are almost as good. It should be noted that `UCB` (not `UCB1`) has the best performance in the testbed in Sutton and Barto (2020), but the authors also acknowledged that `UCB`'s application beyond the Multi-Armed Bandit problem is limited (in the context of reinforcement learning).

1.11 Summary and Extensions

In this article, I have introduced five algorithms that can be used in *real-time* A/B Testing and Randomized Controlled Trials. Compared to traditional methods that often use Power Analysis to determine the minimum sample size, algorithms introduced in this article have one additional advantage: They can be

easily extended to experiments with more than 2 choices/versions/bandits, as shown throughout the article.

The algorithms introduced in this article are known as algorithms for the `Multi-Armed Bandit` problem, which is considered as the simplest form of **reinforcement learning**. We will come back to more reinforcement learning algorithms and problems in later articles.

Two extensions of the `Multi-Armed Bandit` problem should be mentioned: `Non-stationary Bandit` and `Contextual Bandit`. Non-stationary Bandit is the situation in which that the win rates change over time. One particular algorithm that we introduced in this article is known to do badly in non-stationary bandit problems: `Optimistic Initial Values`. The reason is obvious. `Optimistic Initial Values` algorithm is designed to explore aggressively in the beginning. When the win rates change after this initial exploration stage has ended, it is hard for it to change course.

As the name suggested, `Contextual Bandit` means that there exists contextual information to be used when making a decision. In a simple example, a casino may have slot machines in different colors, and the colors are not random: the green machines have higher win rates than the red ones. A new player would not know that information at first, but as time goes on and if the player has explored enough, it is possible to figure out the *association* between color and win rate, and hence decisions are made accordingly. This is also why `Contextual Bandit` is also known as `Associative Search`. We will come back to this in another article, since `Contextual Bandit` algorithms can be used in regression problems.

(Visit my GitHub for the latest Python scripts: https://github.com/DataHurdler/Econ-ML/tree/main/Multi-Arm%20Bandits)

1.12 References

- https://www.udemy.com/course/bayesian-machine-learning-in-python-ab-testing/
- http://incompleteideas.net/book/the-book-2nd.html (Chapter 2)
- https://en.m.wikipedia.org/wiki/Multi-armed_bandit
- https://www.tensorflow.org/agents/tutorials/intro_bandit

Chapter 2

Discrete Choice, Classification, and Tree-Based Ensemble Algorithms

2.1 Introduction

Suppose you are the owner of an e-commerce website that sells a musical instrument in 5 sizes: soprano, alto, tenor, bass, and contrabass. You are now considering to open up some physical stores. With physical stores, you need to make more careful inventory decisions. Based on your past experience in shipping out your instruments, you are convinced that different communities can have different tastes toward different sizes. And you want to make inventory decisions accordingly.

If the stake for a musical store is too small, consider the classic discrete choice example: automobiles. Ultimately, we want to understand the question of "who buys what." This can inform many business decisions, not only inventory, since the aggregation of individuals tells us the demand of a product in a market. Discrete choice is different from decisions about "how much," for example, do I exercise 15 minutes or 25 minutes tonight?

In economics and social sciences, the popular approaches are "top-down": it starts with a understanding of the data-generating process and some assumptions. Logit and Probit are two widely used models in economics. If the error term is believed to follow a logistic distribution, then use the logit model or logistic regression. If the error term is believed to follow a normal distribution,

then use the probit model. If there is nested structure, then use nested-logit. And so on.

This is fine, since the focus of economics and social sciences is hypothesis testing and to understand mechanisms. On the contrary, the machine learning approach is *bottom-up*: it cares about making good predictions. In a way, the economics "top-down" approach of discrete choice modeling cares much more about "bias" whereas the machine learning approach considers the bias-variance tradeoff more holistically.

2.2 The Bias-Variance Tradeoff

Let's begin with *Variance*. A model with a high variance is sensitive to the *training* data and can capture the fine details in the training data. However, such model is usually difficult to generalize. On the one hand, the *test* data, or the data that the model is actually applied to, may lack the fine detail presented in the training data. On the other hand, those fine details may not be as important in the actual data as compared to the training data.

A model that can capture fine details is almost guaranteed to have low *bias*. A model with low bias is one that explains the known, or training, data well. In order to predict, we need our machine learning model to learn from known data. A model with high bias can rarely predict well.

While models with low bias *and* low variance do exist, they are rare. Since a model with high bias rarely works well, lowering bias is often considered the first-order task. One way to do so is using models, or specifying hyperparameters of a model, such that more fine details in the data are taken in to consideration. By doing so, higher variance is introduced. And hence the trade off.

Consider the following example: a zoo wants to build a machine learning algorithm to detect penguin species and deploy it on their smart phone application. Let's say all that the zoo and its users care about is to tell apart King, Magellanic, and Macaroni penguins. The zoo's staffs and data scientists took hundreds of photos of penguins in their aquarium, split the penguins into training and test datasets, as how tasks like this are usually performed, and built the machine learning model. In their test, with photos that they have set aside earlier, they find that the algorithm is able to identify the penguins correctly 98% of the time.

However, when their users use the algorithm to identify penguins in other zoos, the algorithm fails miserably. Why? It turns out that the machine learning algorithm was not learning to identify penguins by their different features such as head, neck, and tails. Instead, the algorithm identifies the different species of penguins by the tag on their wings: blue is for King penguin, red for Magellanic, and yellow for Macaroni. These are the colors used by the zoo who developed the algorithm, but different zoos have different tags. As a result, this algorithm,

which has low bias but high variance, is unable to predict or detect the species of penguins outside of the zoo where photos for developing the machine learning model were taken.

As we will see next, tree-based algorithms are extremely prone to high variance, or *over-fitting*.

2.3 Decision Tree

Let's first talk about the basic decision tree algorithm. Because we will be using scikit-learn for `Python` implementation in this chapter, I am using notations and languages similar to that in scikit-learn's documentations. It should be mentioned at the outset that this is not a comprehensive lecture on the decision tree algorithm. You can easily find more in-depth discussions in books such as An Introduction to Statistical Learning and The Elements of Statistical Learning, or other online learning sources. Our focuses here are aspects of the decision tree algorithm that matter the most for the understanding of the ensemble methods and their applications in economics and business.

The simplest way to think about a decision tree algorithm is to consider a flow-chart, especially one that is for diagnostic purposes. Instead of someone building a flow-chart from intuition or experience, we feed data into the computer and the decision tree algorithm would build a flow-chart to explain the data. For example, if we know some characteristics of consumers of the music store, and want to know who is more likely to buy the smaller size instruments, a flow-chart, built by the decision tree algorithm, may look like this: * Is the customer under 30? * Yes: is the customer female? * Yes: has the customer played any instrument before? * Yes: the customer has a 90% chance buying a smaller instruments * No: the customer has 15% chance buying a smaller instruments * No: is the customer married? * Yes: the customer has a 5% chance buying a smaller instruments * No: the customer has 92% chance buying a smaller instruments * No: is the customer under 50? * Yes: the customer has a 10% chance buying a smaller instruments * No: has the customer played any instrument before? * Yes: the customer has a 100% chance buying a smaller instruments * No: the customer has a 20% chance buying a smaller instruments

You can see several basic elements of a decision tree algorithm from the above example:

1. As expected, the tree algorithm resulted in a hierarchical structure that can be easily represented by a tree diagram;
2. The tree structure does not need to be symmetrical. For example, when the answer to "is the customer under 50" is a "yes", the branch stopped, resulted in a shorter branch compared to the rest of the tree;
3. You may use the same feature more than once. In this example, the question "has the customer played any instrument before" has appeared twice. Also there are two splits based on two different age cutoffs;

4. You can use both categorical and numerical features. In this example, age is numerical, whereas all other features are categorical;
5. It is accustomed to split to only two branches at each node. If you want three branches, you can do it at the next node: two branches at the first node, then one or both of the next nodes split into another 2 branches.

There are other elements of a decision tree algorithm that you can not observe directly from this example but are very important. We will examine these in more details below.

2.4 Split Criterion

At each node, the split must be based on some criterion. The commonly used criteria are **Gini impurity** and **Entropy** (or **Log-loss**). According to the scikit-learn documentation, let

$$p_{mk} = \frac{1}{n_m} \sum_{y \in Q_m} I(y = k)$$

denote the proportion of class k observations in node m, where Q_m is the data at node m, n_m is the sample size at node m, and $I(\cdot)$ returns 1 when $y = k$ and 0 otherwise. Then, the **Gini impurity** is given by:

$$H(Q_m) = \sum_k p_{mk}(1 - p_{mk})$$

whereas **Entropy** is given by:

$$H(Q_m) = -\sum_k p_{mk} \log(p_{mk})$$

At each node m, a `candidate` is defined by the combination of feature and threshold. For example, in the above example, for the question "Is the customer under 30," the feature is age and the threshold is 30. Let θ denote a candidate, which splits Q_m into two partitions: Q_m^{left} and Q_m^{right}. Then the quality of a split with θ is computed as the weighted average of the criterion function $H(Q_m)$:

$$G(Q_m, \theta) = \frac{n_m^{\text{left}}}{n_m} H(Q_m^{\text{left}}(\theta)) + \frac{n_m^{\text{right}}}{n_m} H(Q_m^{\text{right}}(\theta))$$

The objective of the decision tree algorithm is to find the candidate that minimizes the quality at each m:

$$\theta^* = \underset{\theta}{\text{argmin}} G(Q_m, \theta)$$

It is straightforward to see that, from either the **Gini impurity** or the **Entropy** criterion function, the unconstrained minimum of $G(Q_m, \theta)$ is achieved at $p_{mk} = 0$ or $p_{mk} = 1$, i.e., when the result of the split consists of a single class.

Before we move on, here is a quick remark: while there exists a global optimum for building a decision tree, where the quality function is minimized for the *whole* tree, the computation of such algorithm is too complex. As a result, practical decision tree algorithms resolve to using *local* optima at each node as described above.

2.5 Pruning

If achieving a "pure" branch, where there remains observations from a single class after a split, minimizes the quality function $G(Q_m, \theta)$, then why did we not achieve that "pure" state in the illustrative example with the music store? There are two main reasons. First, we may not have enough features. Imagine you have two individuals in your data set, one bought a small instrument and the other bought a large instrument. These two individuals are almost identical: the only difference is in their eye colors. If "eye color" is not one of the features captured in your data set, you will have no way to distinguish these two individuals. On the other hand, imagine we know *everything* about each and every individual, then it is almost guaranteed that you can find a "perfect" tree, such that there is a single class of individuals at each end node. Such "perfect" tree may not be unique. At the extreme, imagine a tree such that each end node represents a single individual.

The second reason is something we have discussed: the Bias-Variance Tradeoff. Because the ultimate goal is to predict, fitting a "perfect" tree can result in too high of a variance. Continued with the previous example, your ability to build a perfect tree would totally depend on whether you have "eye color" as a feature in your data set. That means that your algorithm is too sensitive to one particular feature - if this feature does not exist, your algorithm would fail to build a "perfect" tree (assuming that was the goal). Or, if this feature is somehow absent or incorrect in the data set you are predicting on, your algorithm would have a breakdown.

This is why a decision tree needs to be pruned. This is often done by specifying two hyperparameters in the decision tree algorithm in scikit-learn: the maximum depth of the tree (`max_depth`) and the minimum number of samples required to split (`min_samples_split`). Without going into the technical details, we can intuitively understand that both of these restrictions can prevent us from splitting the tree to the extreme case such that each end node represents an individual. In other words, they restrict the growth of a tree.

The caveat of a single decision tree algorithm is obvious: it can easily suffer from either high bias or high variance, especially the latter. This is why ensemble methods such as **bagging** and **boosting** were introduced. In practice, a

single decision tree is rarely used as the "finale" model. It is often used as a demonstrative example.

2.6 Bagging and Random Forest

`Bagging` is one of two ensemble methods based on the decision tree algorithm. `Bagging` is short for *bootstrap aggregation*, which explains what bagging algorithms do: select random subsets from the training data set, fit the decision tree algorithm on each subset, and aggregate to get the prediction. There are several variations of `Bagging` algorithms depending on how random samples are drawn:

1. When random subsets were drawn with replacement (bootstrap), the algorithm is known as `Bagging` (Breiman, 1996)
2. When random subsets were drawn without replacement, the algorithm is known as `Pasting` (Breiman, 1999)
3. When random subsets are drawn based on features rather than individuals, the algorithm is known as `Subspaces` (Ho, 1998)
4. When random subsets are drawn based on both features and individuals, the algorithm is known as `Random Patches` (Louppe and Geurts, 2012)
5. When random subsets were drawn with replacement (bootstrap) *and* at each split, a random subset of features is chosen, the algorithm is known as `Random Forest` (Breiman, 2001)

In scikit-learn, the first four algorithms are implemented in `BaggingClassifier` whereas `Random Forest` is implemented in `RandomForestClassifier`.

In bagging algorithms, the "aggregation" of results during prediction is usually taken by votes. For example, suppose you have fit your data with the `Random Forest` algorithm with 1,000 trees, and now you want to know whether a new customer is going to buy a small or a large instrument. When the algorithm considers the first split, it will look at all 1,000 trees and see which candidate was used the most. Suppose "Is the customer under 30" appeared in 800 of the trees, then the algorithm would split according to `age=30`. And so, at each split, the algorithm would take a tally from the 1,000 individual trees and act accordingly, just like how one would look at a flow-chart to determine actions.

While a `Bagging` algorithm helps to reduce bias, the main benefit of bootstrapping is to reduce variance. The `Random Forest` algorithm, for example, is able to reduce variance in two ways: First, bootstrapping random samples is equivalent to consider many different scenarios. Not only does this mean that the algorithm is less reliant on a particular scenario (the whole training data set), it also makes it possible that one or some of the random scenarios may be similar to the "future," i.e., the environment that the algorithm needs to make prediction on. Second, by considering a random set of features at each split, the algorithm is less reliant on certain features, and is hence resilient to "future" cases where certain features may be missing or have errors.

2.7 Boosting and AdaBoost

While the main benefit of `Bagging` is in reducing variance, the main benefit of `Boosting` is to reduce bias, while maintaining a reasonably low variance. Boosting is able to maintain a low variance because, like Bagging, it also fits many trees. Unlike Bagging, which builds the trees in parallel, Boosting builds them sequentially.

The basic idean of boosting is to have incremental (small/"weak") improvements from the previous model, which is why the learning algorithms are built sequentially. This idea can be applied to all types of algorithms. In the context of decision tree, a boosting algorithm can be demonstrated by the following pseudocode:

```
Step 1: Build a simple decision tree (weak learner)
Step 2: Loop:
            Minimize weighted error
```

Currently, there are three popular types of tree-based boosting algorithms: `AdaBoost`, `Gradient Boosting`, and `XGBoost`. The different algorithms are different in how they *boost*, i.e., how to implement Step 2.

`AdaBoost` was introduced by Freund and Schapire (1995). It is short for *Ada*ptive *Boost*ing. `AdaBoost` implements boosting by changing the weights of observations. That is, by making some observations/individuals more important than the other. In a training data set with N individuals, the algorithm begins by weighting each individual the same: at a weight of $1/N$. Then it fits a simple decision tree model and makes predictions. Inevitably, it makes better decision for some individuals than the other. The algorithm then increases the weight for individuals that it did not make correct/good predictions on in the first model. Effectively, this asks the next decision tree algorithm to focus more on these individuals that it has failed to understand in the first tree. And this process continues until a stopping rule is reached. A stopping rule may be, for example, "**stops** when 98% of the individuals are correctly predicted".

It is straightforward to see that a boosting algorithm lowers bias. But was it often able to main a low *variance* too? It was able to do so because a boosting algorithm effectively builds different trees at each iteration. When making predictions, it takes a weighted average of the models. Some mathematical details may be helpful.

Let w_{ij} denote the weight of individual i in stage/iteration j. In the beginning of the algorithm, we have $w_{i1} = 1/N$ for all i where N is the the total number of individuals. After the first weak tree is built, we can calculate the error/misclassification rate of stage j as

$$e_j = \frac{\sum_N w_{ij} \times I_{ij}(\text{incorrect})}{\sum_N w_{ij}}$$

where I_{ij}(incorrect) equals 1 if the prediction for individual i is incorrect in stage j and 0 otherwise. We can then calculate the *stage value* of model j:

$$v_j = \frac{1}{2} \log\left(\frac{1 - e_j}{e_j}\right)$$

The stage value is used both in updating w_{ij+1}, i.e., the weight of individual i in the next stage, and to act as the weight of model j when prediction is computed. To update the weight for the next stage/model, we have

$$w_{ij+1} = w_{ij} \times \exp\left(v_j \times I_{ij}(\hat{y}_{ij} = y_i)\right)$$

where $\hat{y}_{ij}$ is the prediction for individual i in stage j, and y_i is the true label for individual i. For binary classification, it is conventional to expression $\hat{y}_{ij}$ and y_i as 1 and -1, so that the above equation can be simplified into

$$w_{ij+1} = w_{ij} \times \exp\left(v_j \times \hat{y}_{ij} \times y_i\right)$$

At each stage $j(> 1)$, the `AdaBoost` algorithm aims to minimize e_j.

To compute the overall/final prediction, let $\hat{y}_{ij}$ denote the prediction of model/stage j for individual i, then the predicted value is calculated by:

$$\hat{y}_i = \sum_J \hat{y}_{ij} \times v_j$$

where J is the total number of stages.

2.8 Gradient Boosting and XGBoost

`Gradient Boosting` (Friedman, 2001) is another approach to boost. Instead of updating the weight after each stage/model, Gradient Boosting aims to minimize a loss function, using method such as gradient decent. The default loss function in scikit-learn, which is also the most common in practice, is the binomial deviance:

$$L_j = -2 \sum_N y_i \log\left(\hat{p}_{ij}\right) + (1 - y_i) \log\left(1 - \hat{p}_{ij}\right)$$

where N is the number of individuals, y_i is the true label for individual i, and $\hat{p}_{ij}$ is the predicted probability that individual i at stage j having a label of y, and is given by the softmax (logistic) function when log-loss is specified:

$$\hat{p}_{ij} = \frac{\exp\left(F_j(x_i)\right)}{1 + \exp\left(F_j(x_i)\right)}$$

where $F_j(x_i)$ is a numerical predicted value for individual i by regressor $F_j(x)$. Here, $F_j(x)$ is the aggregated regressor in stage j, which is given by

$$F_j(x) = F_{j-1}(x) + h_j(x)$$

where $h_j(x)$ is the weak learner/regressor at stage j that minimizes L_j. Substituting $F_M(x)$, the final regressor, into the above formula for $\hat{p}_{ij}$ gives the overall prediction of the Gradient Boosting model.

Finally, using first-order Taylor approximation, it can be shown that minimizing L_j is approximately equivalent to predicting the negative gradient of the samples, where the negative gradient for individual i is given by

$$-g_i = -\left[\frac{\partial l_{ij-1}}{\partial F_{j-1}(x_i)}\right]$$

where l_{ij-1} is the term inside the summation in L_j (but lagged one stage):

$$l_{ij-1} = y_i \log\left(\hat{p}_{ij-1}\right) + (1 - y_i) \log\left(1 - \hat{p}_{ij-1}\right)$$

In other words, while the basic decision tree algorithm aims to predict the true classes, usually represented by 0's and 1's, `Gradient Boosting` aims to predict a numerical value which is the gradient. This means that, at each stage, Gradient Boosting is a regression problem rather than a classification problem. Predicting the gradient allows the algorithm to utilize many well developed methods for such task, for example, the Nelder-Mead method or simple grid search.

The discussion above focused on binary classification, which requires a single tree to be built in each stage. In multiclass classification, K trees would be built for K classes. For example, if `Gradient Boosting` is used to identify the 26 English alphabets, 26 trees are built and fitted in each stage.

`XGBoost` was introduced by Tianqi Chen in 2014. It is short for "e*X*treme *G*radient *Boost*ing". Instead of gradient decent, `XGBoost` implements Newton's Method, which is computationally much more demanding than gradient decent and requires a second-order Taylor approximation (instead of first-order as in `Gradient Boosting`). Due to this, in addition to **Gradients**, `XGBoost` also calculates the **Hessians**, which are a set of second-order derivatives (whereas gradients are the first-order derivatives).

`Python` library `xgboost` implements `XGBoost` and can easily be integrated with `scikit-learn`, which is the library we use to implement all algorithms covered in this chapter.

2.9 Python Implementation with scikit-learn

As we have done in other chapters, we will first generate a data set, then fit the data with various algorithms. After we have fitted the models, we will print out some basic performance metrics, chief among which is the `confusion matrix` and conduct a cross validation exercise.

The algorithms we will consider include: * Logistic regression * Decision tree classifier * Random forest classifier * Adaboost classifier * Gradient boosting classifier * XGBoost

Even though logistic regression is not covered in this chapter, I included it in the Python implementation for comparison purposes. Although not necessary, I use a Python class in this implementation. Here is the full script:

```
import random
import string
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import xgboost

N_GROUP = 5
N_IND = 50000
N_FEATURES = 10


class TreeModels:
    def __init__(
            self,
            n_group: int = 5,
            n_individuals: int = 10000,
            n_num_features: int = 10,
```

```
        numeric_only: bool = False,
    ):
        """
        Initialize the TreeModels class.

        Args:
            n_group (int): Number of groups. Default is 5.
            n_individuals (int): Number of individuals. Default is 10000.
            n_num_features (int): Number of numerical features. Default is 10.
            numeric_only (bool): Flag to indicate whether to use only numerical features. Defaul

        Returns:
            None
        """
        print(f'There are {n_individuals} individuals.')
        print(f'There are {n_group} choices.')
        print(f'There are {n_num_features} numerical features and 1 categorical feature.')

        self.numeric_only = numeric_only

        # Generate random numerical features and categorical feature
        self.num_features = np.random.rand(n_individuals, n_num_features + 2)
        cat_list = random.choices(string.ascii_uppercase, k=6)
        self.cat_features = np.random.choice(cat_list, size=(n_individuals, 1))

        # Create a DataFrame with numerical features and one-hot encoded categorical feature
        self.df = pd.DataFrame(self.num_features[:, :-2])
        self.df['cat_features'] = self.cat_features
        self.df = pd.get_dummies(self.df, prefix=['cat'])
        self.df.columns = self.df.columns.astype(str)

        if numeric_only:
            # Cluster the data based on numerical features only
            # Logistic regression performs the best in this condition
            kmeans = KMeans(n_clusters=n_group, n_init="auto").fit(self.num_features)
            self.df['target'] = kmeans.labels_
        else:
            # Cluster the data based on both numerical and categorical features
            cat_columns = self.df.filter(like='cat')
            kmeans1 = KMeans(n_clusters=n_group, n_init="auto").fit(cat_columns)
            kmeans2 = KMeans(n_clusters=n_group, n_init="auto").fit(self.num_features)
            self.df['target'] = np.floor((kmeans1.labels_ + kmeans2.labels_) / 2)

        # Add some random noise to the numerical features
        numerical_columns = [str(i) for i in range(n_num_features)]
```

```
        for column in numerical_columns:
            self.df[column] = self.df[column] + random.gauss(mu=0, sigma=3)

        # Split the data into training and testing sets
        self.X = self.df.drop(columns=['target'])
        self.y = self.df['target']
        self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(
            self.X, self.y, test_size=0.3, random_state=42)

        # Initialize the y_pred variable
        self.y_pred = np.empty([n_individuals, 1])

        # Initialize a dictionary to save results
        self.results = dict()

    def show_results(self, clf, clf_name, print_flag=False, plot_flag=True):
        """
        Train and evaluate a classifier.

        Args:
            clf: Classifier object.
            clf_name (str): Name of the classifier.
            print_flag (bool): Whether to print results. Default is False.
            plot_flag (bool): Whether to draw CM plots and save them. Default is True.

        Returns:
            None
        """
        print(clf_name)
        clf.fit(self.X_train, self.y_train)
        self.y_pred = clf.predict(self.X_test)

        # Calculate evaluation metrics
        train_acc = clf.score(self.X_train, self.y_train)
        acc = accuracy_score(self.y_test, self.y_pred)
        precision = precision_score(self.y_test, self.y_pred, average='weighted')
        recall = recall_score(self.y_test, self.y_pred, average='weighted')
        f1 = f1_score(self.y_test, self.y_pred, average='weighted')

        # Perform cross-validation and print the average score
        cv_score = cross_val_score(clf, self.X, self.y, cv=10)

        if print_flag:
            if isinstance(clf, LogisticRegression):
                print(f'Coefficients: {clf.coef_}')
```

```
            else:
                print(f'Feature Importance: {clf.feature_importances_}')
            print(f'Training accuracy: {train_acc:.4f}')
            print(f'Test accuracy: {acc:.4f}')
            print(f'Test precision: {precision:.4f}')
            print(f'Test recall: {recall:.4f}')
            print(f'Test F1 score: {f1:.4f}')
            print(f'Average Cross Validation: {np.mean(cv_score)}')

        if plot_flag:
            # Plot the confusion matrix
            cm = confusion_matrix(self.y_test, self.y_pred, labels=clf.classes_)
            disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
            disp.plot()

            plt.savefig(f"cm_{clf_name}_{self.numeric_only}.png", dpi=150)

        plt.show()

        # Save results in self.result dictionary
        self.results[clf_name] = {
            'train_acc': train_acc,
            'acc': acc,
            'precision': precision,
            'recall': recall,
            'f1_score': f1,
            'cv_score': np.mean(cv_score)
        }


def run_tree_ensembles(
        n_group: int = 5,
        n_num_features: int = 10,
        print_flag: bool = True,
        plot_flag: bool = True,
        numeric_only_bool: list = (False, True),
        n_individuals: int = 50000,
) -> dict:

    for i in numeric_only_bool:
        tree = TreeModels(n_group, n_individuals, n_num_features, numeric_only=i)

        logit = LogisticRegression(max_iter=10000)
        tree.show_results(logit, 'logit', print_flag, plot_flag)
```

```
        d_tree = DecisionTreeClassifier()
        tree.show_results(d_tree, 'decisiontree', print_flag, plot_flag)

        rf = RandomForestClassifier()
        tree.show_results(rf, 'randomforest', print_flag, plot_flag)

        ada = AdaBoostClassifier()
        tree.show_results(ada, 'adaboost', print_flag, plot_flag)

        gbm = GradientBoostingClassifier()
        tree.show_results(gbm, 'gbm', print_flag, plot_flag)

        xgb = xgboost.XGBClassifier()
        tree.show_results(xgb, 'xgboost', print_flag, plot_flag)

        return {n_individuals: tree.results}
```

Here are some remarks about the script. First, the number of numerical features in the generated data set is given by `n_num_features`. Two additional columns of numerical features and six columns of string/categorical features are also included to add randomness and complexity to the generated data. The numerical features are stored in the `numpy` array `num_features` while the categorical features are stored in `cat_features`. These features are then properly processed and stored in the `pandas` dataframe `df`:

- Only the original numerical feature columns are stored (`self.num_features[:, :-2]`);
- The categorical features are one-hot encoded with `pd.get_dummies()`.

In the `if` statement that followed, the `Kmeans` algorithm is called to generate `n_group` classes/clusters.

Additional randomness is added to the numerical features by:

```
        # Add some random noise to the numerical features
        numerical_columns = [str(i) for i in range(n_num_features)]
        for column in numerical_columns:
            self.df[column] = self.df[column] + random.gauss(mu=0, sigma=3)
```

The rest of the `TreeModels` class performs the train-test split and adds a method named `show_results()` to run the selected algorithm then print out (based on the value of `print_flag`) several performance metrics.

2.10 Confusion Matrix and other Performance Metrics

`Confusion matrix` is the most important and common way to examine the performance of a classification algorithm. It is a matrix showing the numbers of individuals in each true-predicted label combination. In our simulated data, there are 5 classes, which results in a 5-by-5 confusion matrix. Below is the confusion matrix of the test data from the logistic regression. The simulation has included categorical features in generating the target groups:

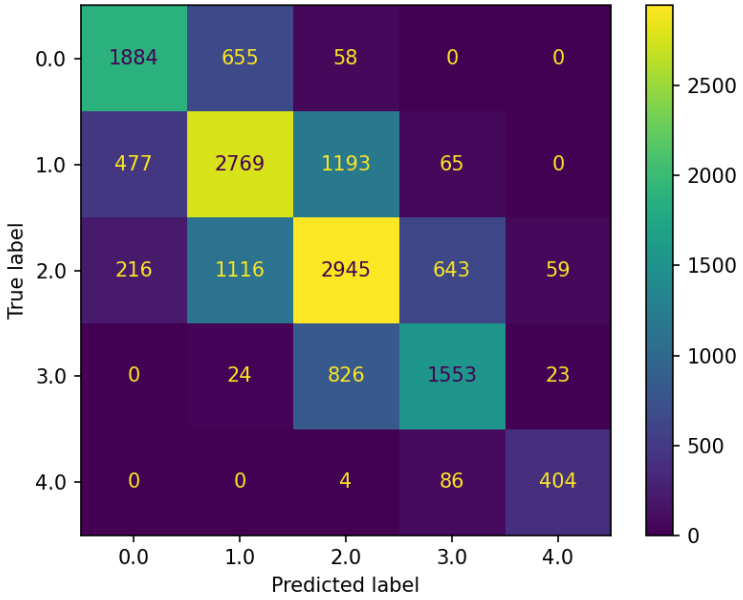

Figure 2.1: Comparison

In the confusion matrix, the rows show the "True label" whereas the columns show the "Predicted label". All the cells on the diagonal are corrected predicted. Based on the confusion matrix, there are three basic performance metrics: **accuracy**, **precision**, and **recall**. There are also various metrics that are weighted averages. For example, the **f1 score** is the harmonic mean of precision and recall.

Accuracy is the proportion of individuals that the algorithm has predicted correctly. To calculate the accuracy score, we sum up the values on the diagonal then decide the total:

$$\frac{1884 + 2769 + 2945 + 1553 + 404}{15000} = 0.6370$$

Precision and recall are usually defined based on a certain class. For overall precision and recall scores, we can then take a weighted average. Precision is the proportion of individuals who the algorithm predicted to be a certain class is actually that class. In the above example, 2577 individuals were predicted to be class 0, but only 1884 actually are. As a result, the precision *for class 0* is:

$$\frac{1884}{2577} = 0.7311$$

On the other hand, recall is the proportion of individuals who belong to a certain class that the algorithm predicted correctly. IN the above example, 4979 individuals belong to class 2, but only 2945 were predicted correctly by the algorithm. As a result, the recall *for class 2* is:

$$\frac{2945}{4979} = 0.5915$$

If we take weighted average of precision and recall of all 5 classes, we get the overall precision and recall scores as 0.6376 and 0.6370, or about 63.7%.

Economics and social sciences often use the terms "Type I" and "Type II" errors, which can be related to the discussion here in a binary classification. In a binary classification, we have 4 quadrants: 1. True positive (TP): those who belong to the "positive" class and are predicted so; 2. True negative (TN): those who belong to the "negative" class and are predicted so. True positive and true negative are on the diagonal; 3. False positive (FP): those who are predicted to be "positive" but are actually "negative'; 4. False negative (FN): those who are predicted to be"negative" but are actually "positive".

Type I error corresponds to false positive and Type II error corresponds to false negative.

Before we move on to formally compare results from the 6 algorithms, it is worth noting that random forest, gradient boosting, and XGBoost performed much better than logistic regression in the above simulated data set (with `random seed = 123`). For example, below is the confusion matrix from XGBoost:

2.11 Comparison the Algorithms

The following `Python` script runs the comparison between different algorithms for between 6000 and 50000 individuals (sample size):

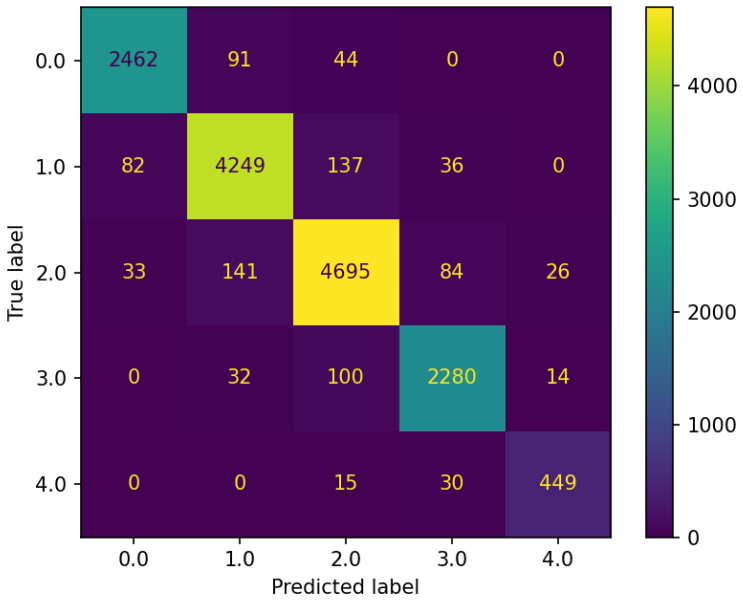

Figure 2.2: Comparison

```
import matplotlib.pyplot as plt
import random
import pandas as pd
from multiprocessing import Pool, cpu_count
from functools import partial
from tree_ensembles import run_tree_ensembles

plt.ion()

n_individuals_range = range(50000, 5999, -2000)


def run_monte_carlo(n_individuals_range, numeric_only_bool):

    with Pool(1) as pool:
        func = partial(run_tree_ensembles, 5, 10, False, False, numeric_only_bool)
        results = list(pool.imap(func, n_individuals_range))

    return results


def plot_monte_carlo(data: list):

    df_list = []
    for item in data:
        for i, inner_dict in item.items():
            for j, inner_inner_dict in inner_dict.items():
                value = inner_inner_dict['cv_score']
                df_list.append({'i': i, 'Model': j, 'cv_score': value})

    df = pd.DataFrame(df_list)

    fig, ax = plt.subplots()

    num_models = len(df['Model'].unique())
    cmap = plt.get_cmap('Set2')  # Use the Set2 color map

    for i, model in enumerate(df['Model'].unique()):
        model_data = df[df['Model'] == model]
        color = cmap(i % num_models)  # Cycle through the color map
        ax.plot(model_data['i'], model_data['cv_score'], '-o', c=color, label=model, al

    ax.set_xlabel('Number of Individuals')
    ax.set_ylabel('Cross Validation Scores')
    ax.set_title('Plot of Cross Validation Scores')
```

```
    ax.legend(['Logit', 'Decision Tree', 'Random Forest', 'Adaboost', 'GBM', 'XGBoost'],
              loc='lower right',
              fontsize=9, markerscale=1.5, scatterpoints=1,
              fancybox=True, framealpha=0.5)
```

The script intends to use parallel computing, but algorithms in `scikit-learn` already utilized parallel computing, we set `Pool(1)` at the end to run it with single thread. Two comparisons, with and without using the categorical features in generating the target groups, are run. The average score from 10-fold cross validations are recorded and plotted. Here is the result from when `kmeans` generated the target groups without using the categorical columns (but they are still in the training data):

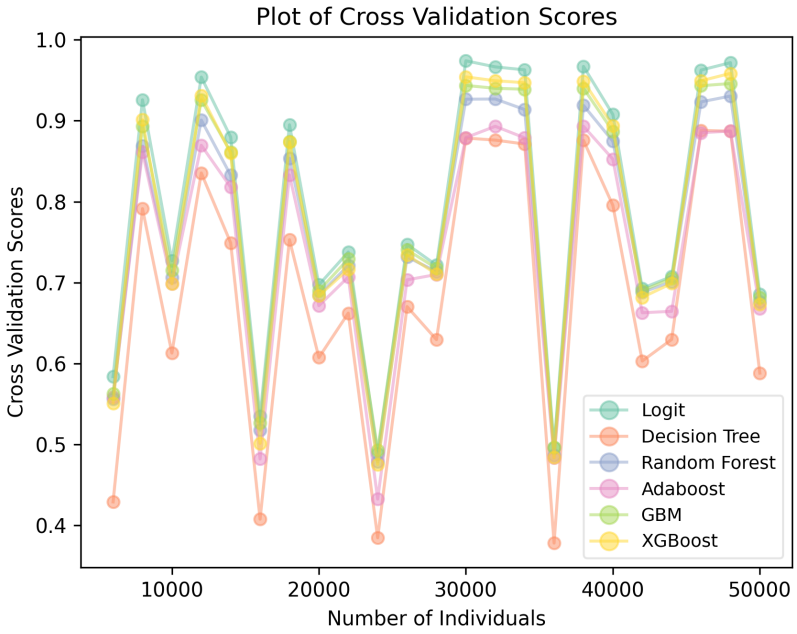

Figure 2.3: Comparison

A single decision tree performed the worst, while, surprisingly, logistic regression performed the best. Keep in mind that the ups and downs at different sample sizes do not indicate that more data is worse. There is some random components in how the data was generated, namely with `kmeans` algorithm.

When categorical columns are included in generating the target groups, there exists more variations among algorithms:

In here, `Adaboost` performed noticeably worse than all other algorithms. Lo-

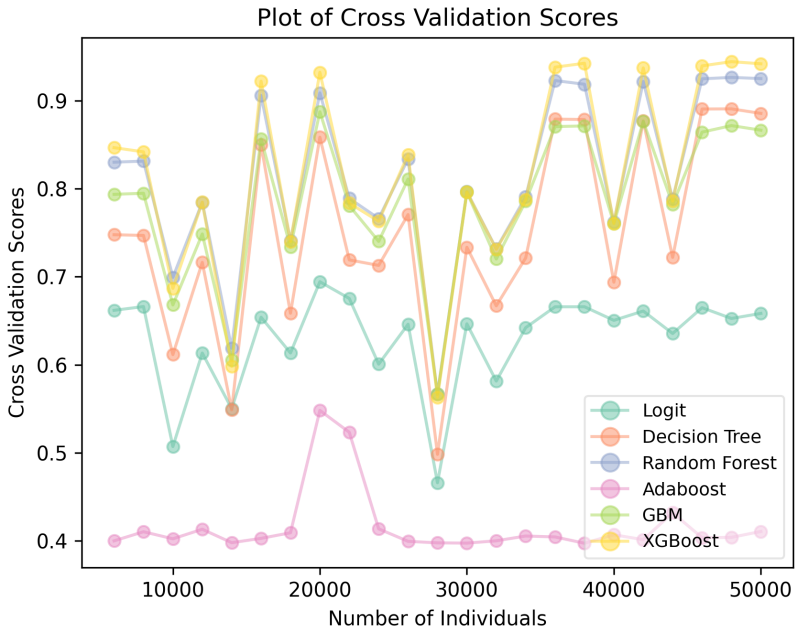

Figure 2.4: Comparison

gistic regression also fell, partly because its inability to deal with categorical features (even with one-hot encoding). Not surprisingly, the performances of `Random Forest`, `Gradient Boosting`, and `XGBoost` remain strong.

2.12 Summary

In this chapter, we have covered the decision tree algorithm as well as bagging and boosting algorithms based on decision tree. There are few important takeaways and remarks.

First, ensemble methods is a general method that applies to algorithms beyond tree-based models. You could easily applied the same principle of bagging and boosting on regression models. For example, you can build several regressors with a bootstrap data set, or include only some of the features, or use weighted methods to boost. As a matter of fact, ensemble can also be built between regression and classification algorithms. Gradient Boosting and XGBoost can actually be considered as such ensemble: while the end-goal of the algorithms were to predict classes, at their core, they are regressions.

Second, tree-based models can be used for regression problems. For example, instead of `RandomForestClassifier`, you can use `RandomForestRegressor` for a regression problem. When you are using a classification algorithm on a continuous target. Instead of trying to predict classes, the `RandomForestRegressor`, as well as other classification used for regression problems, aims to predict the mean of the target. We will cover this in more depth in a later chapter.

Third, in general, it is more accurate to predict classes than continuous values. Due to this, the use of classification algorithms may be broader than most expected. For example, it is possible to convert a regression problem (in predicting continuous quantities) to classification problems. The market share example given in the beginning of the chapter is a good example. Another example is e-commerce. Most e-commerce owners have a limited offering. As a result, instead of predicting the sales per month or the dollar value of a customer, it is easier to predict whether and how many a customer would buy. This method can be especially powerful since a business owner often has control over the price of the products.

Lastly, tree-based methods can be used for causal inference. While causal inference itself is a topic of a later chapter, for readers who are familiar with causal inference methods, you can easily find parallel between decision tree and propensity score matching (PSM): individuals who ended up in the same leave have something in common, and hence can provide good matching samples. This is the basic idea behind `causal tree` (Athey and Imbens, 2016).

2.13 References

- S. Athey and G. Imbens, "Recursive partitioning for heterogeneous causal effects", *PNAS*, 2016.
- L. Breiman, "Bagging predictors", *Machine Learning*, 1996.
- L. Breiman, "Pasting small votes for classification in large databases and on-line", *Machine Learning*, 1999.
- L. Breiman, "Random forest", *Machine Learning*, 2001.
- Y. Freund and R. Schapire, "A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting", 1995.
- J. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine", *The Annals of Statistics*, 2001.
- T. Ho, "The random subspace method for constructing decision forests", *Pattern Analysis and Machine Intelligence*, 1998.
- G. Louppe and P. Geurts, "Ensembles on Random Patches", *Machine Learning and Knowledge Discovery in Databases*, 2012.
- https://scikit-learn.org/stable/modules/tree.html
- https://xgboost.readthedocs.io/en/stable/tutorials/model.html
- https://machinelearningmastery.com/boosting-and-adaboost-for-machine-learning/
- https://stats.stackexchange.com/questions/157870/scikit-binomial-deviance-loss-function
- https://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/slides/

Chapter 3

Parts

You can add parts to organize one or more book chapters together. Parts can be inserted at the top of an .Rmd file, before the first-level chapter heading in that same file.

Add a numbered part: `# (PART) Act one {-}` (followed by `# A chapter`)

Add an unnumbered part: `# (PART\*) Act one {-}` (followed by `# A chapter`)

Add an appendix as a special kind of un-numbered part: `# (APPENDIX) Other stuff {-}` (followed by `# A chapter`). Chapters in an appendix are prepended with letters instead of numbers.

Chapter 4

Footnotes and citations

4.1 Footnotes

Footnotes are put inside the square brackets after a caret `^[]`. Like this one [1].

4.2 Citations

Reference items in your bibliography file(s) using `@key`.

For example, we are using the **bookdown** package [**?**] (check out the last code chunk in index.Rmd to see how this citation key was added) in this sample book, which was built on top of R Markdown and **knitr** [**?**] (this citation was added manually in an external file book.bib). Note that the `.bib` files need to be listed in the index.Rmd with the YAML `bibliography` key.

The RStudio Visual Markdown Editor can also make it easier to insert citations: https://rstudio.github.io/visual-markdown-editing/#/citations

[1] This is a footnote.

Chapter 5

Blocks

5.1 Equations

Here is an equation.

$$f(k) = \binom{n}{k} p^k (1-p)^{n-k} \tag{5.1}$$

You may refer to using `\@ref(eq:binom)`, like see Equation (5.1).

5.2 Theorems and proofs

Labeled theorems can be referenced in text using `\@ref(thm:tri)`, for example, check out this smart theorem 5.1.

Theorem 5.1. *For a right triangle, if c denotes the* length *of the hypotenuse and a and b denote the lengths of the* ***other*** *two sides, we have*

$$a^2 + b^2 = c^2$$

Read more here https://bookdown.org/yihui/bookdown/markdown-extensions-by-bookdown.html.

5.3 Callout blocks

The R Markdown Cookbook provides more help on how to use custom blocks to design your own callouts: https://bookdown.org/yihui/rmarkdown-cookbook/custom-blocks.html

Chapter 6

Sharing your book

6.1 Publishing

HTML books can be published online, see: https://bookdown.org/yihui/bookdown/publishing.html

6.2 404 pages

By default, users will be directed to a 404 page if they try to access a webpage that cannot be found. If you'd like to customize your 404 page instead of using the default, you may add either a `_404.Rmd` or `_404.md` file to your project root and use code and/or Markdown syntax.

6.3 Metadata for sharing

Bookdown HTML books will provide HTML metadata for social sharing on platforms like Twitter, Facebook, and LinkedIn, using information you provide in the `index.Rmd` YAML. To setup, set the `url` for your book and the path to your `cover-image` file. Your book's `title` and `description` are also used.

This `gitbook` uses the same social sharing data across all chapters in your book- all links shared will look the same.

Specify your book's source repository on GitHub using the `edit` key under the configuration options in the `_output.yml` file, which allows users to suggest an edit by linking to a chapter's source file.

Read more about the features of this output format here:

https://pkgs.rstudio.com/bookdown/reference/gitbook.html

Or use: