Machine Learning ENGINEERING

Andriy Burkov

"In theory, there is no difference between theory and practice. But in practice, there is."

— Benjamin Brewster

"The perfect project plan is possible if one first documents a list of all the unknowns."

— Bill Langley

"When you're fundraising, it's AI. When you're hiring, it's ML. When you're implementing, it's linear regression. When you're debugging, it's printf()."

— Baron Schwartz

The book is distributed on the "read first, buy later" principle.

7 Model Evaluation

Statistical models play an increasingly important role in the modern organization. When applied in a business context, a model can affect the organization's financial indicators. However, it may also present a liability risk. Therefore, any model running in production must be carefully and continuously evaluated.

Model evaluation is the fifth stage in the machine learning project life cycle:

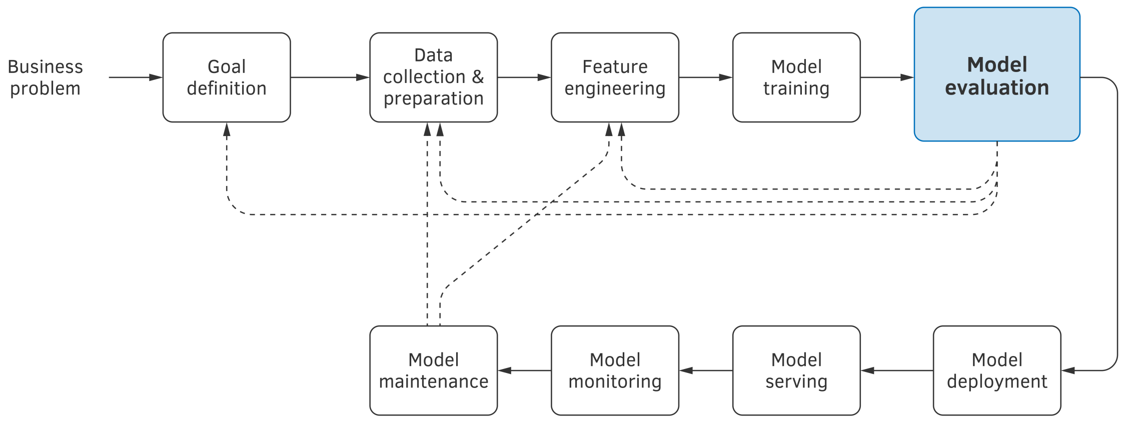

Figure 1: Machine learning project life cycle.

Depending on the model's applicative domain and the organization's business goals and constraints, model evaluation may include the following tasks:

- Estimate legal risks of putting the model in production. For example, some model predictions may indirectly communicate confidential information. Cyber attackers or competitors may attempt to reverse engineer the model's training data. Additionally, when used for prediction, some features, such as age, gender, or race, might result in the organization being considered as biased or even discriminatory.
- Study the main properties of distributions of the training data versus the production data. By comparing the statistical distribution of examples, features, and labels, in both training and production data, is how distribution shift is detected. A significant difference between the two indicates a need to update the training data, and retrain the model.
- Evaluate the performance of the model. Before the model is deployed in production, its predictive performance must be evaluated on the external data, that is, data not used for training. The external data must include both historical and online examples from the production environment. The context of evaluation on the real-time, online data should closely resemble the production environment.

- Monitor the performance of the deployed model. The model's performance may degrade over time. It is important to be able to detect this and, either upgrade the model by adding new data, or train an entirely different model. Model monitoring must be a carefully designed automated process, and might include a human in the loop. We consider this in more detail in Chapter 9.

In this chapter, we look at *some* examples of the kinds of tricks that statisticians use during the model evaluation phase. Machine learning engineering is a developing discipline, and some questions still don't have well established and easy to apply answers. In particular, the evaluation is presented from the point of view of an engineer, while each business has its own success criteria, which are unique. Before evaluating a machine learning solution, it is very important to ensure that the right people have done the most difficult work in the project: figuring out what success looks like and what are the right questions to ask in the form of business-appropriate metrics and objectives.

A common reason for failure is engineers answering convenient questions with basic tools instead of answering the right questions with custom tools — something that may require consultation with a professional statistician after your project's leaders and stakeholders have completed their part in your project. Note that some methods highlighted in this chapter, specifically those used in A/B testing (Section 7.2), are provided as examples only and might not be appropriate for your specific business problem. On important large-scale projects, it would be a mistake to try to do everything yourself. Timely collaboration with your leadership team and consulting a statistician is essential.

7.1 Offline and Online Evaluation

In Section **??**, we overviewed the evaluation techniques applied in what's called **offline model evaluation**. An offline model evaluation happens when the model is being trained by the analyst. The analyst tries out different features, models, algorithms, and hyperparameters. Tools like confusion matrix and various performance metrics, such as precision, recall, and AUC, allow comparing candidate models, and guide the model training in the right direction.

First, validation data is used to assess the chosen performance metric and compare models. Once the best model is identified, the test set is used, also in offline mode, to again assess the best model's performance. This final offline assessment guarantees post-deployment model performance. In this chapter, we talk, among other topics, about establishing statistical bounds on the offline test performance of the model.

A significant part of the chapter is devoted to the **online model evaluation**, that is, testing and comparing models in production by using online data. The difference between offline and online model evaluation, as well as the placement of each type of evaluation in a machine learning system, is illustrated in Figure 2.

In Figure 2, the historical data is first used to train a deployment candidate. Then it is evaluated offline, and, if the result is satisfactory, deployment candidate becomes the deployed

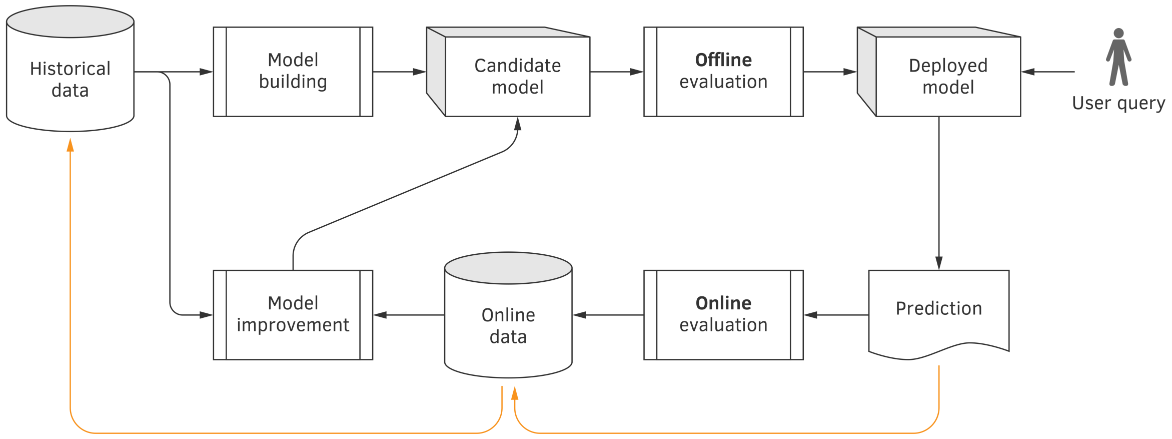

Figure 2: The placement of offline and online model evaluations in a machine learning system.

model, and starts accepting user queries. Then, user queries and the model predictions are used for an online evaluation of the model. The online data is then used to improve the model. To close the loop, the online data is permanently copied to the offline data repository.

Why do we evaluate both offline and online? The offline model evaluation reflects how well the analyst succeeded in finding the right features, learning algorithm, model, and values of hyperparameters. In other words, the offline model evaluation reflects how good the model is from an engineering standpoint.

Online evaluation, on the other hand, focuses on measuring business outcomes, such as customer satisfaction, average online time, open rate, and click-through rate. This information may not be reflected in historical data, but it's what the business really cares about. Furthermore, offline evaluation doesn't allow us to test the model in some conditions that can be observed online, such as connection and data loss, and call delays.

The performance results obtained on the historical data will hold after deployment only if the distribution of the data remains the same over time. In practice, however, it's not always the case. Typical examples of a **distribution shift** include the ever-changing interests of the user of a mobile or online application, instability of financial markets, climate change, or wear of a mechanical system whose properties the model is intended to predict.

As a consequence, the model must be continuously monitored once deployed in production. When a distribution shift happens, the model must be updated with new data and re-deployed. One way of doing such monitoring is to compare the performance of the model on online and historical data. If the performance on online data becomes significantly worse, as compared to historical, it's time to retrain the model.

There are different forms of online evaluation, each serving a different purpose. For example, runtime monitoring is checking whether the running system meets the runtime requirements.

Another common scenario is to monitor user behavior in response to different versions of the model. One popular technique used in this scenario is A/B testing. We split the users of a system into two groups, A and B. The two groups are served the old and the new models, respectively. Then we apply a statistical significance test to decide whether the performance of the new model is better than the old one.

Multi-armed bandit (MAB) is another popular technique of online model evaluation. Similar to A/B testing, it identifies the best performing models by exposing model candidates to a fraction of users. Then it gradually exposes the best model to more users, by keeping gathering performance statistics until it's reliable.

7.2 A/B Testing

A/B testing is one of the most frequently used statistical techniques. When applied to online model evaluation, it allows us to answer such questions as, "Does the new model m_B work better in production than the existing model m_A?" or, "Which of the two model candidates works better in production?"

A/B testing is often used on websites and mobile applications to test whether a specific change in the design or wording positively affects business metrics such as user engagement, click-through rate, or sales rate.

Imagine we want to decide whether to replace an existing (old) model in production with a new model. The live traffic that contains input data for the model is split into two disjoint groups: A (control) and B (experiment). Group A traffic is routed to the old model, while group B traffic is routed to the new model.

By comparing the performance of the two models, a decision is made about whether the new model performs better than the old model. The performance is compared using **statistical hypothesis testing**.

In general, statistical hypothesis testing maintains a **null hypothesis** and an **alternative hypothesis**. An A/B test is usually formulated to answer the following question: "Does the new model lead to a statistically significant change in this specific business metric?" The null hypothesis states that the new model doesn't change the average value of the business metric. The alternative hypothesis states that the new model changes the average value of the metric.

A/B test is not one test, but a family of tests. Depending on the business performance metric, a different statistical toolkit is used. However, the principle of splitting the users into two groups, and measuring the statistical significance of the difference in the metric values between different groups, remains the same.

The description of all formulations of A/B tests is beyond the scope of this book. Here we will consider only two formulations, but they apply to a wide range of practical situations.

7.2.1 G-Test

The first formulation of A/B test is based on the **G-test**. It is appropriate for a metric that counts the answer to a “yes” or “no” question. An advantage of the G-test is that you can ask any question, as long as only two answers are possible. Examples of questions:

- Whether the user bought the recommended article?
- Whether the user has spent more than \$50 during a month?
- Whether the user renewed the subscription?

Let’s see how to apply it. We want to decide whether the new model works better than the old one. To do that, we formulate a yes-or-no question that defines our metric. Then we randomly divide the users into groups A and B. The users of group A are routed to the environment running the old model, while the group’s B traffic is routed to the new model. Observe the actions of each user and record the answer as “yes” or “no.” Fill the following table:

	Yes	No	
A	$\hat{a}_{yes}$	$\hat{a}_{no}$	n_a
B	$\hat{b}_{yes}$	$\hat{b}_{no}$	n_b
	n_{yes}	n_{no}	n_{total}

Figure 3: The counts of answers to the yes-or-no question by users from groups A and B.

In the above table, $\hat{a}_{yes}$ is the number of users in group A, for which the answer to the question is “yes,” $\hat{b}_{yes}$ is the number of users in group B, for which the answer to the question is “yes,” $\hat{a}_{no}$ is the number of users in group A, for which the answer to the question is “no,” and so on. Similarly, $n_{yes} = \hat{a}_{yes} + \hat{b}_{yes}$, $n_{no} = \hat{a}_{no} + \hat{b}_{no}$, $n_a = \hat{a}_{yes} + \hat{a}_{no}$, $n_b = \hat{b}_{yes} + \hat{b}_{no}$, and, finally $n_{total} = n_{yes} + n_{no} = n_a + n_b$.

Now, find the expected numbers of “yes” and “no” answers for A and B, i.e. the number of “yes” and “no” we would get if versions A and B were equivalent.

$$\begin{aligned}
a_{yes} &\overset{\text{def}}{=} n_a \frac{n_{yes}}{n_{total}}, \\
a_{no} &\overset{\text{def}}{=} n_a \frac{n_{no}}{n_{total}}, \\
b_{yes} &\overset{\text{def}}{=} n_b \frac{n_{yes}}{n_{total}}, \\
b_{no} &\overset{\text{def}}{=} n_b \frac{n_{no}}{n_{total}}.
\end{aligned} \tag{1}$$

Now, find the value of the G-test as,[1]

$$G \stackrel{\text{def}}{=} 2\left(\hat{a}_{yes}\ln\left(\frac{\hat{a}_{yes}}{a_{yes}}\right)+\hat{a}_{no}\ln\left(\frac{\hat{a}_{no}}{a_{no}}\right)+\hat{b}_{yes}\ln\left(\frac{\hat{b}_{yes}}{b_{yes}}\right)+\hat{b}_{no}\ln\left(\frac{\hat{b}_{no}}{b_{no}}\right)\right).$$

G is a measure of how different the samples from A and B are. Statistically speaking, under the null hypothesis (A and B are equal), G follows a **chi-square distribution** with one degree of freedom:

$$G \sim \chi^2_1$$

In other words, if A and B were equal, we expect G to be small. A large value of G would make us suspicious that one of the models performs better than the other. For example, imagine you calculated $G = 3.84$. If A and B were equal (i.e. under the null hypothesis) the probability of observing $G \geq 3.84$ is about 5%. We often refer to this probability as the p-value.

If the p-value is small enough (e.g., below 0.05) then the performances of the new and the old model are very likely different (the null hypothesis is rejected). In this case, if b_{yes} is higher than a_{yes}, then the new model is very likely to work better than the old model; otherwise, the old model is better.

If the p-value corresponding to the value of G is not small enough then the observed difference of performance between the new and the old model is not statistically significant, and you can keep the old model in production.

It is convenient to find the p-value of the G-test using a programming language of your choice. In Python, it can be done in the following way:

```python
from scipy.stats import chi2
def get_p_value(G):
    p_value = 1 - chi2.cdf(G, 1)
    return p_value
```

The following code will work for R:

```r
get_p_value <- function(G) {
  p_value <- pchisq(G, df=1, lower.tail=FALSE)
  return(p_value)
}
```

Statistically, the result of the G-test is valid if we have at least 10 “yes” and “no” results in each of the two groups, though this estimate should be taken with a grain of salt. If testing is not too expensive, then having about 1000 “yes” and “no” results in each of the two groups,

[1]More details on the derivation of the above formula can be found in a statistics textbook or on Wikipedia.

with at least 100 answers of each type in each group, should be enough. Note that the total number of answers in the two groups can be different.

If you can't reach at least 100 answers of each type in each group at a reasonable cost, you can use an approximation of the p-value of a very similar test using **Monte-Carlo simulation**.

The following code will work for R:

```
p_value <- chisq.test(x,
             simulate.p.value = TRUE)$p.value
}
```

Where x is the 2×2 contingency table shown in Figure 3.

Note that it is possible to test more than two models (e.g. models A, B, and C) and more than two possible answers to the question that define our metric (e.g., "yes," "no," "maybe"). If we want to test k different models and l different possible answers, the G statistic would follow a chi-square distribution with $(k-1) \times (l-1)$ degrees of freedom. The problem here is that a test with multiple models and answers will tell you whether there is something different somewhere between your models, but it will not tell you where is the difference. In practice, it is easier to compare your current model with only one new model and to formulate a question metric with a binary answer. More complex experiment testing is outside the scope of this book.

Note that it could be tempting, when we have more than two models, to do binary comparisons of pairs of models using a test designed for comparing two models. This is not recommended, however, as it could be scientifically wrong. It's better to consult a statistician.

7.2.2 Z-Test

The second formulation of A/B test applies when the question for each user is, "How many?" or, "How much?" (as opposed to a yes-or-no question considered in the previous subsection). Examples of questions include:

1. How much time a user has spent on the website during a session?
2. How much money a user has spent during a month?
3. How many news articles a user has read during a week?

For simplicity of illustration, let's measure the time a user spends on a website where our model is deployed. As usual, users are routed to versions A and B of the website, where version A serves the old model and version B serves the new model. The null hypothesis is that users of both versions spend, on average, the same amount of time. The alternative hypothesis is that they spend more time on website B than on website A. Let n_A be the number of users routed to version A and n_B be the number of users routed to version B. Let i and j denote users from groups A and B respectively.

To compute the value of the Z-test, we first compute sample mean and sample variance for A and B. The sample mean is given by:

$$\begin{aligned}\hat{\mu}_A &\stackrel{\text{def}}{=} \frac{1}{n_A}\sum_{i=1}^{n_A} a_i,\\ \hat{\mu}_B &\stackrel{\text{def}}{=} \frac{1}{n_B}\sum_{j=1}^{n_A} b_j,\end{aligned} \tag{2}$$

where a_i and b_j is the time spent on the website by, respectively, users i and j.

The sample variance for A and B is given, respectively, by,

$$\begin{aligned}\hat{\sigma}_A^2 &\stackrel{\text{def}}{=} \frac{1}{n_A}\sum_{i=1}^{n_A}(\hat{\mu}_A - a_i)^2,\\ \hat{\sigma}_B^2 &\stackrel{\text{def}}{=} \frac{1}{n_B}\sum_{j=1}^{n_B}(\hat{\mu}_B - b_j)^2.\end{aligned} \tag{3}$$

The value of the Z-test is then given by,

$$Z \stackrel{\text{def}}{=} \frac{\hat{\mu}_B - \hat{\mu}_A}{\sqrt{\frac{\hat{\sigma}_B^2}{n_B} + \frac{\hat{\sigma}_A^2}{n_A}}}.$$

The larger Z, the more likely the difference between A and B is significant. Under the null hypothesis (i.e. A and B are equivalent), Z approximately follows a standardized normal distribution,

$$Z \approx \mathcal{N}(0,1)$$

This is true only if the sample size is large and if $\sigma_A^2 \approx \sigma_B^2$. If not, it is recommended to ask advice from a statistician.

As for the G-test, we will use the p-value to decide whether or not Z is large enough to think that the time spent on B is really greater than time spent on A. To compute the p-value, you check the probability of getting a Z-value from this distribution that is at least as extreme (out of line with the null hypothesis) as the Z-value you calculated. For example, let's imagine your sample gave you $Z = 2.64$. If A and B were equal, the probability of observing $Z \geq 2.64$ is about 5%.

To see the result of the test, you compare the p-value with the significance level you chose. If your significance level is 5%, then if the p-value is below 0.05, we reject the null hypothesis that says that the difference in performance of the two models is not statistically significant. Thus, the new model works better than the old one.

If the p-value is above or equal to 0.05, then we do not reject the null hypothesis. Note that this is not the same as accepting the null hypothesis. The two models could still be different, we just didn't get evidence in support of that. In this case, we will stick with the old model unless evidence changes our mind. No evidence means we keep doing what we were doing. Note also that we cannot simply keep gathering evidence until the p-value goes below 0.05, as it would not be scientifically sound. It's recommended to consult a statistician and design a different test.

As for significance levels, there's no universal consensus on which threshold is optimal. The values of 0.05 or 0.01 are commonly used in practice. They were favorites of a trendsetting statistician Ronald Fisher in the 1920s. You should select a higher or a lower value if it's appropriate for your application. The lower the value, the more evidence it takes to change your mind.

Similar to the G-test, it is convenient to find the p-value of the Z-test using a programming language. In Python, it can be done in the following way:

```python
from scipy.stats import norm
def get_p_value(Z):
    p_value = norm.sf(Z)
    return p_value
```

The following code will work for R:

```r
get_p_value <- function(Z) {
  p_value <- 1-pnorm(Z)
  return(p_value)
}
```

For best results, it is recommended to set n_A and n_B to a value 1000 or higher.

7.2.3 Concluding Remarks and Warnings

As mentioned in the beginning of this chapter, some methods highlighted in this chapter are provided as examples only and might not be appropriate for your specific business problem. In particular, the two statistical tests presented above are taught in schools and are indeed often used in practice, but, unfortunately, not all of those uses are appropriate for your business problem. While pointing this out, Cassie Kozyrkov, the Chief Decision Scientist at Google and one of the reviewers of this chapter, emphasized that the above two tests are rarely a good idea to apply in practice because they only show that two models are different, but they don't show whether the difference is "of at least x." If replacing the old model with the new one has a significant cost or poses a risk, then just knowing that the new model is "somewhat" better is not enough to make a replacement decision. In this case, an adjusted test must be specifically crafted for the problem at hand, and the best way to do is to consult a statistician.[2]

Carefully test the programming code of your A/B test. You will only have a valid model evaluation if you implemented everything right. Otherwise, you will not know that something is wrong: your test will not reveal that it's broken.

Also, make sure to apply measurements in groups A and B at the same time. Remember that traffic on a website behaves differently at different times of the day, or at different days of the week. For the purity of the experiment, avoid comparing measurements from different times. The same reasoning applies to other possible measurable parameters that might significantly affect user behavior, such as country of residence, speed of internet connection, or version of web browser.

7.3 Multi-Armed Bandit

A more advanced, and often preferable way of online model evaluation and selection, is **multi-armed bandit** (MAB). A/B testing has one major drawback. The number of test results in groups A and B you need to calculate the value of the A/B test is high. A significant portion of users routed to a suboptimal model would experience suboptimal behavior for a long time.

Ideally, we would like to expose a user to a suboptimal model as few times as possible. At the same time, we need to expose users to each of the two models a number of times sufficient to get reliable estimates of both models' performance. This is known as the **exploration-exploitation dilemma**: on one hand, we want to explore the models' performance enough to be able to reliably choose the better one. On the other hand, we want to exploit the performance of the better model as much as possible.

In probability theory, the multi-armed bandit problem is a problem in which a fixed and limited set of resources must be allocated between competing choices in a way that maximizes the expected reward. Each choice's properties are only partially known at the time of allocation, and may become better understood as time passes and we allocate resources to the choice.

Let's see how the multi-armed bandit problem applies to an online evaluation of two models. (The approach for more than two models is the same.)

The limited set of resources we have are the users of our system. The competing choices, also called "arms," are our models. We can allocate a resource to a choice (in other words, we can "play an arm") by routing a user to a version of the system running a specific model. We want to maximize the expected reward, where the reward is given by the business performance metric. Examples might be the average amount of time spent on the website during a session, the average number of news articles read during a week, the percentage of users who purchased the recommended article, etc.

[2] Unfortunately, in a compact book describing all special cases and tests would be impractical. Please consult the book's companion wiki from time to time. More statistical tests will be added over time.

UCB1 (for Upper Confidence Bound) is a popular algorithm for solving the multi-armed bandit problem. The algorithm dynamically chooses an arm, based on the performance of that arm in the past, and how much the algorithm knows about it. In other words, UCB1 routes the user to the best performing model more often when its confidence about the model performance is high. Otherwise, UCB1 might route the user to a suboptimal model so as to get a more confident estimate of that model's performance. Once the algorithm is confident enough about the performance of each model, it almost always routes users to the best performing model.

The mathematics of UCB1 works as follows. Let c_a denote the number of times the arm a was played since the beginning, and let v_a denote the average reward obtained from playing that arm. The reward corresponds to the value of the business performance metric. For the purpose of illustration, let the metric be the average time spent by the user in the system during one session. The reward for playing an arm is, thus, a particular session duration.

In the beginning, c_a and v_a are zero for all arms, $a = 1, \ldots, M$. Once an arm a is played, a reward r is observed, and c_a is incremented by 1; v_a is then updated as follows:

$$v_a \leftarrow \frac{c_a - 1}{c_a} \times v_a + \frac{r}{c_a}.$$

At each time step (that is, when a new user logs in), the arm to play (that is, the version of the system the user will be routed to) is chosen as follows. If $c_a = 0$ for some arm a, then this arm is played; otherwise, the arm with the greatest UCB value is played. The UCB value of an arm a, denoted as u_a, is defined as follows:

$$u_a \overset{\text{def}}{=} v_a + \sqrt{\frac{2 \times \log(c)}{c_a}}, \text{ where } c \overset{\text{def}}{=} \sum_{a}^{M} c_a.$$

The algorithm is proven to converge to the optimal solution. That is, UCB1 will end up playing the best performing arm most of the time.

In Python, the code that implements UCB1, would look as follows:

```python
class UCB1():
    def __init__(self, n_arms):
        self.c = [0]*n_arms
        self.v = [0.0]*n_arms
        self.M = n_arms
        return

    def select_arm(self):
        for a in range(self.M):
            if self.c[a] == 0:
                return a
        u = [0.0]*self.M
        c = sum(self.c)
        for a in range(self.M):
            bonus = math.sqrt((2 * math.log(c)) / float(self.c[a]))
            u[a] = self.v[a] + bonus
        return u.index(max(u))

    def update(self, a, r):
        self.c[a] += 1
        v_a = ((self.c[a] - 1) / float(self.c[a])) * self.v[a] \
                + (r / float(self.c[a]))
        self.v[a] = v_a
        return
```

The corresponding code in R would look as shown below:

```
setClass("UCB1", representation(count="numeric", value="numeric", M="numeric"))

setGeneric("select_arm", function(x) standardGeneric("select_arm"))
setMethod("select_arm", "UCB1", function(x) {
    for (a in seq(from = 1, to = x@M, by = 1)) {
        if(x@count[a] == 0) {
            return(a)
        }
    }
    u <- rep(0.0, x@M)
    count <- sum(x@count)
    for (a in seq(from = 1, to = x@M, by = 1)){
        print(a)
        bonus <- sqrt((2 * log(count)) / x@count[a])
        u[a] <- x@value[a] + bonus
    }
    match(c(max(u)),u)
})

setGeneric("update", function(x, a, r) standardGeneric("update"))
setMethod("update", "UCB1", function(x, a, r) {
    x@count[a] <- x@count[a] + 1
    v_a <- ((x@count[a] - 1) / x@count[a]) * x@value[a] + (r / x@count[a])
    x@value[a] <- v_a
})

UCB1 <- function(M) {
    new("UCB1", count = rep(0, M), value = rep(0.0, M), M = M)
}
```

7.4 Statistical Bounds on the Model Performance

When reporting the model performance, sometimes, besides the value of the metric, it is required to also provide the statistical bounds, also known as the **statistical interval**.

A reader familiar with other books on machine learning or some popular online blogs might wonder, why we use the term "statistical interval" and not "confidence interval." The reason is that in some machine learning literature, what the authors call a "confidence interval" is in fact a "credible interval." The difference between the two is clear and important for a statistician because the two terms have different meanings in frequentist and Bayesian statistics. In this book, I decided not to burden the reader with the subtlety of the difference between the two terms. For a non-expert in statistics, it would be beneficial to think of the statistical interval as follows: a 95% statistical interval indicates that there's a 95% chance the parameter you're estimating is between the intervals bounds. Strictly speaking, this is the definition of the credible interval. A confidence interval's interpretation is subtly different, most newcomers to statistics won't start to appreciate the difference until they're a few textbooks in. For our purposes, the above interpretation of a statistical interval will suffice.

There are several techniques that allow establishing statistical bounds for a model. Some techniques apply to classification models, and some can be applied to regression models. We will consider several techniques in this section.

7.4.1 Statistical Interval for the Classification Error

If you report the error ratio "err" for a classification model (where $\text{err} \overset{\text{def}}{=} 1 - \text{accuracy}$), then the following technique can be used to obtain the statistical interval for "err."

Let N be the size of the test set. Then, with probability 99%, "err" lies in the interval,

$$[\text{err} - \delta, \text{err} + \delta],$$

where $\delta \overset{\text{def}}{=} z_N \sqrt{\frac{\text{err}(1-\text{err})}{N}}$, and $z_N = 2.58$.

The value of z_N depends on the required **confidence level**. For the confidence level of 99%, $z_N = 2.58$. For other confidence level values, the values of z_N can be found in the table below:

confidence level	80%	90%	95%	98%	99%
z_N	1.28	1.64	1.96	2.33	2.58

As with p-values, it is convenient to find the value of z_N using a programming language. In Python, it can be done in the following way:

```python
from scipy.stats import norm
def get_z_N(confidence_level): # a value in (0,100)
```

```
    z_N = norm.ppf(1-0.5*(1 - confidence_level/100.0))
    return z_N
```

The following code will work for R:

```
get_z_N <- function(confidence_level) {# a value in (0,100)
  z_N <- qnorm(1-0.5*(1 - confidence_level/100.0))
  return(z_N)
}
```

In theory, the above technique works even for very tiny test sets with $N \geq 30$. However, a more accurate rule of thumb for obtaining the minimum size N of the test set is as follows: find the value of N such that $N \times \text{err}(1 - \text{err}) \geq 5$. Intuitively, the greater the size of the test set, the lower our uncertainty about the true performance of the model.

7.4.2 Bootstrapping Statistical Interval

A popular technique for reporting the statistical interval for any metric, and which applies to both classification and regression, is based on the idea of **bootstrapping**. Bootstrapping is a statistical procedure that consists of building B samples of a dataset, and then training a model or computing some statistic using those B samples. In particular, the **random forest** learning algorithm is based on this idea.

Here's how bootstrapping applies for building a statistical interval for a metric. Given the test set, we create B random samples $\mathcal{S}_b$, one for each $b = 1, \ldots, B$. To obtain a sample $\mathcal{S}_b$ for some b, we use **sampling with replacement**. Sampling with replacement means that we start with an empty set, and then pick at random an example from the test set and put its exact copy in $\mathcal{S}_b$ by keeping the original example in the test set. We keep picking examples at random and putting them to $\mathcal{S}_b$ until $|\mathcal{S}_b| = N$.

Once we have B bootstrap samples of the test set, we compute the value of the performance metric m_b using each sample $\mathcal{S}_b$ as the test set. Sort the B values in ascending order. Then find the value S of the sum of all B values of the metric: $S \stackrel{\text{def}}{=} \sum_{b=1}^{B} m_b$. To obtain a c percent statistical interval for the metric, pick the tightest interval between a minimum a and a maximum b such that the sum of the values m_b that lie in that interval accounts for at least c percent of S. Our statistical interval is then given by $[a, b]$.

The above paragraph might sound vague, so let's illustrate it with an example. Let's have $B = 10$. Let the values of the metric, computed by applying the model to B bootstrap samples, be $[9.8, 7.5, 7.9, 10.1, 9.7, 8.4, 7.1, 9.9, 7.7, 8.5]$. First, we sort those values in the increasing order: $[7.1, 7.5, 7.7, 7.9, 8.4, 8.5, 9.7, 9.8, 9.9, 10.1]$. Let our confidence level c be 80%. Then, the minimum a of the statistical interval will be 7.46 and the maximum b will be 9.92. The above two values were found using the `percentile` function in Python:

```
from numpy import percentile
def get_interval(values, confidence_level):
    # confidence_level is a value in (0,100)
    lower = percentile(values, (100.0-confidence_level)/2.0)
    upper = percentile(values, confidence_level+((100.0-confidence_level)/2.0))
    return (lower, upper)
```

The same can be done in R by using the `quantile` function:

```
get_interval <- function(values, confidence_level) {
    # confidence_level is a value in (0,100)
    cl <- confidence_level/100.0
    quant <- quantile(values, probs = c((1.0-cl)/2.0, cl+((1.0-cl)/2.0)),
    names = FALSE)
  return(quant)
}
```

Once you have the boundaries $a = 7.46$ and $b = 9.92$ of the statistical interval, you can report that the value of the metric for your model lies in the interval $[7.46, 9.92]$ with confidence 80%.

In practice, analysts use confidence levels of either 95% or 99%. The higher the confidence, the wider the interval. The number B of bootstrap samples is usually set to 100.

7.4.3 Bootstrapping Prediction Interval for Regression

Until now, we considered the statistical interval for an entire model and a given performance metric. In this section, we will use bootstrapping to compute the **prediction interval** for a regression model and a given feature vector $\mathbf{x}$, which this model receives as input.

We want to answer the following question. Given a regression model f and an input feature vector $\mathbf{x}$, what is an interval of values $[f_{min}(\mathbf{x}), f_{max}(\mathbf{x})]$ such that the prediction $f(\mathbf{x})$ lies inside that interval with confidence c percent?

The bootstrapping procedure here is similar. The only difference is that now we build B bootstrap samples of the training set (and not the test set). By using B bootstrap samples as B training sets, we build B regression models, one per bootstrap sample. Let the input feature vector be $\mathbf{x}$. Fix a confidence level c. Apply B models to $\mathbf{x}$ and obtain B predictions. Now, by using the same technique as above, find the tightest interval between a minimum a and a maximum b such that the sum of the values of predictions that lie in the interval accounts for at least c percent of the sum of B predictions. Then return the prediction $f(\mathbf{x})$, and state that, with confidence c percent, it lies in the interval $[a, b]$.

As previously, the confidence level usually is either 95% or 99%. The number B of bootstrap samples is set to 100 (or as many as the time allows).

7.5 Evaluation of Test Set Adequacy

In traditional software engineering, tests are used to identify defects in the software. The collection of tests is constructed in such a way that they allow discovery of bugs in the code before the software reaches production. The same approach applies to the testing of all the code developed "around" the statistical model: the code that gets the input from the user, transforms it into features, and the code that interprets the outputs of the model, and serves the result to the user.

However, an additional evaluation must be applied to the model itself. The test examples used to evaluate the model must also be designed in such a way that they allow the discovery of the model's defective behavior before the model reaches production.

7.5.1 Neuron Coverage

When we evaluate a neural network, especially one to be used in a mission-critical scenario, such as a self-driving car or a space rocket, our test set must have good coverage. **Neuron coverage** of a test set for a neural network model is defined as the ratio of the units (neurons) activated by the examples from the test set, to the total number of units. A good test set has close to 100% neuron coverage.

A technique for building such a test set is to start with a set of unlabeled examples, and all units of the model uncovered. Then, iteratively, we

1) randomly pick an unlabeled example i and label it,
2) send the feature vector $\mathbf{x}_i$ to the input of the model,
3) observe which units in the model were activated by $\mathbf{x}_i$,
4) if the prediction was correct, mark those units as covered,
5) go back to step 1; continue iterating until the neuron coverage becomes close to 100%.

A unit is considered activated when its output is above a certain threshold. For ReLU, it's usually zero; for a logistic sigmoid, it's 0.5.

7.5.2 Mutation Testing

In software engineering, good test coverage for a **software under test** (SUT) can be determined using the approach known as **mutation testing**. Let's have a set of tests designed to test an SUT. We generate several "mutants" of the SUT. A mutant is a version of the SUT in which we randomly make some modifications, such as replacing in the source code, a "+" with a "−", a "<" with a ">", delete the `else` command in an `if-else` statement, and so on. Then we apply the test set to each mutant, and see if at least one test breaks on that mutant. We say that we kill a mutant if one test breaks on it. We then compute the ratio of killed mutants in the entire collection of mutants. A good test set makes this ratio equal to 100%.

In machine learning, a similar approach can be followed. However, to create a mutant statistical model, instead of modifying the code, we modify the training data. If the model is deep, we can also randomly remove or add a layer, or remove or replace an activation function. The training data can be modified by,

- adding duplicated examples,
- falsifying the labels of some examples,
- removing some examples, or
- adding random noise to the values of some features.

We say that we kill a mutant if at least one test example gets a wrong prediction by that mutant statistical model.

7.6 Evaluation of Model Properties

When we measure the quality of the model according to some performance metric, such as accuracy or AUC, we evaluate its **correctness** property. Besides this commonly evaluated property of the model, it can be appropriate to evaluate other properties of the model, such as robustness and fairness.

7.6.1 Robustness

The **robustness** of a machine learning model refers to the stability of the model performance after adding some noise to the input data. A robust model would exhibit the following behavior. If the input example is perturbed by adding random noise, the performance of the model would degrade proportionally to the level of noise.

Consider an input feature vector $\mathbf{x}$. Let us, before applying a model f to that input example, modify the values of some features, chosen randomly, by replacing them with a zero, to obtain a modified input $\mathbf{x}'$. Continue randomly choosing and replacing values of features in $\mathbf{x}$, as long as the **Euclidean distance** between $\mathbf{x}$ and $\mathbf{x}'$ remains below some δ. Then apply the model f to $\mathbf{x}$ and $\mathbf{x}'$ to obtain predictions $f(\mathbf{x})$ and $f(\mathbf{x}')$. Fix values of δ and ϵ. The model f is said to be ϵ-robust to a δ-perturbation of the input, if, for any $\mathbf{x}$ and $\mathbf{x}'$, such that $\|\mathbf{x} - \mathbf{x}'\| \leq \delta$, we have $|f(\mathbf{x}) - f(\mathbf{x}')| \leq \epsilon$.

If you have several models that perform similarly according to the performance metric, you would prefer to deploy in production a model that is ϵ-robust, when applied to the test data, with the smallest ϵ. However, in practice, it's not always clear how to set the appropriate value of δ. A more practical way to identify a robust model among several candidates is as follows.

Let us say that a certain test set is δ-perturbed if we obtained it by applying a δ-perturbation to all examples in a certain original test set. Pick the model f you want tested for robustness. Set a reasonable value of $\hat{\epsilon}$ such that, if the model prediction in production is not farther from the correct prediction than $\hat{\epsilon}$, you would consider that acceptable. Start with a small value of

δ and build a δ-perturbed dataset. Find the minimum ϵ such that for each example $\mathbf{x}$ from the original test set and its counterpart $\mathbf{x}'$ from the δ-perturbed test set, $|f(\mathbf{x}) - f(\mathbf{x}')| \leq \epsilon$.

If $\epsilon \geq \hat{\epsilon}$, you have chosen too high a value for δ; set a lower value and start over.

If $\epsilon < \hat{\epsilon}$, then slightly increase δ, build a new δ-perturbed test set, find ϵ for this new δ-perturbed test set, and continue increasing δ as long as ϵ remains less than $\hat{\epsilon}$. Once you find the value of $\delta = \hat{\delta}$ where $\epsilon \geq \hat{\epsilon}$, note that the model f you are testing for robustness is $\hat{\epsilon}$-robust to $\hat{\delta}$-perturbation of the input. Now pick another model you want to test for robustness, and find its $\hat{\delta}$; continue like that until all models are tested.

Once you have the value of $\hat{\delta}$-perturbation for each model, deploy in production the model whose $\hat{\delta}$ is the greatest.

7.6.2 Fairness

Machine learning algorithms tend to learn what humans are teaching them. The teaching comes in the form of training examples. Humans have biases which may affect how they collect and label data. Sometimes, bias is present in historical, cultural, or geographical data. This, in turn, as we have seen in Section **??** in Chapter 3, may lead to biased models.

The attributes that are sensitive and need protection from unfairness are called **protected** or **sensitive attributes**. Examples of legally-recognized and protected attributes include race, skin color, gender, religion, national origin, citizenship, age, pregnancy, familial status, disability status, veteran status, and genetic information.

Fairness is often domain-specific, and each domain may have its own regulations. Regulated domains include credit, education, employment, housing, and public accommodation.

The definition of fairness varies greatly, depending on the domain. At the time of writing this book, there is no firm consensus, in the scientific and technical literature, on what is fairness. Most commonly cited concepts are demographic parity and equal opportunity.

Demographic parity (also known as **statistical parity**, or **independence parity**) means the proportion of each segment of a protected attribute receives a positive prediction from the model at equal rates.

Let a positive prediction mean "acceptance to university," or "granting a loan." Mathematically, demographic parity is defined as follows. Let G_1 and G_2 be the two disjoint groups belonging to the test data, divided by a sensitive attribute j, such as gender. Let $\mathbf{x}^{(j)} = 1$ if $\mathbf{x}$ represents a woman, and $\mathbf{x}^{(j)} = 0$ otherwise. A binary model f under test satisfies demographic parity if $\Pr(f(\mathbf{x}_i) = 1 | \mathbf{x}_i \in G_1) = \Pr(f(\mathbf{x}_k) = 1 | \mathbf{x}_k \in G_2)$. That is, as measured on the test data, the chance to predict 1 with the model f for women is the same as the chance to predict 1 for men.

The exclusion of the protected attributes from the feature vector in the training data doesn't guarantee that the model will have demographic parity, as some of the remaining features may be **correlated** with the excluded ones.

Equal opportunity means each group gets a positive prediction from the model at equal rates, assuming that people in this group qualify for it.

Mathematically, a binary model f under test satisfies equal opportunity if $\Pr(f(\mathbf{x}_i) = 1 | \mathbf{x}_i \in G_1 \text{ and } y_i = 1) = \Pr(f(\mathbf{x}_k) = 1 | \mathbf{x}_k \in G_2 \text{ and } y_k = 1)$, where y_i and y_k are the actual labels of the feature vectors $\mathbf{x}_i$ and $\mathbf{x}_k$, respectively. The above equality means that, as measured on the test data, the chance to predict 1 by the model f for women who qualify for that prediction is the same as the chance to predict 1 for men who also qualify. In the terms of the **confusion matrix**, equal opportunity requires the **true positive rate** (TPR) to be equal for each value of the protected attribute.

7.7 Summary

All statistical models running in production must be carefully and continuously evaluated.

Depending on the model's applicative domain and the organization's goals and constraints, model evaluation will include the following tasks:

- estimate legal risks of putting the model in production,
- understand the main properties of the distribution of the data used to train the model,
- evaluate the performance of the model prior to deployment, and
- monitor the performance of the deployed model.

An offline model evaluation happens after the model was trained. It is based on the historical data. The online model evaluation consists of testing and comparing models in the production environment using online data.

A popular technique of online model evaluation is A/B testing. When performing A/B testing, we split users into two groups, A and B. The two groups are served the old and the new models, respectively. Then we apply a statistical significance test to decide whether the new model is statistically different from the old model.

Multi-armed bandit is another popular technique of online model evaluation. We start by randomly exposing all models to the users. Then we gradually reduce the exposure of the least-performing models until only one, the best performing model, gets served most of the time.

In addition to reporting training model performance metrics, one may also need to provide the statistical bounds known as the statistical interval.

For both classification and regression models, a statistical interval for any metric can be computed using a popular technique called bootstrapping. It is a statistical procedure that consists of building B samples of a dataset, and then training a model and computing some statistic using each of those B samples.

The test examples used to evaluate the model must allow the discovery of defective behavior before the model reaches production. Such techniques as neuron coverage and mutation testing can be used to evaluate the test set.

When the model is used in a mission-critical system, or in regulated domains (such as credit, education, employment, housing, and public accommodation) accuracy, robustness, and fairness may have to be evaluated.