Andriy Burkov

MACHINE LEARNING ENGINEERING

“An optimist sees a glass half full. A pessimist sees a glass half empty. An engineer sees a glass that is twice as big as it needs to be.”
— Unknown

“Death and taxes are unsolved engineering problems.”
— Unknown

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

6 Model Evaluation

Statistical models play an increasingly important role in a modern organization. When applied in a business context, a model can affect not just an organization's financial indicators but also be a reason for liability risk. Therefore, any statistical model that is supposed to be applied in production or already running there has to be carefully and continuously evaluated.

Model evaluation is the fourth stage of machine learning engineering in the machine learning project life cycle:

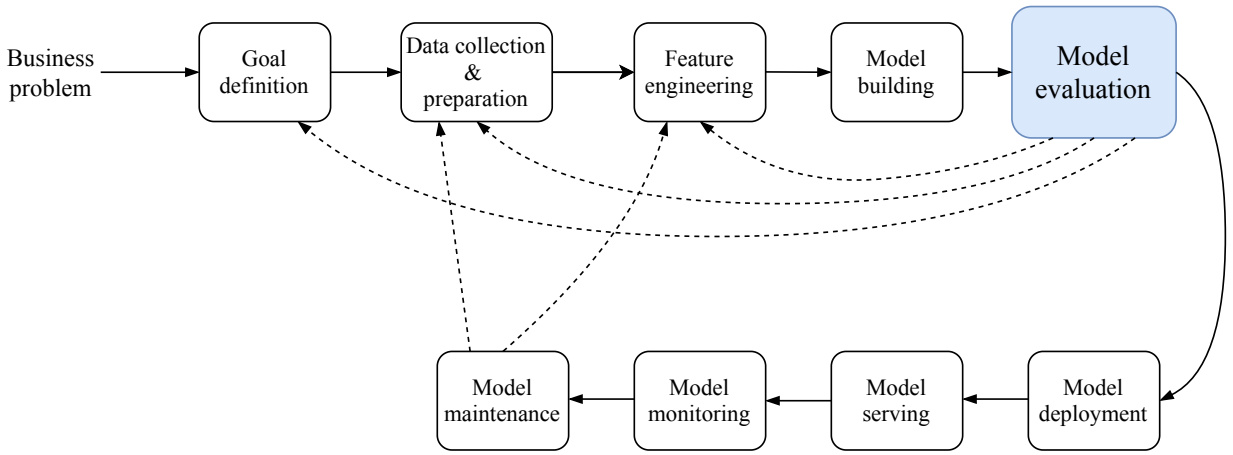

Figure 1: Machine learning project life cycle.

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

- Estimate legal risks of putting the model in production. For example, some predictions generated by the model can indirectly communicate confidential information or be used (by attackers or competitors) to reverse engineer the data used to build the model. Additionally, some features, such as age, gender or race, when used for prediction might result in the organization being considered as biased or even discriminatory.
- Study and understand the main properties of the distribution of the data used to build the model. For instance, it is important to measure the properties of the distribution of example, features and labels in order to detect the **distribution shift**. In the case when the latter is detected, it might be necessary to update the training data and retrain the model.
- Evaluate the performance of the model. Once the data analyst or the machine learning engineer finished working on the model and before the model is deployed in production, the predictive performance of the model must be evaluated on the external data, that is the data not used in the process of building the model. To better evaluate the

performance of the model, the external data must include both historical and real-time examples from the production environment. The evaluation of the model on the real-time (or online) data must happen in the environment very closely resembling the production environment.
- Monitor the performance of the deployed model. Once a model is deployed in production, typically its performance degrades over time. It is important to be able to detect performance degradation and, once that happens, either to upgrade the model by adding new data or to build an entirely different model. The latter observation means that the model monitoring has to be a carefully designed automated process that might include a human in the loop. (We consider model performance monitoring in more detail in one of the further chapters.)

In the previous chapter, I gave an overview of the techniques used to evaluate the model in what's called **offline model evaluation** setting. The offline model evaluation happens when the model is being built by the analyst in an iterative way: the analyst tries out different features, models, and hyperparameters, as we discussed in the previous chapter. Such tools for model evaluation as confusion matrix and various performance metrics, such as precision, recall, and AUC allow comparing candidate models and guide the model building in the right direction.

Once the model that performs best on the validation data according to the chosen performance metric is identified, the test set is used to assess the performance of the model in the offline mode.

In this chapter, I will talk about establishing statistical bounds on the performance of the model on the test data in the offline mode. However, a significant part of this chapter will be devoted to the so-called **online model evaluation**, that is, testing and comparing models in the production environment on online data. The difference between offline and online model evaluation is schematically shown in Figure 2, while the place of each type of model evaluation in a machine learning system is shown in Figure 3.

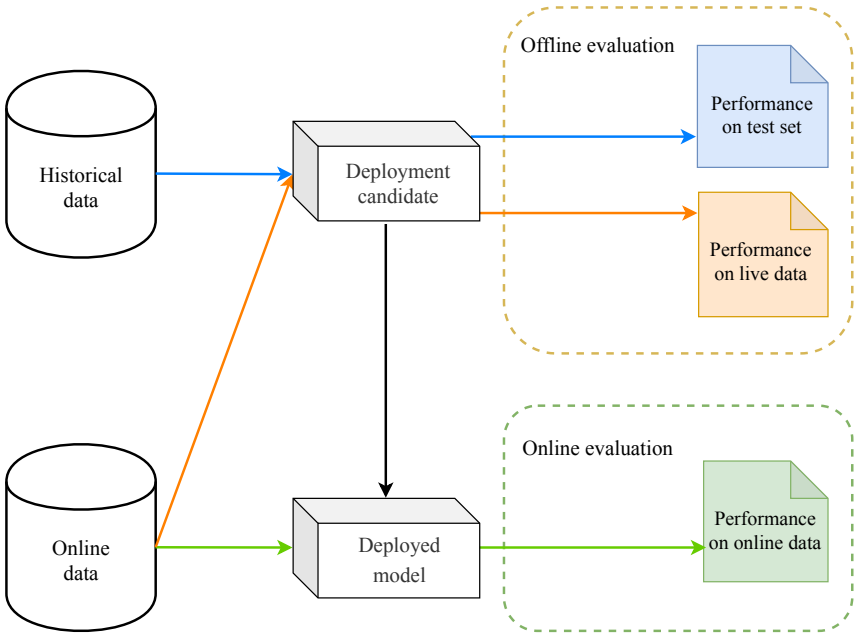

Figure 2: Offline and online model evaluation. (Adapted from Alice Zheng.)

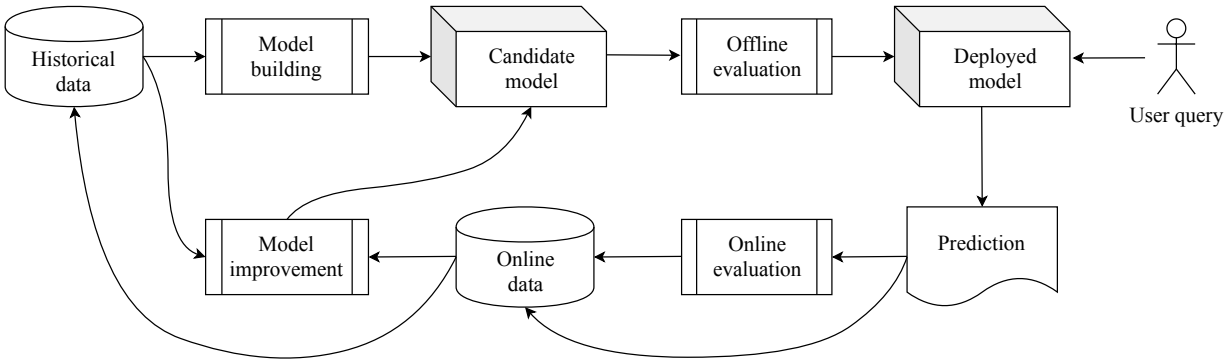

Figure 3: The place of the two types of model evaluation in a machine learning system.

In Figure 3, the model is first built using the historical data, then the deployment candidate model is evaluated offline, then, if the result of the evaluation is satisfactory, the deployment candidate becomes the deployed model and starts accepting user queries. User queries and the model predictions are used for the online evaluation of the model; the online data is then used to improve the model. To close the loop, online data is permanently copied to the offline data repository.

Why do we need to evaluate both offline and online? First of all, as we saw in the previous chapter when the analyst works on the model (in the offline setting), they usually strive to optimize one numerical metric, such as accuracy or AUC, allowing to easily compare models during the iterative model improvement process. The offline model evaluation reflects how well the analyst succeeded in finding the right features, learning algorithm, and values of hyperparameters. In other words, the offline model evaluation reflects how good the model is from a technical standpoint.

Online evaluation, on the other hand, often measures business metrics such as customer satisfaction, average online time, open rate, and click-through rate; this information may not be reflected in historical data but it is, in the end, what the business really cares about. Additionally, offline evaluation is not able to test the model in some conditions that can be observed only in real-life applied scenarios, such as data loss and call delays.

Furthermore, there are two potentially very different sources of data: historical and online. We already considered the concept of **distribution shift** in the previous chapter. As you remember, it is generally unknown whether there is a difference in the properties of the statistical distribution of the historical and the online data, and if there is a distribution shift, what type it is: covariate shift, prior probability shift or concept shift.

The performance results obtained on the historical data (the performance on the test data) will hold after the model deployment only if the distribution of data remains the same over time. In practice, however, it's often not the case: the data used to build the model and the data, whose properties this model will be used to predict in production, can change over time. Typical examples of a distribution shift include the ever-changing interests of the user of a mobile or online application, instability in financial markets, climate change, or wear of a mechanical system whose properties the model is intended to predict.

As a consequence of the distribution shift, the model, once deployed in production, has to be continuously monitored. Once the distribution shift is detected, the model has to be updated and re-deployed. One way to do such monitoring is to compare the model's performance (as given by the performance metric) on online data. If the performance degrades sufficiently as compared to the value of the metric obtained on historical data when the model was built, it's time to retrain the model. We will talk more on the model monitoring in one of the later chapters. For the moment, assume that monitoring for distribution shift, which is usually done offline by using periodical dumps of the data from the production environment, is a kind of offline evaluation.

There are different methods of conducting the online evaluation, each method serving a different purpose. For example, runtime monitoring keeps checking whether the running machine-learning-based system meets the requirements or violates some desired runtime properties. We will consider the runtime monitoring in a subsequent chapter. Another commonly used scenario is to monitor user behavior in response to different versions of the model, and based on the behavior of the user to find out whether the new model is superior to the old model. A/B testing, which we consider below, is one typical type of such online evaluation. As you will see below, when performing A/B testing on a machine-learning-

based system, the sampled users will be split into two groups using the new and old models separately. Multi-armed bandit (MAB), on the other hand, first conducts A/B testing for a short time, identifies the best model, and then dynamically exposes the chosen model to more users until a better performing model is identified.

6.1 A/B Testing

One of the most frequently used techniques for online model evaluation is **A/B testing**. It's a statistical technique that allows obtaining answers to such questions as "Whether the new model A works better in production than the existing model B?" or "Which one among the two model candidates works better in production?"

A/B testing is often used in electronic commerce on websites or mobile applications to test whether a specific change in the design positively affects business metrics such as user engagement or sales. Let us want to decide whether we have to replace an existing (old) model in production by the new model we have just built. The live traffic that contains input data for the model is split into two disjoint groups: A (control) and B (experiment). The group A traffic is routed to the old model, while the group B traffic is routed to the new model. The performance of the two models is then compared and a decision is made about whether the new model performs substantially better than the old model. Whether one model is substantially better than the other is tested by using the statistical hypothesis testing, the same tool we used in Chapter 4 to decide whether a certain feature is important for prediction or can be excluded.

In a general scenario, **statistical hypothesis testing** maintains a **null hypothesis** and an **alternate 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 stands that the new model doesn't change the average value of the business metric." The alternative hypothesis, on the other hand, stands that the new model does change the average value of the key metric.

It's important to understand that A/B test is not one test, it's a family of tests. Depending on the business performance metric, a different statistical toolkit is used. However, independently of the choice of the performance metric, the principle of splitting the users into two (or more) 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 an A/B test is beyond the scope of this book. Here I will describe only two formulations that will apply to a wide range of practical situations.

6.1.1 G-Test

The first formulation is based on the ***G*-test**. This formulation is appropriate to the metric that counts whether the answer to a business question is "yes" or "no". An advantage of this

specific formulation of A/B 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 the month?
- Whether the user renewed subscription?

As long as the results of your model's work are perceived (directly or indirectly) by the users and the business question is a yes-or-no question, this A/B test will work. Now let's see how to apply it.

Let's say we want to decide whether the new model works better than the old one. Formulate the yes-or-no question that will define your metric. Randomly divide the users of the website or software application where your model is deployed into groups A and B. The users of group A will be routed to the environment running the old model, while the group B traffic is routed to the new model. Decide whether each person's actions answer the business question either as "yes" or "no". Fill the following table:

	Yes	No	
A	a_{yes}	a_{no}	a
B	b_{yes}	b_{no}	b
	yes	no	total

Figure 4: The counts of answers the yes-or-no question by users from groups A and B.)

In the above table, a_{yes} is the number of users in group A for which the answer to the question is "yes", b_{yes} is the number of users in group B for which the answer to the question is "yes", a_{no} is the number of users in group A for which the answer to the question is "no", and so on. Similarly, $\text{yes} = a_{yes} + b_{yes}$, $\text{no} = a_{no} + b_{no}$, $a = a_{yes} + a_{no}$, $b = b_{yes} + b_{no}$, and, finally $\text{total} = \text{yes} + \text{no} = a + b$.

Now, find the expected numbers of "yes" and "no" answers for A and B:

$$\begin{aligned} \text{ea}_{yes} &\stackrel{\text{def}}{=} a\,\frac{\text{yes}}{\text{total}}, \\ \text{ea}_{no} &\stackrel{\text{def}}{=} a\,\frac{\text{no}}{\text{total}}, \\ \text{eb}_{yes} &\stackrel{\text{def}}{=} b\,\frac{\text{yes}}{\text{total}}, \\ \text{eb}_{no} &\stackrel{\text{def}}{=} b\,\frac{\text{no}}{\text{total}}. \end{aligned} \tag{1}$$

Now, find the value of the G-test as,

$$G \stackrel{\text{def}}{=} 2\left(a_{yes}\log_2\left(\frac{a_{yes}}{ea_{yes}}\right) + a_{no}\log_2\left(\frac{a_{no}}{ea_{no}}\right) + a_{yes}\log_2\left(\frac{b_{yes}}{eb_{yes}}\right) + b_{no}\log_2\left(\frac{b_{no}}{eb_{no}}\right)\right).$$

Find the highest value in the Chi-square distribution table in Figure **??** in Chapter 4 for $d.f. = 1$ such that this value is below G. Then find the corresponding p-value whose values are given in the top-most row of the Chi-square distribution table. If the p-value is small enough (below 0.01) then the real performance of the new model and the old model is very likely different (the null hypothesis is rejected). If it's the case that the performance of two models is different, then 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 is not small enough then the real performance of the new model and the old model are not different 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:

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

The following code will work for R:

```
get_p_value <- function(G_test) {
  p_value <- pchisq(2.5, 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 5000 "yes" and "no" results in each of the two groups, 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.

6.1.2 Z-Test

The second formulation applies when the question applied to 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 us 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 on average users of both versions spend the same amount of time on the website. The alternative hypothesis is that they spend a different amount of time on the website on average. Let n_A be the number of users routed to version A and n_B be the number of users routed to version B of the website. Let denote as i or j a specific user.

To compute the value of the Z-test, we must first compute the sample mean and sample variance for A and B:

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

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

If $\hat{\mu_B} \leq \hat{\mu_A}$ then it is unlikely that the new model performs better than the old one, so we stop the A/B test. Otherwise, we continue the A/B test and we want to find out whether the positive difference between $\hat{\mu_B}$ and $\hat{\mu_A}$ is statistically significant.

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,

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

By using the Z-test value, find the corresponding p-value in the standard normal distribution table (also known as Z-table) given in Table 1. To find the p-value in the Z-table, you proceed as follows. Let the value of Z-test be 2.63. To find the corresponding p-value, you select the row corresponding to 2.6 and the column corresponding to .03. The number in the Z-table is 0.9957 which gives us the p-value of $1 - 0.9957 = 0.0043$. The p-value is below 0.01. In this case, we reject the null hypothesis and, thus, the new model works better than the old one. If the p-value is above or equal to 0.01, then we accept the null hypothesis that says that the

new model is not better than the old one. The choice of 0.01 threshold is recommended, but you can select a higher or a lower value, there's no consensus which threshold is optimal.

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

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

The following code will work for R:

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

For best results, it is recommended to set n_A and n_B to a value higher than 5000.

6.1.3 Concluding Remarks

Do carefully test the programming code that implements your A/B test. You will only have a correct evaluation of your model if you did everything right. Otherwise, you will not know that something is wrong: you'll just get random answers and believe them.

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, do not try to compare people from different times.

6.2 Multi-Armed Bandit

As I already mentioned, a more advanced and often preferable way of online model evaluation and selection is to use a **multi-armed bandit** (MAB) algorithm. A/B testing has one major drawback. The number of test results in each group, A and B, needed to find the value of the A/B test is high, which means that a significant part of the users routed to the suboptimal model would experience suboptimal behavior of the product 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**: we want to explore the performance of the models enough to be able to reliably choose the best one, but we also want to exploit the performance of the best model as much as possible to reduce the negative effect of exposing the users to a suboptimal model.

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

Let's see how the multi-armed bandit problem applies to the online evaluation of two models, the old one and the new one. (The number of models we can evaluate in the multi-armed bandit problem setting can be higher than two; the approach will remain the same.)

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

One popular algorithm for solving the multi-armed bandit problem is **UCB1** (for Upper Confidence Bound). The algorithm dynamically chooses which version of the model to route the user to based on the performance of that version of the model in the past and how much the algorithm knows about that performance. In other words, the 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 to get a more confident estimate of that model's performance. With time, once the algorithm is confident enough about the performance of each model, it almost always routes users to the best performing model.

As we already said, to play an arm means to route the user to a version of the system that runs a specific version of the model. A multi-armed bandit problem can deal with multiple arms (more than two), which means that we can test multiple competing models at the same time. The 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 the arm a. As you know, the reward corresponds to the value of the business performance metric. For the purpose of illustration, let the business performance 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, $a = 1, \ldots, A$. 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:

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

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 biggest UCB value is played. The UCB value of an arm a, let's denote it as u_a is defined as follows:

$$u_a \stackrel{\text{def}}{=} v_a + \sqrt{\frac{2\log(c)}{c_a}},$$

where $c \stackrel{\text{def}}{=} \sum_a^A c_a$.

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

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

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

    def select_arm(self):
        for a in range(self.A):
            if self.c[a] == 0:
            return a
        u = [0.0]*self.A
        c = sum(self.c)
        for a in range(A):
            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", A="numeric"))

setGeneric("select_arm", function(x) standardGeneric("select_arm"))
setMethod("select_arm", "UCB1", function(x) {
    for (a in seq(from = 1, to = x@A, by = 1)) {
        if(x@count[a] == 0) {
```

```
            return(a)
        }
    }
    u <- rep(0.0, x@A)
    count <- sum(x@count)
    for (a in seq(from = 1, to = x@A, 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(A) {
    new("UCB1", count = rep(0, A), value = rep(0.0, A), A = A)
}
```

6.3 Establishing Statistical Bounds on the Model Performance

When you must report the performance of your model, sometimes it is important not only to report the value of the performance metric (obtained by applying the model to the test data) but also to provide the statistical bounds also known as the **confidence interval**. A confidence interval for a statistic is a numerical interval such that the value of the statistic lies within the interval with a high probability.

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

6.3.1 Confidence Interval for the Classification Error

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

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

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

where $\delta \stackrel{\text{def}}{=} z_N \sqrt{\frac{\text{err}(1-\text{err})}{N}}$ 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 values of confidence level, the values of z_N can be found in the below table:

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

As previously 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:

```
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) {
  z_N <- qnorm(1-0.5*(1 - 95/100.0))
  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 larger the size of the test set, the smaller should be our uncertainty about the true performance of the model.

6.3.2 Bootstrapping Confidence Interval

A frequently used technique for reporting the confidence interval for any metric that 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 using the B samples to build a model or compute some statistics. In particular, the **random forest** learning algorithm is based on the idea of bootstrapping.

Here's how bootstrapping applies to build a confidence interval for a metric. Given the test set, we create B random samples $\mathcal{S}_b$, one for each $b = 1, \ldots, B$. To sample $\mathcal{S}_b$ for some b, we do the **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 to $\mathcal{S}_b$ by keeping the original example in the original test set. We keep picking examples at random until the $|\mathcal{S}_b| = N$.

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. Then, to obtain a c percent confidence interval for the metric, we sort B values of the metric in the increasing order and then pick tightest interval between the minimum a and the maximum b such that the sum of the values of the metric that lie in the interval accounts for at least c percent of the sum of all B values of the metric.

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

```
from numpy import percentile
def get_interval(values, confidence_level): # 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) {
    cl <- confidence_level/100.0
    quant <- quantile(x, probs = c((1-cl)/2.0, cl+((1.0-cl)/2.0)), names = FALSE)
  return(quant)
}
```

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

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

6.3.3 Bootstrapping Prediction Interval for Regression

Until now, we considered the confidence 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 percents?

The bootstrapping procedure here is similar to the one we considered above. The 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, 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 all 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.

6.4 Evaluation of Test Set Adequacy

In traditional software engineering, tests are used to identify defects in the software. The set of test is constructed in such a way that they allow successfully discover 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 and the test examples used to evaluate the model also must be designed in such a way that they allow discovering defective behavior of the model before the model reaches production.

6.4.1 Neuron Coverage

When we evaluate a neural network model, especially the one to be used in a mission-critical scenario, such as a self-driving car, it is important to make sure that our test set has good coverage. **Neuron coverage** of a test set for a neural network model is defined as the ratio of the activated neurons (units) by the examples from the test set to the total number of neurons. A good test set has a 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,
2) manually label it,
3) send the feature vector of that example to the model's input,
4) see which units in the model were activated by that feature vector,
5) if the prediction was correct, mark those units as covered
6) go back to step 1 and continue iterating like that until the neuron coverage reaches the desired level (near 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.

6.4.2 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 us have a set of tests designed to test a 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 "+" by a "-", a "<" by a ">", delete the else part 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. A good test set has 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 and, if the model is deep, then 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,
- add random noise to the values of some features.

6.5 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 in some context, to evaluate other properties of the model, such as robustness and fairness.

6.5.1 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 some random noise, the performance of the model would degrade proportionally to the level of noise.

Consider an input example $\mathbf{x}$. Let us, before applying a model f to that input example, modify the values of some features, chosen randomly, in $\mathbf{x}$ by replacing them with a zero to obtain a modified input $\mathbf{x}'$. Let us randomly choose and replace 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 $\mathbf{x}'$ 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 $\|\mathbf{x} - \mathbf{x}'\|_2 \leq \delta$ 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 right value of δ. A more practical way to identify a more robust model among several candidates is as follows.

Let us say that a certain test set is δ-perturbed if we obtained that test set by applying a δ-perturbation to all examples in a certain original test set. Pick a model f you want to test for robustness. Set a reasonable value of $\hat{\epsilon}$ such that if the prediction of the model 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 a too high 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 below $\hat{\epsilon}$. Once you find such a value of $\delta = \hat{\delta}$ that $\epsilon \geq \hat{\epsilon}$, 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 $\hat{\delta}$, and 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 largest.

6.5.2 Fairness

Machine learning algorithms tend to learn what humans are teaching them. The teaching comes in the form of training data. Humans have their biases that may affect how they collect and label data, which, in turn, could lead to biased models learned from that data.

The attributes that are sensitive and need to be protected against unfairness are called **protected** or **sensitive attributes**. Examples of legally recognized 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 thus varies greatly depending on the domain and at the time of writing of this book in the scientific and technical literature, there is no firm consensus on what fairness is.

6.5.2.1 Demographic Parity

Demographic parity (also known as statistical parity or independence parity) means that the proportion of each segment of a protected attribute such as gender receives a positive prediction from the model at equal rates. Let a positive prediction means "acceptance to university", or "granting a loan".

Mathematically, demographic parity can be 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 $\Pr(f(\mathbf{x}_i) = 1 | \mathbf{x}_i \in G_1) = \Pr(f(\mathbf{x}_k) = 1 | \mathbf{x}_k \in G2)$, that is, as measured on the the test data, the chance to predict 1 by 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 can be correlated with the excluded ones.

6.5.2.2 Equal Opportunity

Equal opportunity means that 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 $\Pr(f(\mathbf{x}_i) = 1 | \mathbf{x}_i \in G_1 \text{ and } y_i = 1) = \Pr(f(\mathbf{x}_k) = 1 | \mathbf{x}_k \in G2 \text{ and } y_k = 1)$, where y_i and y_k are the actual labels of the feature vectors $\mathbf{x}_i$ and $\mathbf{x}_k$ respectivbely. The aboe euqality means that, as measured on the 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 qualify for that prediction. In the terms of the **confusion matrix**, equal opportunity requires the **true positive rate** (TPR) to be the same for each value of the protected attribute.

6.6 Contributors

I'm grateful to the following people for their valuable contributions to the quality of this chapter: Oliver Proud, Carlos Salas, Ji Hui Yang, Jonas Atarust.

Z	0	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
0	0.5000	0.5040	0.5080	0.5120	0.5160	0.5199	0.5239	0.5279	0.5319	0.5359
0.1	0.5398	0.5438	0.5478	0.5517	0.5557	0.5596	0.5636	0.5675	0.5714	0.5753
0.2	0.5793	0.5832	0.5871	0.5910	0.5948	0.5987	0.6026	0.6064	0.6103	0.6141
0.3	0.6179	0.6217	0.6255	0.6293	0.6331	0.6368	0.6406	0.6443	0.6480	0.6517
0.4	0.6554	0.6591	0.6628	0.6664	0.6700	0.6736	0.6772	0.6808	0.6844	0.6879
0.5	0.6915	0.6950	0.6985	0.7019	0.7054	0.7088	0.7123	0.7157	0.7190	0.7224
0.6	0.7257	0.7291	0.7324	0.7357	0.7389	0.7422	0.7454	0.7486	0.7517	0.7549
0.7	0.7580	0.7611	0.7642	0.7673	0.7704	0.7734	0.7764	0.7794	0.7823	0.7852
0.8	0.7881	0.7910	0.7939	0.7967	0.7995	0.8023	0.8051	0.8078	0.8106	0.8133
0.9	0.8159	0.8186	0.8212	0.8238	0.8264	0.8289	0.8315	0.8340	0.8365	0.8389
1	0.8413	0.8438	0.8461	0.8485	0.8508	0.8531	0.8554	0.8577	0.8599	0.8621
1.1	0.8643	0.8665	0.8686	0.8708	0.8729	0.8749	0.8770	0.8790	0.8810	0.8830
1.2	0.8849	0.8869	0.8888	0.8907	0.8925	0.8944	0.8962	0.8980	0.8997	0.9015
1.3	0.9032	0.9049	0.9066	0.9082	0.9099	0.9115	0.9131	0.9147	0.9162	0.9177
1.4	0.9192	0.9207	0.9222	0.9236	0.9251	0.9265	0.9279	0.9292	0.9306	0.9319
1.5	0.9332	0.9345	0.9357	0.9370	0.9382	0.9394	0.9406	0.9418	0.9429	0.9441
1.6	0.9452	0.9463	0.9474	0.9484	0.9495	0.9505	0.9515	0.9525	0.9535	0.9545
1.7	0.9554	0.9564	0.9573	0.9582	0.9591	0.9599	0.9608	0.9616	0.9625	0.9633
1.8	0.9641	0.9649	0.9656	0.9664	0.9671	0.9678	0.9686	0.9693	0.9699	0.9706
1.9	0.9713	0.9719	0.9726	0.9732	0.9738	0.9744	0.9750	0.9756	0.9761	0.9767
2	0.9772	0.9778	0.9783	0.9788	0.9793	0.9798	0.9803	0.9808	0.9812	0.9817
2.1	0.9821	0.9826	0.9830	0.9834	0.9838	0.9842	0.9846	0.9850	0.9854	0.9857
2.2	0.9861	0.9864	0.9868	0.9871	0.9875	0.9878	0.9881	0.9884	0.9887	0.9890
2.3	0.9893	0.9896	0.9898	0.9901	0.9904	0.9906	0.9909	0.9911	0.9913	0.9916
2.4	0.9918	0.9920	0.9922	0.9925	0.9927	0.9929	0.9931	0.9932	0.9934	0.9936
2.5	0.9938	0.9940	0.9941	0.9943	0.9945	0.9946	0.9948	0.9949	0.9951	0.9952
2.6	0.9953	0.9955	0.9956	0.9957	0.9959	0.9960	0.9961	0.9962	0.9963	0.9964
2.7	0.9965	0.9966	0.9967	0.9968	0.9969	0.9970	0.9971	0.9972	0.9973	0.9974
2.8	0.9974	0.9975	0.9976	0.9977	0.9977	0.9978	0.9979	0.9979	0.9980	0.9981
2.9	0.9981	0.9982	0.9982	0.9983	0.9984	0.9984	0.9985	0.9985	0.9986	0.9986
3	0.9987	0.9987	0.9987	0.9988	0.9988	0.9989	0.9989	0.9989	0.9990	0.9990

Table 1: Standard normal distribution table.