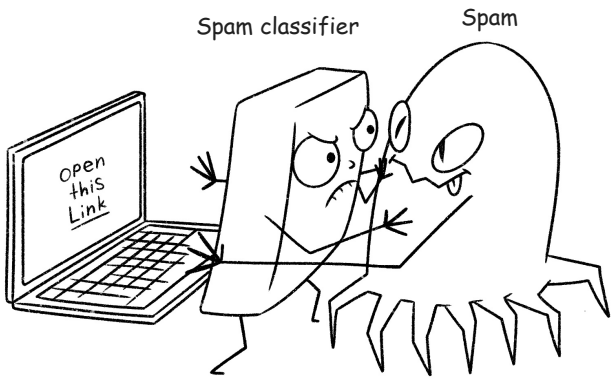
Chapter 9: Classification

Welcome to another very popular category of machine learning algorithms: classification algorithms. In the previous chapter, we explained regressions that are used for quantitative information (a sexy name for datasets with numerical relations). However, it's important to note that many real-world machine learning datasets comprise qualitative (categorical or non-numerical) and quantitative (numerical) data.

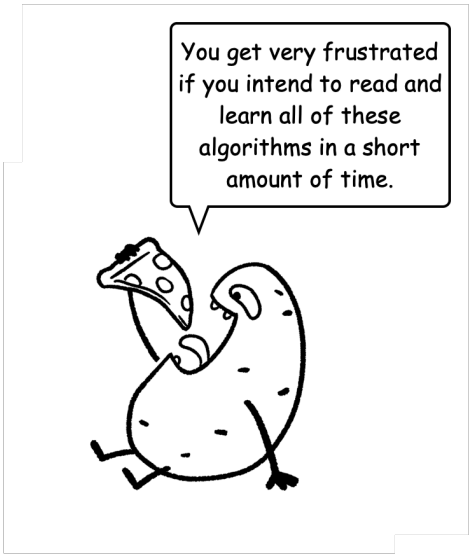
Classification algorithms are used to classify and separate the data points of a dataset based on their labels. For example, an algorithm analyzes microscopic images of a patient's kidney and recommends to the physician if the patient has kidney cancer or not. An email spam filter tries to identify spam from real emails; a social media agent tries to identify fake accounts from real users on social media, etc. Classification is not limited to binary decisions; for example, we can feed a dataset of handwritten digits into a classification algorithm, and it can recognize the digit.



We have explained in Chapter 1 that classification algorithms are supervised learning algorithms and work based on the assumption to separate the dataset into at least two subsets, a test set and a train set. First, we run a classification algorithm on the train set, which builds a model based on the given data. The model could be a set of *if-then-else* rules or mathematical equations, such as regression models. Afterward, we use the built model and run it on the test set. Then, we can evaluate the result of the classification algorithm with the assistance of the ground truth dataset (a.k.a validation, reference, or benchmark dataset). The ground truth dataset is a small part of the training set that is annotated accurately by human experts, and thus, building it is a labor-intensive and expensive process.

There are two types of classifications. One is *non-exclusive classification,* which means a data point could participate in more than one class. The other form is a *mutually exclusive classification* in which any data points are assigned to only one class.

Some algorithms that we have explained in previous chapters are classification algorithms, including logistic regression, softmax regression (Chapter 8), and linear discriminant analysis (LDA) (Chapter 7), which can also be used for classification. In this chapter, we start by describing a rule-based classifier. Then, we describe Naive Bayes, *k* Nearest Neighbor (*k*NN), Support Vector Machine (SVM), and decision tree algorithms. Afterward, we focus on ensemble learning methods, including Bagging, Boosting, Stacking, Random Forest, and AdaBoost, and then we finalize this chapter with Gradient Boosting Decision trees and their new derivations, including XGBoost, lightGBM, and Catboost.



Note that the gradient boosting algorithms of this chapter are not easy to learn; you might need to read them more than once and check other resources to learn them. We have tried our best to make them easy to understand, but keep in mind that they are complex in their nature. Besides, despite teaching this material for many semesters at Boston University, students still spot some numerical mistakes, and there might also be problems. Please let us know if you spot any numerical mistakes.

Rule-Based Classifier

Although it is the oldest classification approach, this classifier does not fit the definition of machine learning algorithms. It does really not involve any machine learning and is very straightforward computer programming. However, since we can often resolve our problem with this simple method, we respect its simplicity and list it as a classification algorithm.

A rule-based classifier is a set of IF-THEN-ELSE instructions available in all programming languages. For example, we can create a binary classification for the reader of this book, either s/he goes to heaven or hell, based on the following rules.

- IF (*does not recommend this book*) THEN (*will go hell*).
- IF (*does not pay for this book*) AND (*reads this book*) AND (*does not recommend this book*) THEN (*will go hell*).
- IF (*read this book*) THEN IF (*recommend this book*) THEN (*will go to heaven*).

The statement after IF is called *rule precondition (antecedent)*, and the statement after THEN is called *rule consequent*. Thus, we can formalize a rule as IF *antecedent* THEN *consequent.* A rule in this context has two attributes: coverage and accuracy. *Coverage* of rule *r* is written as:

$$Coverage(r) = \frac{n_{covered}}{n_{total}}$$

Here, $n_{covered}$ is the number of data points (or records if we assume the data is in tabular format) that are covered by rule *r*, and n_{total} is all data points in the dataset. Another concept is *Accuracy,* which is the number of correctly covered by the rule divided by $n_{covered}$, and it is calculated as follows:

$$Accuracy(r) = \frac{n_{correct}}{n_{covered}} = \frac{Antecenents \cap Consequnts}{Antecendents}$$

If a data point belongs to more than one class, the algorithm can use mechanisms such as voting or priority ordering (some classes have higher priorities to get a data point than others) to decide about the label of that class. Rules could be predefined by users and not learned from the training set.

One might ask why such a simple classification approach is not widely used and why we said it might not fit into the machine learning context. The motivation of machine learning is to teach an algorithm to learn on its own and decide. Rule-based classification is a sort of micromanaging of the learning process and hard coding of the rules. Therefore, it is not a good choice for a machine-learning approach.

Nevertheless, we often use rule-based classification in software development and enjoy its simplicity. For example, in a Game project, we define rules to construct the game character's mood and body shape based on predefined rules. In particular, if the user is far away from their daily steps, the character gets fat and angry. If the user passes the daily goal, the game character stays fit and happy.

Naive Bayes

One of the first theories in statistical learning (one of the oldest names of machine learning) is Bayes' theorem [Bayes '63], introduced by Thomas Bayes. It describes the probability of an event's occurrence depending on our prior knowledge about things related to the event[1]. For example, the probability of us getting peace of mind is related to helping others and not expecting any rewards, and getting obese is related to diet, genetics, etc.

In this section, we first describe the Naive Bayes theorem, and then we describe the prediction capability of the Naive Bayes theorem.

Bayes Theorem

We use the Bayes Theorem to use the probability of something we already know (prior) to predict something that is happening in the future (posterior). In other words, it combines prior knowledge (prior probability) with new evidence (likelihood) to form an updated probability (posterior probability). In the context of machine learning, we employ prior knowledge or evidence (*x*) to select the best hypothesis or proposition that predicts the future (*h*). This is the Bayes theorem and is shown in Figure 9-1.

---

[1] If you can't recall the concept of probability, before reading the rest of this algorithm, read Chapter 3.

$P(h \mid x)$ presents the *posterior* probability of hypothesis h given the dataset x.
$P(x \mid h)$ presents the *likelihood* of observing the data x given that the hypothesis h is true.
$P(h)$ presents the *prior* probability of hypothesis h being true before observing the data x.
$P(x)$ presents the *marginal* probability of the data x under all hypotheses, also known as the *evidence*.

You might recall from Chapter 3 that we discussed the conditional probability of P(A|B) will be written as follows:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \text{ or } P(B \mid A) = \frac{P(B \cap A)}{P(A)}.$$

Naive Bayes assumes that all features are independent of each other. In other words, the Naiveness of this method assumes that if we don't have a probability of two events occurring together, we can assume the probability of having them occurring together is the product of their probabilities and events being independent. In other words, we don't know the probability of A and B occurring together, and it is calculated as follows: $P(A \cap B) = P(A) \, . \, P(B)$.

Let's use an example to understand this theorem. One of your friends told you that most people who used essential oils, traditional medicine, herbal medicine, and other non-scientific methods (let's call them the xmed) did not get sick at all. You are curious to see if that theory is right or not, and let's assume you have statistical data of the people who got 'flu' and who 'don't get flu', plus people who 'use xmed' and 'don't use xmed'. We define probabilities of people who had used xmed as $P(x)$ and people who stayed healthy as $P(h)$, we also know the probability of people who have used xmed and didn't get sick, i.e., $P(h \mid x)$. Now, we would like to know if his theory is correct, i.e., the probability of people who use xmed, given they are healthy $P(x \mid h)$. Figure 9-1 visualizes the Bayesian inferences and explains how this question can be answered.


Thomas Bayes

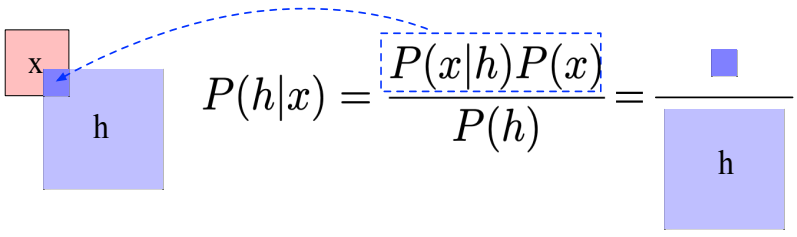


Figure 9-1: Naive Bayes equation and visualization of probabilities.

Prediction with Naive Bayes

The Naive Bayes assumes that each feature is independent of every other feature. This is a naive assumption, but it works fine in many cases. For example, the choice of lunch does not have any relation with the underwear color, i.e., Naive Bayes is correct, but the choice of lunch does have a relation to the diet, and this might negate the Naive Bayes assumption.

Recall to use the Bayesian theorem for prediction, our objective is to predict the $P(h\,|\,x)$, which is called *posterior probability*, from $P(x\,|\,h)$ which is called *likelihood*, $P(x)$ and $P(h)$ probabilities which are known and available. In other words, from the training dataset, we would like to make inferences, and this inference is called *Bayesian inference*. $P(x)$ is known, the objective is to find the maximum of $P(h\,|\,x)$, which is identical to $P(x\,|\,h) \times p(h)$.

Assume we have k number of class lables, $c_1, c_2, \ldots c_k$, and each of our data points has n number of features, i.e. $f_1, f_2, \ldots f_n$, we can use the Naive Bayes classification algorithm and assign one of the available k labels to each of the data points. Based on the Bayes theorem, it is formalized with the following equation:

$$P(c_i\,|\,f_1, f_2, \ldots f_n) = \frac{P(f_1, f_2, \ldots, f_n\,|\,c_i)\,.\,P(c_i)}{P(f_1, f_2, \ldots, f_n)}$$

With some mathematical calculations, which we do not describe, the Naive Bayes Theorem can be rewritten as follows:

$$P(c_i\,|\,f_1, f_2, \ldots f_n) = \Pi_{j=1}^{j=n} P(f_j\,|\,c_i)\,.\,P(c_i) \quad for \quad 1 \leq i \leq k$$

As an example of Naive Bayes classification, we can use it to check whether an email is spam or not. We generalize the example to your daily life because, after reading this book from the beginning to the end, you will be a superstar in data science and AI. You are receiving too many emails, and you have asked your secretary to remove all emails with the word "prestigious" and come to your email address. The word "prestigious" is very common in spam emails, but it could be a new client you want to work with. For example, an email that states, "We are a prestigious journal that publishes your scientific work," seems to come from a predatory journal and thus spam.

The traditional spam filters cannot filter your spam emails properly, and you need a more robust approach to filter emails that are asking for your journal submission (spam). Your staff analyzed some sample emails, and he ended up having the following probabilities:

$P(S)$: Probability of an email that is spam. 20 % of emails you received are spam.

$P(\neg S)$: Probability of an email that is important, and thus not spam, i.e., $100 - 20 = 80\,\%$. The symbol $\neg$ means negation in probability. For example, if A is an event, then ¬A represents the event that A does not occur.

$P(P)$: Probability of an email includes the word "prestigious".15% of your email includes the term "prestigious".

$P(P \mid \neg S)$: The probability of an email having the term "prestigious", given that it is not spam, is 8% of not spammed emails include this term.

$P(P \mid S)$: The probability of having an email that includes the word "prestigious", given that it is marked as spam. Your secretary has labeled about 12% of emails that include the term "prestigious" as spam.

$P(S \mid P)$: Probability of an email being spam, given that it has the word "prestigious" in it. This is what we are trying to find in the dataset or predict. Therefore, we can calculate $P(S \mid P)$ by using the following equation, which we described earlier:

$$P(S \mid P) = \frac{P(P \mid S) \, . \, P(S)}{P(P)} = \frac{0.12 \times 0.2}{0.15} = 0.16$$

This is a very simple example, and we have only one feature. Let's make the spam filter a bit more intelligent because your staff has realized that you still get many spam emails. He went to collect another 100 sample emails and analyze them. He realizes the term "family" in business email means no clear payment policy. For example, the person who says, *"We are like a family in the work environment."* is planning not to pay properly. Let's use $P(F)$ to present the probability of an email including the word "family". By analyzing those 100 emails manually, your staff created Table 9-1 (a), which includes the result of his analysis. If the values of this table are normalized to be between zero and one, this table is also called the likelihood table, which includes the probabilities.

a

	spam	not spam
all emails	20	80
"family"	14	4
"prestigious"	17	6

b

	spam	not spam
all emails	20	80
"family"	14	4
"prestigious"	17	6
"family" and "prestigious"	14/20 x 17/20 = 0.59	4/80 x 6/80 = 0.00375

Table 9-1: (a) describes the number of spam vs non-spam emails and the frequencies of words "family" or "prestigious" in each category. (b) By relying on Navie's assumption that events are independent, we calculate the probability of having both words by multiplying probabilities divided by the number of emails in each category.

He has forgotten to analyze the occurrences of "family" and "prestigious" terms together. Based on the naive assumption, events are independent, and thus, we can assume the following: $P(F \cap P) = P(F) \, . \, P(P)$. Here *P(F)={all email contain 'family'}/{all emails}=18/100,* and *P(P)={all email contain 'prestigious' }/{all emails}*=23/100. As these two events are independent, this formula is true. However, it is not our answer, and the desired probability is conditional probability. Therefore, we can create Table 9-1 (b) Simply by multiplying the

probability by all emails in each category to understand approximately the number of emails that include both "family" and "prestigious" and are spam, i.e., $20 \times 0.59 = 11.8 \sim 12$. The approximate number of emails that include both "family" and "prestigious" that are not spam is $80 \times 0.00375 = 0.3$ (which is close to zero).

Based on the data provided Table 9.1, we can have the following probabilities:

Probability of getting an email that is spam, $P(S) = 20/100$

Probability of getting an email that is not spam, $P(\neg S) = 80/100$

The probability of getting an email containing the word "prestigious", given it is spam, $P(P \mid S) = 17/20$

The probability of getting an email containing the word "prestigious", given it is not a spam $P(P \mid \neg S) = 6/80$

Probability of getting an email containing the word "family", given it is spam, $P(F \mid S) = 14/20$

The probability of getting an email containing the word "family", given it is not spam, $P(F \mid \neg S) = 4/80$

The probability of having a spam email, given that it includes both words "family" and "prestigious" is $P(S \mid F \cap P) = 0.59$. The sign $\cap$ or $\wedge$ stayed for conjunction (and). The results can be read from Table 9-1. Therefore, when an email including these two keywords arrives, the spam filter marks it as spam because there is a 59% probability that it is spam.

This is the *Naive* Bayes algorithm because calculating the probability for each hypothesis is simplified. However, this does not mean that predictive ability is not strong; it is a very useful algorithm for many classification cases; we will describe a classification (prediction) example shortly.

Gaussian Naive Bayes

Gaussian Naive Bayes is a type of Naive Bayes method where continuous attributes are considered, and data features have a Gaussian distribution. The email example that we have previously explained deals with discrete data. We can also use Naive Bayes to make predictions on continuous data. Assuming features are normally distributed, we can use the Gaussian Naive Bayes algorithm. If they are not normally distributed, it is recommended that outliers be removed to make them approximately normally distributed.

For example, let's assume there is another feature in spam email detection, which is the "time of email arrival"; we call it "time" for the sake of brevity. Our objective is to perform a prediction based on time. We assume time is a continuous variable; for the continuous variable, instead of probability, we use their probability distribution function (PDF), which is not the PDF from Adobe but the one we explained back in Chapter 3. Instead of estimating $P(h \mid d)$, we should estimate $P(S \mid pdf(time))$, S denotes the spam.

Naive Bayes Prediction Example

Here, we explain another prediction example in another form. This example can ensure that when we finish this section, we are experts in Naive Bayes, and the soul of Thomas Bayes is praying for the sins we have performed or will perform in our machine learning career by a more complex algorithm when Naive Bayes can answer the same thing.

Assume you have a good friend, and when you call him/her, s/he is always available to hang out with you. Having a social interaction is very important for mental health. Nevertheless, watching "TV" or "playing online games" is also delightful.

Every weekend, you check your favorite TV series webpage to see if the new episode of your favorite series, "How to Waste My Time instead of Learning AI," has arrived. Also, you open your phone and scroll the app market to see if there is a new good game to download and play. You do this every weekend, and you decide to use the Naive Bayes algorithm to predict whether you will go to meet your friend over the weekend or stay home and spend your time on games or watching TV. Table 9-2. presents what you have recorded about your behaviors over the previous weekends. Your friend would like to predict whether you will meet him/her next weekend (based on the availability of a new game or episode). Assuming the # sign will be used to mention the count of events and $\wedge$ is used to specify intersection (and), first, we calculate the conditional probabilities as follows:

tv	game	meet friend
no	no	no
no	no	yes
no	yes	no
yes	yes	no
no	yes	yes
no	yes	yes
yes	yes	no
yes	yes	no
no	no	yes
yes	no	yes

Table 9-2: Data we have collected from our previous observations.

$$P(TV = yes) : 4/10 = 0.4, \quad P(Game = yes) : 6/10 = 0.6, \quad P(Friend = yes) : 5/10 = 0.5$$

$$P(Game = yes \mid Friend = yes) = \frac{\#(Game = yes) \wedge \#(Friend = yes)}{\#(Friend = yes)} = \frac{0.2}{0.5} = 0.4$$

$$P(TV = yes \mid Friend = yes) = \frac{\#(Friend = yes) \wedge \#(TV = yes)}{\#(Friend = yes)} = \frac{0.1}{0.5} = 0.2$$

$$P(Game = no \mid Friend = yes) = \frac{\#(Game = no) \wedge \#(Friend = yes)}{\#(Friend = yes)} = \frac{0.3}{0.5} = 0.6$$

$$P(TV = no \mid Friend = yes) = \frac{\#(Friend = yes) \wedge \#(TV = no)}{\#(Friend = yes)} = \frac{0.4}{0.5} = 0.8$$

$$P(Game = yes \mid Friend = no) = \frac{\#(Game = yes) \wedge \#(Friend = no)}{\#(Friend = no)} = \frac{0.4}{0.5} = 0.8$$

$$P(TV = yes \mid Friend = no) = \frac{\#(TV = yes) \wedge \#(Friend = no)}{\#(Friend = no)} = \frac{0.3}{0.5} = 0.6$$

$$P(Game = no | Friend = no) = \frac{\#(Game = no) \wedge \#(Friend = no)}{\#(Friend = no)} = \frac{0.1}{0.5} = 0.2$$

$$P(TV = no | Friend = no) = \frac{\#(TV = no) \wedge \#(Friend = no)}{\#(Friend = no)} = \frac{0.1}{0.5} = 0.2$$

Recall that we have explained that a probability in a Naive Bayes is constructed based on the following equation: $P(c_i | f_1, f_2, \ldots f_n) = \Pi_{j=1}^{j=n} P(f_j | c_i) \, . \, P(c_i)$. Now, we can construct every combination of two other available information ("tv", "game"), disregarding the real value of the "meet friend" column.

For example, for the four different combinations of "TV" and "Game" based on Table 9-2, we can write the following:

TV = no, Game= no

$$P(meetfriend) = P(TV = no | Friend = yes) \times P(Game = no | Friend = yes) \times P(Friend = yes)$$
$$= 0.8 \times 0.6 \times 0.5 = 0.24$$
$$P(\sim meetfriend) = P(TV = no | Friend = no) \times P(Game = no | Friend = no) \times P(Friend = no)$$
$$= 0.2 \times 0.2 \times 0.5 = 0.02$$

TV= no, Game = yes

$$P(meetfriend) = P(TV = no | Friend = yes) \times P(Game = yes | Friend = yes) \times P(Friend = yes)$$
$$= 0.8 \times 0.4 \times 0.5 = 0.16$$
$$P(\sim meetfriend) = P(TV = no | Friend = no) \times P(Game = no | Friend = no) \times P(Friend = no)$$
$$= 0.2 \times 0.2 \times 0.5 = 0.02$$

TV=yes, Game=no

$$P(meetfriend) = P(TV = yes | Friend = yes) \times P(Game = no | Friend = yes) \times P(Friend = yes)$$
$$= 0.2 \times 0.6 \times 0.5 = 0.06$$
$$P(\sim meetfriend) = P(TV = yes | Friend = no) \times P(Game = no | Friend = no) \times P(Friend = no)$$
$$= 0.6 \times 0.2 \times 0.5 = 0.06$$

TV=yes, Game=yes

$$P(meetfriend) = P(TV = yes | Friend = yes) \times P(Game = yes | Friend = yes) \times P(Friend = yes)$$
$$= 0.2 \times 0.4 \times 0.5 = 0.04$$
$$P(\sim meetfriend) = P(TV = yes | Friend = no) \times P(Game = yes | Friend = no) \times P(Friend = no)$$
$$= 0.6 \times 0.8 \times 0.5 = 0.24$$

Now, with this data, we can populate Table 9-3 and compare its prediction result (fourth and fifth columns) with the actual value (third column). We can calculate accuracy, precision, recall, and other metrics related to the accuracy by comparing the actual and predicted class results.

tv	game	meet friend	Predicted (meet friend)	Predicted (¬ meet-friend)
no	no	no	0.24 x	0.02
no	no	yes	0.24 ✔	0.02
no	yes	no	0.16 x	0.02
yes	yes	no	0.04	0.24 ✔
no	yes	yes	0.16 ✔	0.02
no	yes	yes	0.16 ✔	0.02
yes	yes	no	0.04	0.24 ✔
yes	yes	no	0.04	0.24 ✔
no	no	yes	0.24 x	0.02
yes	no	yes	0.06	0.06

Table 9-3: Meet or not meet with the friend predicted. Based on the "meet friend" column, we use a tick mark to state a correct prediction and x to state an incorrect prediction. Since, in the last row, both predicted values are equal, we can not make any judgment.

The result of a Naive Bayesian algorithm will be a confusion matrix, and via the confusion matrix, the software package, or manually, we can interpret the results.

The two examples we have explained were binary classification, and we chose binary for the sake of simplicity, but keep in mind that we can use Naive Bayes for multilevel classification or prediction as well. Assuming N is the number of training data objects, f is the number of features, and c is the number of classes, the computational complexity of Naive Bayes in the training phase is *O(Nf),* and for the testing phase is *O(fc)*.

NOTES:

* Since there is no parameter being used for Naive Bayes, an optimization method is not needed to find a good fit for the parameter. Naive Bayes and similar models are referred to as parameter-free models.

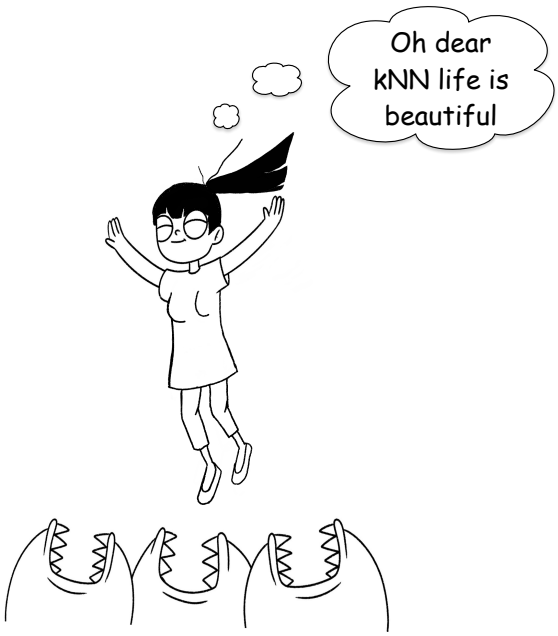
* Naive Bayesian can perform both binary and multi-class classification.

* While dealing with classification, we might encounter generative models, generative statistics, and the term generative often. In simple terms, the generative model is a model that is composed of joint probability distributions, e.g. $P(X, Y)$. Another term in discriminative models describes conditional probability, e.g. $P(Y \mid X =' something')$. Later, we will learn more about these two types of models.

k Nearest Neighbor (kNN)

The algorithms we have explained, such as regression algorithms and naive Bayes, are based on building a model that generalizes the training data to make predictions. However, other types of supervised learning algorithms do not build a generalized model; instead, they use the whole dataset for prediction. These are referred to as *instance-based learning* algorithms. In instance-based learning, predictions are made by analyzing the specific instances in the dataset that are closest to the new data point. A popular example of these algorithms is the *k*-nearest Neighbor (kNN) algorithm [Cover '67], which classifies new instances based on the majority label among the *k*-closest training examples.

Please close your eyes, take a deep breath, and think about Chapter 4, where we explained clustering algorithms. Imagine yourself surfing among clustering algorithms as a data science expert without the need to learn the mathematics we have described afterward. *k*NN is very easy to understand and operates similarly to unsupervised learning algorithms. Now, wake up; real life is waiting for you.



Upcoming Algorithms

kNN operates under the assumption that similar data points are located near each other. In supervised learning, this means starting with a dataset where each data point is labeled. kNN uses a distance function to measure the closeness between data points. For example, the Euclidean distance function, which is also common in clustering algorithms (check Chapter 4), can be used in kNN. The algorithm then labels unlabeled data points based on the labels of the closest data points. The number of these 'closest data points' to consider is specified by the hyperparameter *k*. For instance, if *k*=3, the algorithm looks at the three nearest neighbors of an unlabeled point to determine its label.

As illustrated in Figure 9-2, labeled points might be colored red and blue, while unlabeled points are grey. kNN would predict the label of the grey points based on the majority label of their three closest neighbors. This algorithm receives k as a hyperparameter, which specifies the number of neighbor data points. Assuming $k = 3$, check Figure 9-2 (a), we have a dataset in two-dimensional space, and we label a few of its data points as red and blue data points, and the grey ones in Figure 9-2 (b) do not have labels [2].

[2] We could say it is three-dimensional space because another dimension or feature is blue/red color. It is better to refer to dimensions as features because having a dimension that includes either blue or red is not wrong, but it is not common.

The kNN algorithm calculates the distance of each unlabeled data point into k nearest data points and decides on the label. For example, the algorithm chooses a data point that is marked with "?" in Figure 9-2 (b) to decide about its label (blue or red). Figure Figure 9-2 (c) shows that there are three data points close to this particular data point; two are red, and one is blue. Therefore, the majority are red, and thus, this data point will receive the red label as well (Figure 9-2 (d)). This process continues until all data points receive a label. Also, if a new data point is added to the dataset, it will be treated as another unlabeled data point.

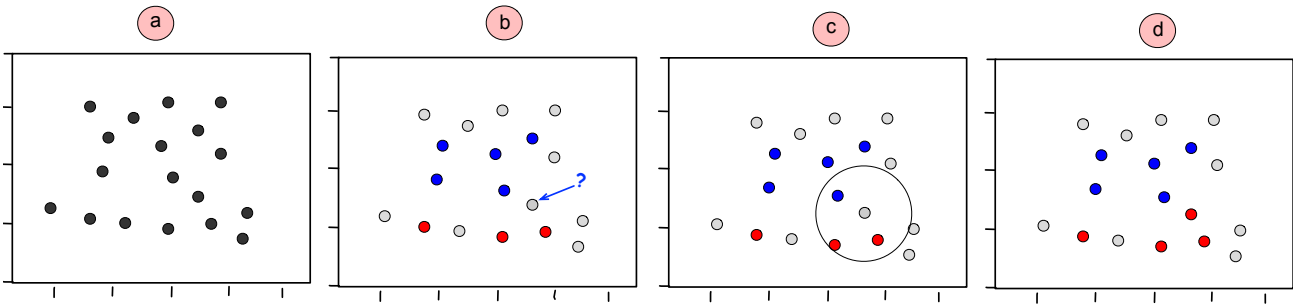


Figure 9-2: (a) The dataset without labeled data points (b) few data points in the dataset has been labeled with blue and red color. (c) a data point have been selected, we need to know what label (red or blue) does this data point receive ?

Our lovely *k*NN belongs to the category of *lazy learning* algorithms because it stores all the data points first and then uses them, which does not actively classify data points. Due to its simplicity in implementation, kNN is one of the most used supervised learning algorithms and has many applications. For example, online housing agencies can use the kNN algorithm to propose a price for a house based on the nearest neighbor prices they have.

Similar to clustering techniques, the best approach to identifying the optimal value for k is to test with different values for k and choose the one with the highest accuracy. This is known as *hyperparameter tuning* or sensitivity analysis as well. Assuming n is the number of data points, d is the number of dataset dimensions, and k is the number of nearest neighbors, the computational complexity of kNN is $O(nd + kn)$ or $O(kdn)$.

As brute force kNN is very slow, some approaches are used to make kNN perform faster. In the following, we explain three of these approaches, including Voronoi Tessellation, KD-Tree, and Local Sensitive Hashing.

Voronoi Tessellation

Voronoi tessellation [Voronoi '08] is the process of partitioning a 2D plane that includes several data points into regions, in which each region presents the area closest to its related data point and is centered around a data point. Check Figure 9-3(a); we have data points on a 2D plane, and in Figure 9-3(b), the regions were drawn by a Voronoi Tessellation, and the result is the Voronoi diagram. The result is called the Voronoi diagram (right side of Figure 9-3). Each region represents the area that is closest to its corresponding data point, and it is centered around a data

point. All points within a region are closer to its center data point than to any other data point. If a new data point is added to the plane, it creates a new region in the Voronoi diagram, altering the boundaries of the existing regions to ensure that each point in the plane is within the region of its closest data point.

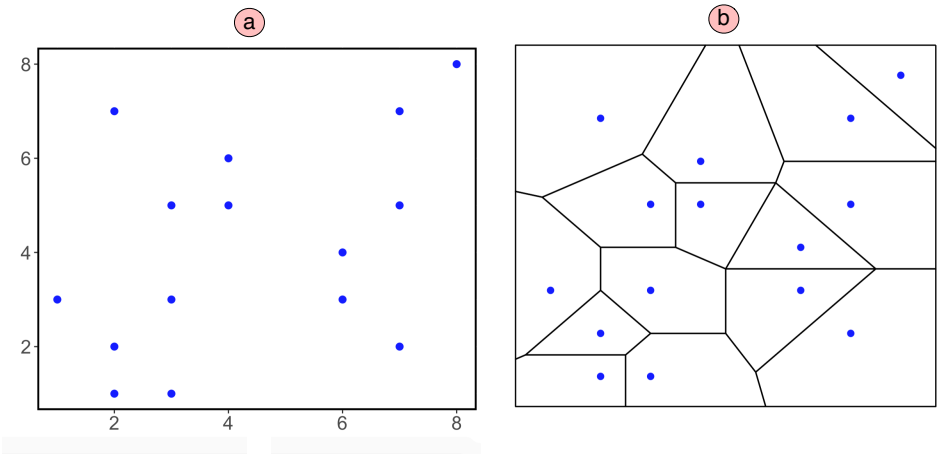


Figure 9-3: (a) Original dataset (b) Voronoi tessellation of the original dataset.

If a *k*NN algorithm sets $k = 1$, it can use Voronoi tessellation to decide about data points. In other words, Voronoi tessellation is the approach we use to quantify the 2D space for each data point. For $k > 1$, the process involves checking multiple regions around the test point. In particular, the nearest k neighbors (in terms of Euclidean distance) to the test point are identified. The labels of these k neighbors are then considered. The test point is assigned the label that is most frequent among these k neighbors.

Voronoi tessellation improves the search time to *O(log n)*, but it is only good for low-dimensional data and for higher than two-dimensional data, assuming *d* is the number of dimensions the computational complexity gets close to *O(d.n),* reflecting the challenge of distance computations and neighbor searches in higher dimensions.

KD-Tree

If you recall, in Chapter 5, we explained that one of the big challenges in working with data is to reduce the search space, and trees are very useful in reducing the search space. K Dimensional Tree or KD-Tree [Bentley '75] is used for the kNN algorithm to partition the search space efficiently for the nearest neighbors. While KD-Tree is used for exact nearest-neighbor searches, it can also be adapted for approximate nearest-neighbor searches to improve efficiency, especially in high-dimensional spaces. In the latter case, there is a trade-off where some of the nearest neighbors might be missed in favor of faster computation.

Let's use an example to understand how this algorithm distributes data into the tree. Assume we have the following dataset in two-dimensional space: {(2,1), (1,3), (2,2), (3,1), (3,5), (4,5), (4,6), (6,4), (7,4), (7,7), (8,8)}. We plot the dataset as it has been shown in Figure 9-4 (a).

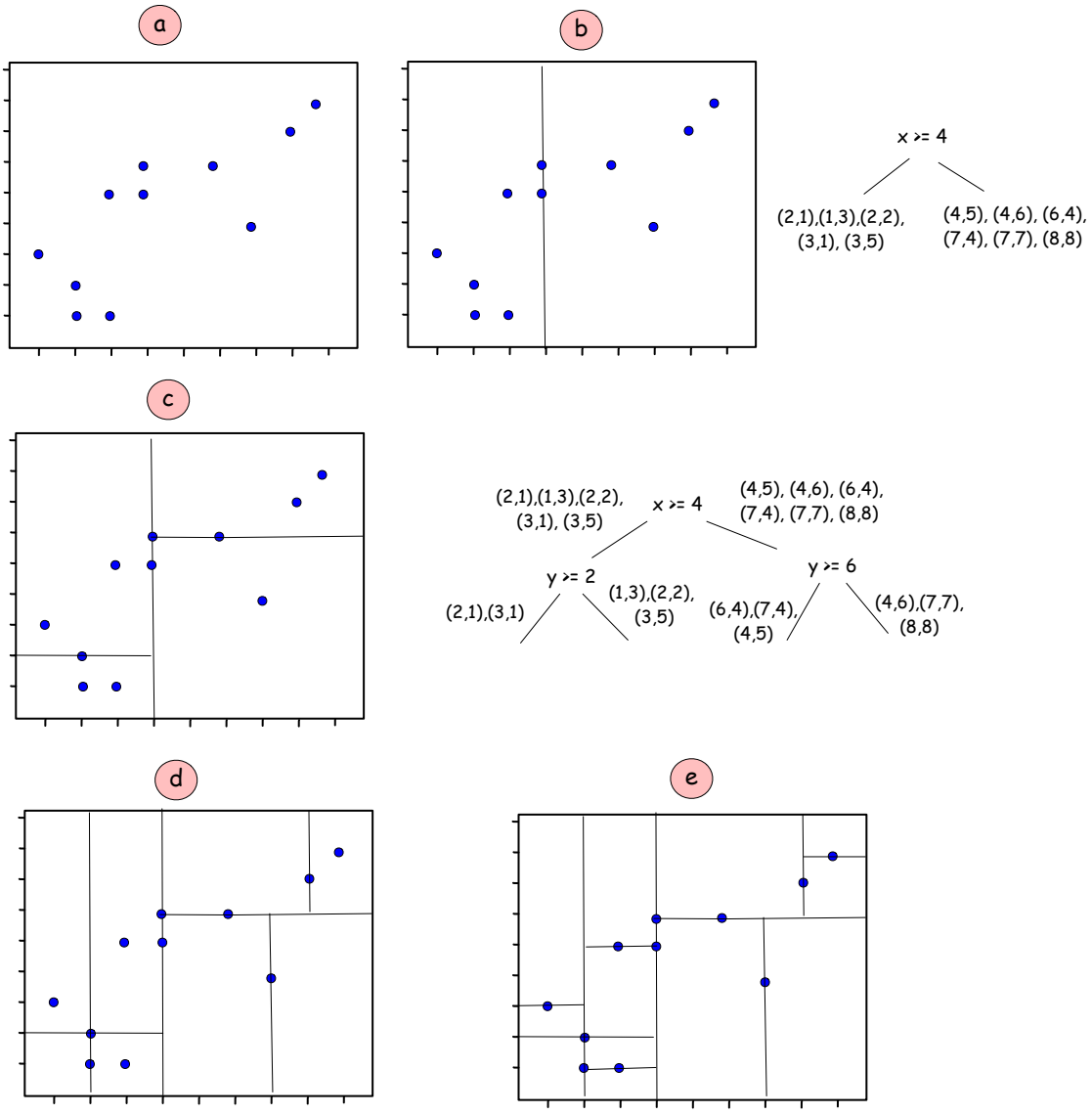


Figure 9-4: (a) The original dataset. (b) The median has been chosen, and the first branch of the tree is constructed to separate the dataset into two sub-datasets. (c) Now the dimension changed (Y-axis), and the same process will be repeated. (d) Again, the axis changed, and the same process. (e) No other data point left, and each node in the final tree has only one data point.

The process of KD-Tree partitioning starts by finding the median in one of the dimensions, e.g., here, we have two dimensions, and we choose x (we could also choose y). The values on the x-axis are {1,2,2,3,3,4,4,6,7,7,8} , and the median of this set is 4. Next, we plot the KD-Tree based on the median, as is shown in Figure 9-4 (b). We can see we have a tree with a node $x >= 4$ on its right side are nodes with larger or equal x values, {(4,5), (4,6), (6,4), (7,4), (7,7), (8,8)}, and on the left side, nodes with smaller x values are located, i.e., {(2,1), (1,3), (2,2), (3,1), (3,5)}.

Next, we switch to another dimension, i.e., the y-axis, and get the median of *y* coordinates. Since our dataset is partitioned into two segments, we do it once for each segment. The values on the y-axis on the left segment of Figure 9-4 (b) are {1,1,2,3,5}, its median is '2'. Now, the next node to construct the KD-Tree will be drawn on the y-axis at '2'; the same process will be done on the right side (the median on the y-axis is 6). See Figure 9-4 (c) to check its results. Again, as you can see from the tree on the right side of Figure 9-4 (c), the KD-tree algorithm goes into each branch and does the same (see Figure 9-4 (d)) until each branch has only one data point and the tree generation stops, as shown in Figure 9-4 (e).

By building such a tree to search for the nearest neighbor, the algorithm can navigate through the right branches instead of brute force search to find the nearest neighbors (check Chapter 5 if you can't recall the search on trees). Since it uses a tree, it improves the search for the nearest neighbor. To determine the label for a new data point, the new data point traverses the KD-tree to find its nearest neighbors. Once the k nearest neighbors are found, their labels are used to determine the label of the new point, typically through voting.

KD-Tree is useful when we have low dimensional data; if we need to deal with high dimensional data, it is recommended to use Local Sensitive Hashing. Assuming we have *n* data points and *k* dimensions, the computational complexity of KD-Tree is *O (k.n log n)*.

Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) [Indyk '98] is employed in various applications, such as finding duplicate documents, enabling search engines to locate similar images, aiding biologists in identifying similar gene expressions in genomic datasets, and detecting audio and visual similarities used in authentication systems. Therefore, please switch to full attention mode while reading this section.

Given the prevalence of high-dimensional data in real-world scenarios, LSH is often utilized to enhance kNN efficiency by hashing similar data points into the same buckets, thereby reducing the search space. LSH is somewhat analogous to clustering because it groups similar data points. However, while traditional hash functions aim to minimize collisions (check Chapter 5), LSH works in the opposite, and it is designed to maximize collisions for similar data points, effectively grouping them into the same buckets, which simplifies and speeds up the search process in kNN algorithms.

LSH receives the number of *separator hyperplanes* or *lines*, i.e., *k*, and the *number of iterations*, *l*, as input parameters (hyperparameters). It cuts the data space with lines or hyperplanes with *k* lines, and we end with a maximum 2^k of regions (or buckets) to search. The small regions that are generated by the separator lines are called buckets. Then, it separates the dataset into different regions, and when a new data point arrives, it compares it only to the other data points in that bucket. The algorithm runs this process *l* times.

For example, assume we set *k=3* and *l=3*. Figure 9-5 (a) shows the dataset that is converted into 7 buckets in Figure 9-5 (b). Each new data point that arrives will be compared only to its bucket. This is good, but we might miss some data points. For instance, in Figure 9-5 (b). 'f' belongs to

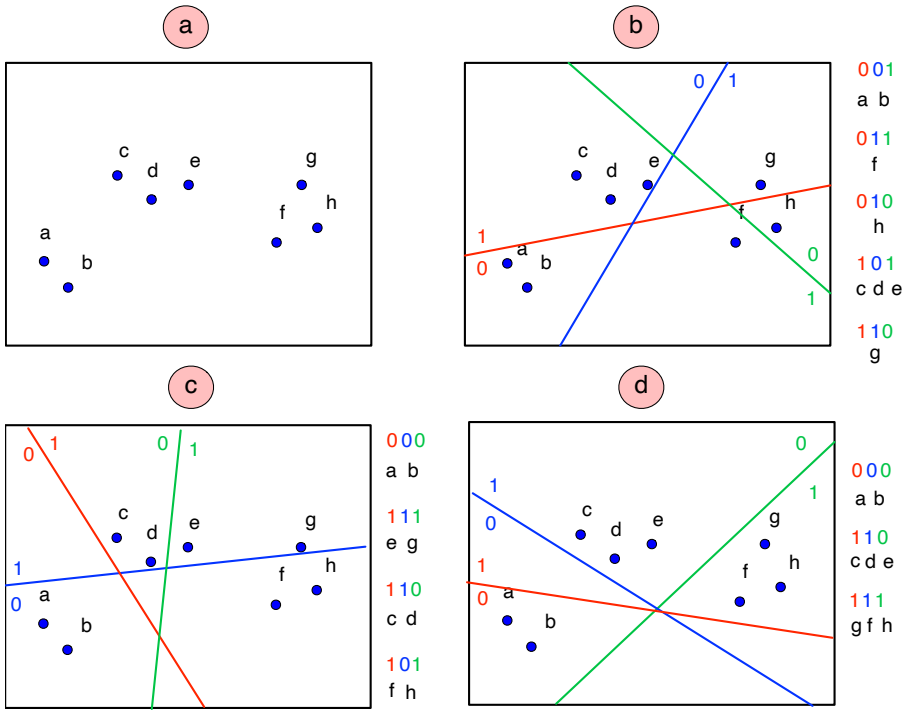


Figure 9-5: (a) The original dataset. (b) Three lines drawn as a hash function to separate each data point. (c) Another three separator lines drawn, and (d) for the third time, another set of line drawn. Note the result of this hashing is written on the right side of each figure.

the 011 and does not belong to the same bucket that 'g' (110) or 'h' (010) belongs. To mitigate this issue, we repeat the process of assigning data points to a bucket three times (because l=3), as shown in Figure 9-5 (c) and (d).

Considering the result of this bucketing, we can see 'a' and 'b' stay in the same region, 'c','d', and 'e' in the same region, and 'f' and 'h' are in the same region. However, we cannot judge for 'g', with only l=3. We should make l larger to identify neighbors of 'g'. When a new data object is received by the algorithm, kNN can use LSH and assign this data point to one of the existing regions. Also, the list of other data points for that particular region is available, and thus, the algorithm compares the newly arrived data point only to the data points that belong to that region (i.e., the same region). For example, if the hash code assigns region 000 to the newly arrived data object, based on Figure 9-5, it will be compared to 'a' and 'b' only.

Assuming n is the number of data objects, k is the number of separator lines or hyperplanes, and d is the number of data dimensions, the computational complexity of finding the right bucket (hash code) for a new data object is $O(d \,.\, k + O(\frac{d \,.\, n}{2^k}))$, or we can say it is close to $O(d \log n)$, which is close to KD-Tree complexity.

NOTES:

* While deciding about a *k* in *k*NN, there is no optimization-related algorithm exists for that. The best way to decide about *k* is to perform the parameter sensitivity analysis or hyperparameter tuning, i.e., experimenting and checking the result until we get the best parameter.
* It is recommended that the data be rescaled to have all features in the same numerical range while using the *k*NN algorithm.
* KD-Tree is also used to order the multidimensional space and facilitate search in the multidimensional environment.
* One of the biggest advantages of kNN is its non-linear decision boundary, which can even handle classifying complex shapes. For example, take a look at Figure 9-6, which shows a complex classification required to separate the class of blue from red data objects.

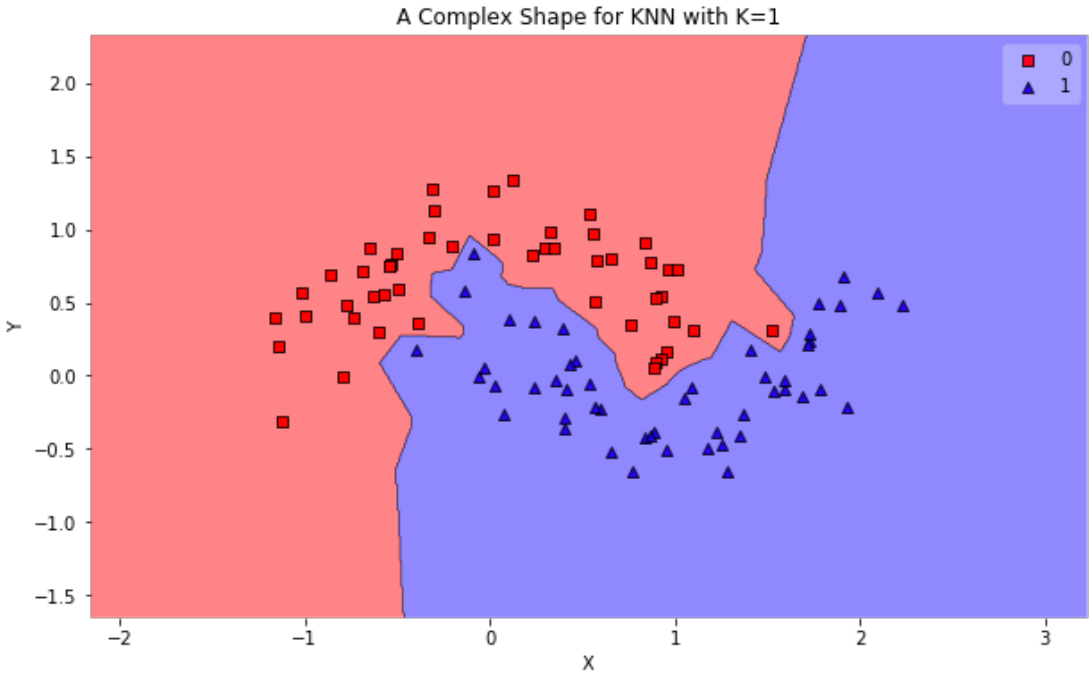


Figure 9-6: *k*NN algorithm that can classify fairly complex structures. We can see from this image that blue and red dots are interviewed, but the kNN can successfully classify them (look at the red and blue region).

Support Vector Machine (SVM)

An old but popular machine learning algorithm for classification is the Support Vector Machine (SVM) [Cortes '95]. It can classify linear and nonlinear data and performs very well despite its complex approach to classifying data. Besides, SVM can be used for regression as well, but it is commonly used for classification purposes, and we have rarely seen it used for regression.

To explain SVM, let's start with a simple example: we use SVM for binary classification. This example will give us intuition about the rationale of SVM. Then, we explain how a binary classifier extends into a multi-label classifier. Take a look at Figure 9-7 (a). it shows fairly well-separated datasets of blue and red dots. SVM can draw a line and separate them. Let's draw a random line that separates these two sets from each other, as shown in Figure 9-7 (b). After we have this separation line, we would like to test the separation quality and assume we get one new data point as a test, i.e., the grey data points in Figure 9-7 (c). We can see that this data point stays in the blue region, which does not seem correct because it stays closer to the red data points. We draw another separator line in Figure 9-7 (d), and this separates points better because the grey data point, which is closer to the red dots, stays in the red region now. The motivation of the SVM algorithm is to find the best possible *separation line* (or *hyperplane* for multidimensional data) that can separate data points from each other.

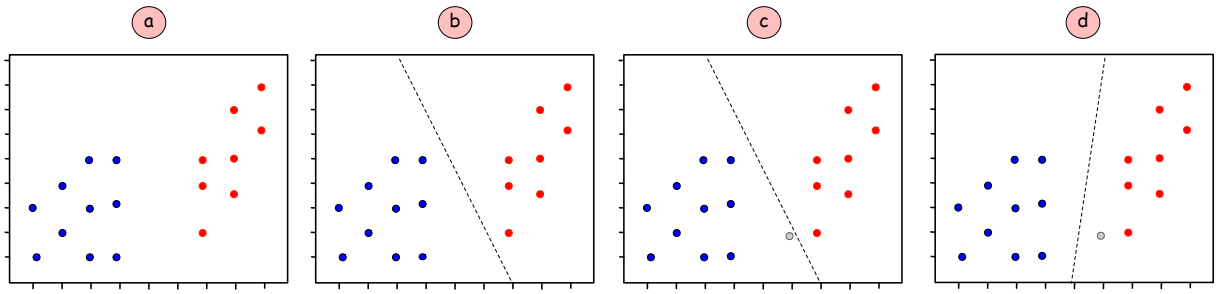


Figure 9-7: (a) sample data points that we intend to find a line to classify the space into the blue region and red region. (b) A random line has been drawn that it seems the space is well separated. (c) a new data point (from test set) is added and based on the drawn line if went into the blue region. However, it doesn't seem at the right place (d) another separator line is drawn and the data point fits into the red region.

To be more accurate from now on, we refer to this separator as a hyperplane instead of dots, lines, etc. The hyperplane for p dimensional space is formalized with a line equation as follows: $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_p x_p = 0$. In Chapter 8, we explained that β_0 is an intercept, and others are the slopes of the hyperplane. $\vec{\beta} = \{\beta_1, \beta_2, \ldots\}$ is called a *weight vector*, and some literature uses $\overrightarrow{w}$ to present it. Thus, assuming X is a set of features (feature vector), $X = \{x_1, x_2, \ldots, x_p\}$ and $\beta_0 = b$ (bias), the hyperplane equation is written as: $\overrightarrow{w} \,.\, X + b = 0$.

Our objective in the given example is to specify the new data points' label as blue or red (classify data points from the test set), and it is a binary classifier, i.e., $y = -1, \quad y = 1$. Unlike other binary classifiers, instead of using 0 and 1, SVM uses -1 and 1. Therefore, we can say one side of the hyperplane is blue ($y = -1$), and the other side is red ($y = 1$) and formalize it with the following equations:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_p x_p \leq 1 \quad or \quad \overrightarrow{w} . X + b \leq 1 \rightarrow y = -1$$
$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_p x_p > 1 \quad or \quad \overrightarrow{w} . X + b > 1 \rightarrow y = 1$$

The test observation data point x^* can be classified by substituting its features $\{x_1, x_2, \ldots\}$ into the hyperplane equation $f(x^*) = \beta_0 + \beta_1 x_1^* + \beta_2 x_2^* + \ldots \beta_p x_p^*$. If $f(x^*) > 1$ then the new test data point x^* will get the label 1 (e.g., red), and if $f(x^*) \leq 1$, then, x will get label -1 (e.g., blue).

In other words, when a new data point from the test dataset $\{x_1, x_2, \ldots\}$ arrives, the algorithm substitutes these x values into the hyperplane equation, and the result will be a number (either ≤ 1 or > 1), which is used by the algorithm to decide about the label of this data point. Therefore, the output will be something like the following: $\{x_1 = -1, x_2 = 1, x_3 = 1, \ldots\}$. Now, the question is, how do we draw this separator hyperplane? Or how do we identify weight vectors $\vec{\beta}$ or $\overrightarrow{w}$ of the hyperplane equation?

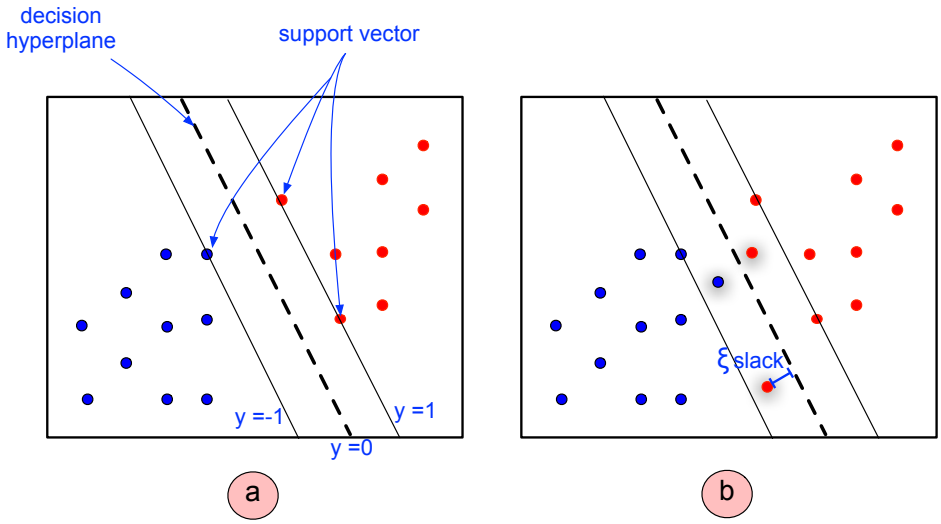


Figure 9-8: (a) The sample data points used by the maximum margin classifier support vectors (points on the line construct support vector) and the decision hyperplane were identified and highlighted. (b) Three sample data points (they have shadows) stay inside support vectors, and one data point stays on the wrong side of the hyperplane (slack variable). A good margin classifier should reduce these issues as much as possible.

The answer is to use the *Maximal Margin Classifier*. Here, the *margin* refers to the distance between a data point and the separator hyperplane. The maximal margin classifier uses a hyperplane with the largest margins (largest distance between data points and hyperplane). Figure 9-8 (a). shows another dataset with blue and red classes. It looks like a street; the centerline is called the decision hyperplane**,** and the sidelines parallel to it are called support

vectors. Figure 9-8 (b) shows the margin lines and the hyperplane, but there are some data points inside the street, which we explain later how to deal with.

The separator hyperplane will be identified with algorithms such as the *Sequential Minimal Optimization* algorithm (check Chapter 8 to recall optimization), which we put under the rug, not to explain it. You might not need to learn SVM optimization unless you intend to design an optimization algorithm or do research on mathematics and theories of optimization.

In short, the SVM optimization algorithm tries to find the hyperplane based on two objectives: (i) it makes the maximum margin possible (making the street wider in Figure 9-8) while limiting margin violations. Data points between vector spaces mean margin violation (see Figure 9-8 (b)). However, since this phenomenon is unavoidable in real-world datasets where data is noisy, we tolerate it and call the classifier that tolerates these points a *soft margin classifier*. (ii) There is a constraint that specifies that each side of the hyperplane should cover most data points from each class. The hyperplane must stay between two classes to separate the data points.

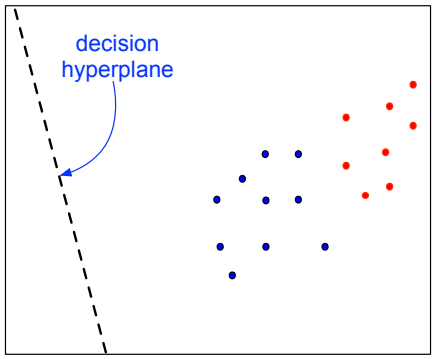
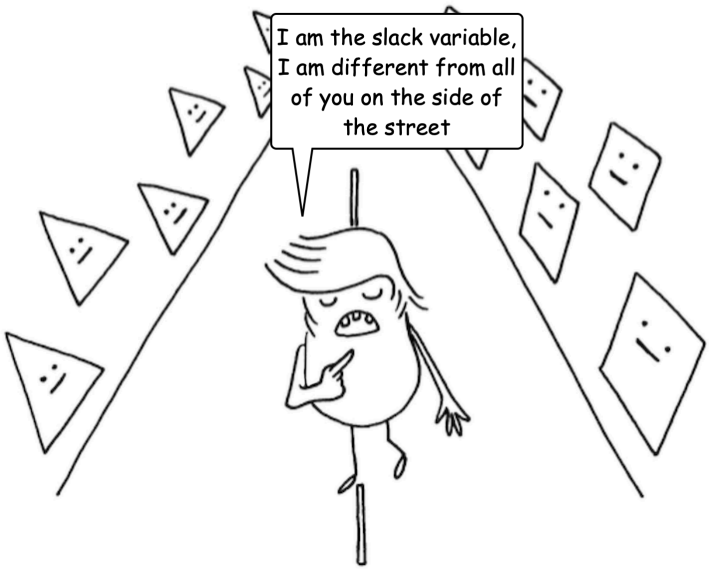


Figure 9-9: A decision hyperplane that has a very large separator margin, but it doesn't make sense. Because the separator constraint does not exist.



Otherwise, this hyperplane can stay far away on the other side of the world and thus make a very large margin, but it doesn't make sense. For example, look at Figure 9-9; the margin is very large but does not classify the dataset.

The loss function used by optimizations for maximum margin classification algorithms (including SVM) is called *hinge loss*. Some implementations of SVM enable the user to choose a different loss function as a hyperparameter for the algorithm.

We have explained that optimization will take care of hyperplane creation. Now, take a look at Figure 9-8 (b). In reality, outliers and data points can not stay in the correct class, either between decision boundaries or on the wrong side of the hyperplane. We have no other choice than to tolerate their misclassification in hyperplane. Data points that stay between support vector lines or on the wrong side of the hyperplane are called *slack variables*[3]. Figure 9-8 (b) highlights three slack variables three between support vectors, and one of them stays on the wrong side of the hyperplane. The objective function of SVM is not only to maximize the margin but also to minimize these slack variables, which essentially means reducing the hard margin violations as much as possible.

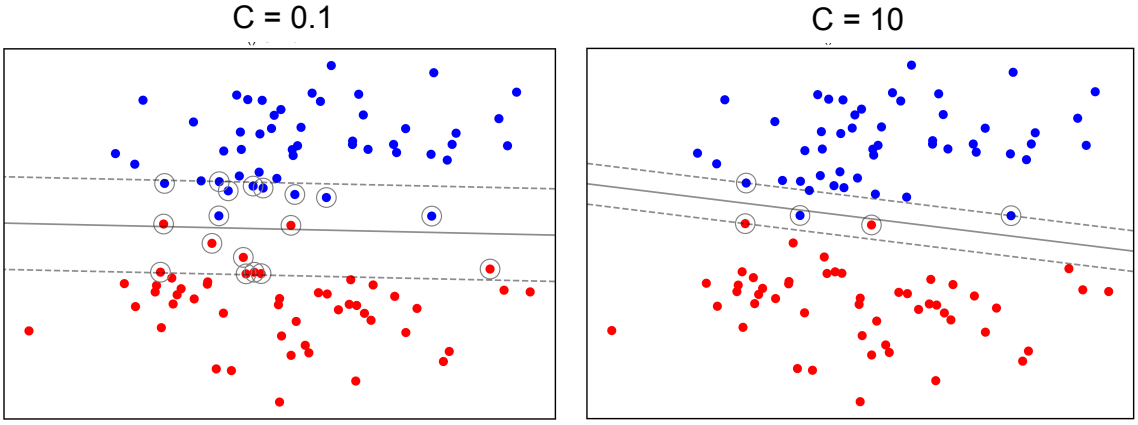


Figure 9-10: (a) A sample dataset and C=0.1 present a very small margin. Therefore, the variance is low, but as you can see, we have high bias. (b) A larger value has been set for C=10, and the bias is getting lower, but the variance has increased.

The *C* hyperparameter in SVM controls the trade-off between the model's margin width and the tolerance for margin violations; the lower *C* means less tolerance for any misclassification is allowed. Figure 9-10 presents two different values for *C*. We can not set *C* to *0,* its value must be positive, and a low but positive *C* value would indicate a high tolerance for margin violations, not zero tolerance. The choice of the C hyperparameter indeed represents a bias-variance trade-off. A higher *C* value (resulting in a "narrower street") minimizes the number of margin violations, aiming for higher accuracy on the training set at the risk of overfitting, thus exhibiting low bias but high variance. On the other hand, a lower *C* value (resulting in a "wider street") allows for more margin violations. This approach prioritizes generalization over training set accuracy, potentially leading to a model with higher bias but lower variance due to its increased tolerance for misclassifications.

---

[3] In the context of optimization to convert an inequality to equality, we can use a slack variable. For example, if $x + 2y < 3$, by using a slack variable ξ (read it as ksee), it can be written as: $x + 2y + \xi - 3 = 0$

Adjusting the C hyperparameter in SVMs is crucial for balancing the bias-variance trade-off. Higher values of C focus on reducing training errors, potentially at the expense of generalizability, while lower values of C aim for better generalization, potentially at the expense of increased training errors.

Handling Non-linear Data with Maximum Margin Classifier

Until now, all examples we have defined have been easily separable from a line. Take a look at Figure 9-11 (a); how can we use a maximum margin classifier and draw a hyperplane to classify it? To handle non-linear data, we must transform the feature space into quadratic (to the power of two), cubic (to the power of three), or higher-order polynomials. In other words, we need to increase the dimensionality of the data. Back in Chapter 6, we explained that dimensionality reduction is very useful, but in this particular scenario, we need to increase the dimension of the data to classify it. To our knowledge, this is the only place in machine learning where we increase the dimensionality.

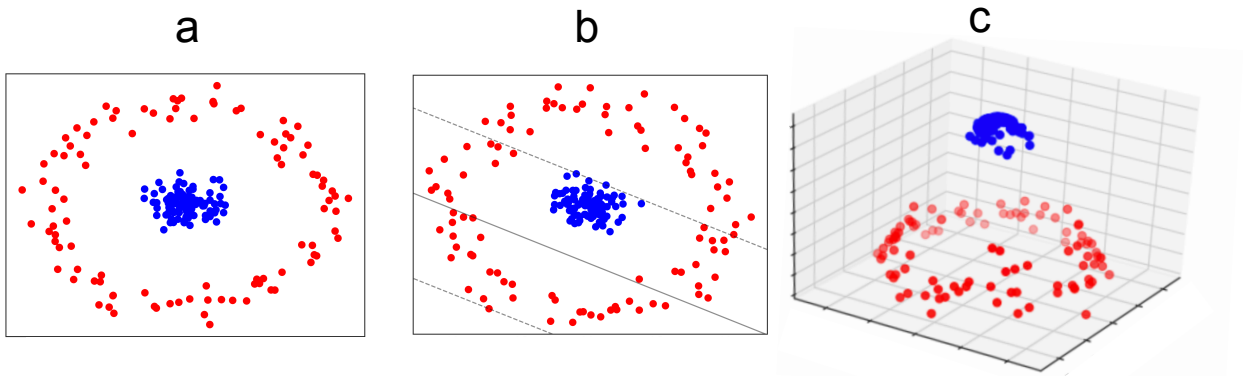


Figure 9-11: (a) A sample dataset that has two different labels. (b) The maximal margin classifier can not handle the data in two dimensions, and you can see that the maximal margin classifier line can not separate the data. (c) After applying the kernel function and bringing the data into a higher dimension, a hyperplane can separate the blue from red dots.

To implement such a high-dimensional transformation, we use the *Kernel* function. Therefore, the kernel function transforms a non-linear, inseparable dataset into a linear and separable dataset. For example, without applying a kernel function in Figure 9-11 (a), we can see that the maximal margin classifier lines cannot be classified as blue and red dots, as shown in Figure 9-10 (b). However, by transforming each data point into its polynomial degree, Figure 9-10 (c), we can project them in a higher dimensional space where blue and red dots can be separated with a linear hyperplane that acts as a border between red and blue dots. Imagine we draw a page (2D hyperplane) between red and blue dots, and this page separates them from each other.

To summarize, SVM uses a computationally efficient kernel function that transforms the input dataset into non-linear higher-dimension data. In the next section, we describe some details about the kernel functions.

Kernel Trick

As we have explained before, the kernel function is used by the SVM algorithm to create an extended representation of the data, and it *converts* a *non-linear separable dataset into a linear separable dataset*. In other words, we use the kernel function, which is a similarity calculation function, to bring our dataset into a higher dimensional space and thus make it separable with a linear hyperplane. After transforming the dataset into a higher dimensional space, the SVM can separate them with a linear hyperplane (see Figure 9-10).

Assume we have a dataset X with each of its data points has two features x_1 and x_2. Data points in this dataset do not have linear relations. $\Phi(X)$[4] is a function that creates a representation of X in the higher dimensional space[5], let's say, with two polynomial degree dimensions. Therefore, we can write the following: $\Phi(X) = \{x_1, x_2, x_1^2, x_2^2, x_1x_2\}$. A linear hyperplane might separate the dataset in this higher dimensional space. This means that from smaller dimensions, which were not linear, we transform the data to have higher dimensions, and data in the higher dimensional space have a higher chance of getting separated with a linear hyperplane.

If the dataset's data points have many features, a simple second-degree polynomial transformation increases the number of features quadratically, i.e., X^2. What happens if the algorithm still can not separate them? We can increase the degree of polynomiality. For example, using a third-degree polynomial will result in X^3 features to process. For example, if we have 10 features and 3000 data points, we end up having $10^3 \times 3000 = 3{,}000{,}000$ features to deal with, and this is computationally very inefficient or even infeasible. This shows that the polynomial transformation increases the computational complexity, and we need a more subtle approach to deal with this problem.

Since transforming data into a higher dimension is computationally expensive, the kernel trick is used. SVM employs a kernel function to compute the similarity (or inner product) between data points directly in the original space, avoiding explicit transformation into a higher dimension. This kernel function is a similarity measure that facilitates the identification of patterns or structures in the data, particularly in SVM and other kernel-based algorithms. The process primarily involves labeled data in supervised learning to determine the decision boundary or classification rules.

Computing the similarity between data points can be considered as a *dot product* in a high-dimensional feature space without explicitly mapping the data to this space. If you can't recall from math what is an inner product, assume we have two vectors $a = [3,2,4]$ and $b = [1,4,0]$, and we would like to compare them together. The inner product of a and b is written as $a \,.\, b = 3 \times 1 + 2 \times 4 + 4 \times 0 = 11$. In particular, a kernel trick allows the comparison of two data points by converting the complex dot product in the high-dimensional space into a simpler calculation in the original input space. This is a good thing because instead of computing the dot

---

[4] ϕ is a greek letter and read it as 'fee' like feel.

[5] There is a theory called the *Mercer theorem***.** It states if there is a continuous function that is symmetric, i.e., $F(x, y) = F(y, x)$, there is a function ϕ that can bring $F(x, y)$ into another dimension that can be represented as the inner product of $\phi(x)^T$ and $\phi(y)$, i.e., $F(x, y) = \phi(x)^T \,.\, \phi(y)$.

product between high dimensional features, we can calculate a dot product in low dimensional space and raise it to the power of polynomials. For example, assume we have a labeled train-set $X = \{x_1, x_2, x_3\}$ and we have an unlabeled test-set $X' = \{x'_1, x'_2, x'_3\}$. Two degrees of polynomiality transformation are $\Phi(X) = \{x_1^2, x_1x_2, x_1x_3, x_2x_1, x_2^2, x_2x_3, x_3x_1, x_3x_2, x_3^2\}$ and $\Phi(X') = \{x_1'^2, x'_1x'_2, x'_1x'_3, x'_2x'_1, x_2'^2, x'_2x'_3, x'_3x'_1, x'_3x'_2, x_3'^2\}$. Now, we would like to compare $\Phi(X)$ with $\Phi(X')$ to label $\Phi(X')$ set from $\Phi(X)$ the labeled set. We can see that we have too many features to compare because, from a three-dimensional space, we moved into a nine-dimensional space.

By using the kernel trick, we avoid the explicit computation of the dot product between $\Phi(X)$ and $\Phi(X')$, in the transformed higher dimensional space. Instead, the kernel function calculates a value representing this dot product, but it does so based on the original data, i.e., X and X'. Therefore, we can write that $K(X, X') = < \Phi(X), \Phi(X') >$.

Let's summarize how a kernel trick works: To compute the similarity between two data points as if they were in a higher-dimensional space, the kernel function calculates a dot product (or another similarity measure) in the original, lower-dimensional space. This process simulates the comparison in the higher-dimensional space without explicitly performing the transformation, thus efficiently achieving the same result as if the data were compared in that higher-dimensional space.

Kernel Functions

Now we have understood the advantage of the kernel trick; congratulations, you have learned a secret in mathematics; if you think mathematicians' secrets are also very boring, we agree with you. There are several popular kernel functions used for SVM. The popular ones are listed as follows.

Linear kernel: it is written as $K(X, X') = X \,.\, X'$, which is a simple dot product of two vectors.

Polynomial kernel: it is written as $K(X, X') = (X \,.\, X' + r)^p$. Here, r presents the polynomial coefficient, and p presents the polynomial degree, which both are hyperparameters of the polynomial kernel function.

Gaussian Radial Basis Function (RBF) kernel: RBF Kernel is written as follows: $K(X, X') = exp(-\gamma \,||X - X'||^2)$. In this equation, γ is a hyperparameter that controls the spread and smoothness of the kernel. A higher γ leads to a narrower kernel, which responds more strongly to nearby data points and less to distant ones), and a lower γ responds more uniformly to nearby and distant data points.

Sigmoid kernel: it is written as $K(X, X') = tanh(\gamma X^T X' + r)$. Here, r presents the polynomial coefficient and γ is a hyperparameter representing the curvedness of the separation area (similar to the Gaussian RBF kernel, *tanh* is a hyperbolic tangent function.

RBF is the most common kernel function used for the SVM algorithm. Figure 9-12 shows an SVM algorithm run with three values for γ. Gaussian RBF kernel is a popular kernel function for SVM. Geron [Geron '19] states that γ acts as a regularization parameter. If the SVM model is

overfitting, this means the decision boundary is too complex and closely fits the minor fluctuations in the training data. By decreasing the value of γ we might resolve it. If the model is underfitting, the decision boundary is too simple and cannot capture the complexity of the data. By increasing γ we can resolve the underfitting.

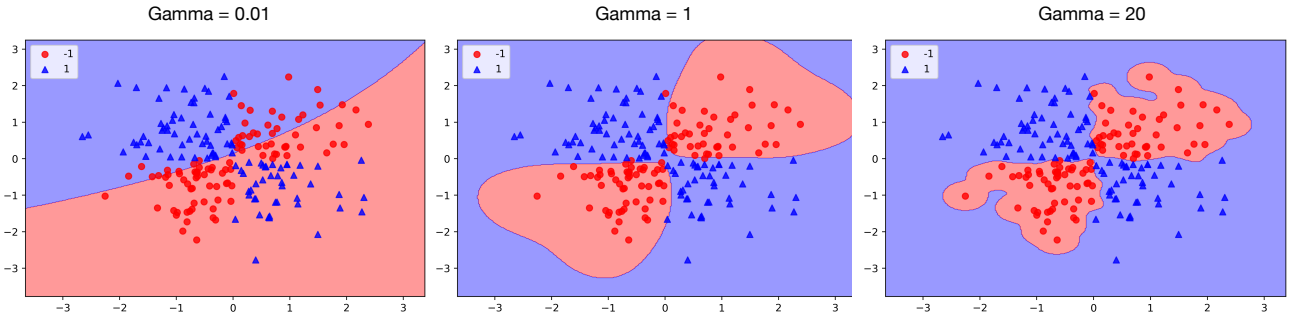


Figure 9-12: Different values for gamma of the RBF kernel function and their impact on the classification quality of the SVM. Setting gamma high improves the separation capability of the classifier, but setting gamma too high, like the most right figure makes the model prone to overfitting.

Unless you would like to be a kernel function designer, you don't need to understand the mathematics behind any of these kernel methods, but it might be useful to know their equation. However, we heard from our students that a psycho-recruiter asked a poor candidate to describe Kernel equations at job interviews for data science.

While using SVM, we can specify the kernel function as a parameter of the SVM algorithm and hyperparameters of the kernel function. A question might arise: How does the kernel function decide the polynomial degree, or how were polynomial parameters determined? By using the cross-validation the algorithm performs hyperparameter tuning, and some implementations automatically choose the best possible parameters for the selected kernel method. In other implementations, we should give the parameter as an input variable. Therefore, this depends on the SVM library or package we are using. It is better to start with a linear kernel, and as we have explained before, linear models are always favored over non-linear models because of their resource efficiency.

Multi-label Classification for SVM

Binary classification is straightforward. But what if we want to use SVM to classify a dataset based on four labels, such as {blue, red, green, orange}? In this case, SVM can perform binary classification in a one-versus-all manner. This means the SVM will first classify blue against all other labels {red, green, orange}. Then, it will classify red against all other labels, followed by green against the rest, and finally orange against the others.

In each iteration, each data point receives a label. For example, data point x might be classified as green in one iteration and as not orange, not blue, and not red in the others. Ultimately, x will receive the green label. This approach to handling multi-class classification is called one-versus-all classification.

How Does the SVM Perform the Prediction?

We have explained that SVM uses an optimizer to determine the support vector and hyperplane equation parameters. We have not explained how the optimizer works. Very briefly, a Lagrangian equation is constructed first, and then the optimizer solves the solution using Karush-Kuhn-Tucker (KKT) conditions. In particular, the Lagrangian function introduces multipliers (α_i) for each data point, which helps in transforming the problem into a form that can be solved using optimization techniques. The optimization problem is then solved using the Karush-Kuhn-Tucker (KKT) conditions, which provide the necessary conditions for a solution to be optimal. These conditions help find the support vectors and determine the coefficients of the hyperplane. Based on the quadratic optimizerthe following equation can be used to calculate , the maximal margin classifier.

$$d(X') = \begin{cases} -1 \quad if \quad \sum_{i=1}^{l} y_i \alpha_i X_i X' + \beta_0 \leq 0 \\ 1 \quad if \quad \sum_{i=1}^{l} y_i \alpha_i X_i X' + \beta_0 > 0 \end{cases}$$

In this equation, α_i (the Lagrangian multiplier) and β_0 (bias) parameters are determined by the optimizer. y_i is the class label of input data points X_i (training set), and X' is the test set. Therefore, the X' data points will be substituted into this equation, and the sign of the equation will determine which side of the hyperplane X' belongs (what label it receives).

SVM Computational Complexity

Calculating the computational complexity of SVM is dependent on the optimization algorithm to identify the maximal margin classifier and the kernel method we choose to use for SVM. For example, for the training phase, the computational complexity depends on the optimization algorithm, which is a quadratic problem-solving that has $O(n^3)$ complexity [Bordes '05]. For the testing phase, the computational complexity depends on the kernel method; assuming n presents the number of data points and m presents the number of features, linear kernel SVM has $O(n \times m)$ computational complexity. Polynomial kernel SVM with p degree of the polynomial has $O(n \times m^p)$. Therefore, there is no constant complexity that we can report on SVM; it is one of the most accurate traditional classification algorithms, but it has a high computational complexity.

NOTES:

* Both kNN and SVM can handle nonlinear data structures, and when a dataset is small, they are good options for classification.
* kNN is not good for higher dimensional data; SVM is particularly known for its effectiveness in high-dimensional spaces, even with a small number of samples.
* SVM is sensitive about scaling, and scaling the data will result in a change in SVM behavior. Therefore, if your dataset changes frequently, be careful when using SVM.

* The Support Vector Classifier (SVC) algorithm is the SVM that is used for classification. SVM alone can be used for both regression and classification.
* If classes are highly overlapped, logistic regression is performing better than SVM. If they are well separated, SVM usually performs better than logistic regression. However, the best approach is to try many algorithms and observe which performs better or use ensemble learning methods, which we will explain later in this chapter.
* If you remember, we discussed that if the optimization has a convex shape, local minima are the global minima as well. SVM parameter optimizations also use a convex shape function.
* In comparison to other classification algorithms, the SVM algorithm is generally slow, but it is fairly robust against overfitting, which is a common problem in classification algorithms.

Decision Trees

Back in Chapter 5, we explained that trees reduce search space and thus make the search algorithm faster and more resource-efficient. There is another group of trees that can be used for classification and regression, which we explain in this section. Decision trees extract human-understandable rules, and their explainability makes them very popular. Similar to previous trees we have explained in Chapter 5, decision trees split the search space (dataset) into smaller segments and search those segments instead of the entire dataset. Because of this feature, they are usually resource-efficient.

The first practical version of the decision tree was proposed in the early 80s, i.e., CHAID [Kass ‘80]. A few years after CHAID, in 1984, Breimanetal [Breimanetal ’84] introduced the CART algorithm, another popular decision tree algorithm that is still in use. Two years later, Quilan introduced ID3 [Quilan ’86], and then proposed C4.5 [Quilan ’94]. All of these four algorithms are still in use. After the introduction of these trees, Gradient Boosting Decision Trees [Mason ’99] were introduced, and in 2014, XGBoost [Chen ’15] won several Kaggle competition awards. The success of XGBoost leads to shifting the machine learning community’s attention to decision trees again. At the time of writing this book, deep learning algorithms have the highest accuracy for unstructured data, but gradient-boosting decision trees perform better than deep learning methods when analyzing tabular data.

Decision trees, similar to other supervised algorithms, use a training dataset to construct a tree. Then, after the decision tree is built, the unlabeled data point navigates through the conditions of the trees to get the label. Leaf nodes (blue-colored nodes in Figure 9-13) of a decision tree are used to assign labels to the tree.

For example, a person who is going under mental therapy is journaling his happiness level based on three different factors: ‘having gratitude’, ‘comparing himself with others’, and ‘expecting from others’. Figure 9-13 presents the daily information that he collects about himself and what he is thinking about his behaviors and attitudes. He would like to know when he goes to bed and thinks about his day, whether he feels happy or not. We make his job easier by building a

	Gratitude	Compare himself with others	Expecting from others	Happy
train	X	X	high	X
	✓	✓	low	✓
	✓	X	medium	✓
	✓	X	low	✓
	X	✓	medium	✓
	X	X	high	X
	✓	X	medium	✓
	X	X	high	X
	X	✓	medium	✓
	✓	✓	low	✓
	X	X	medium	X
	✓	X	high	X
test	✓	X	low	?

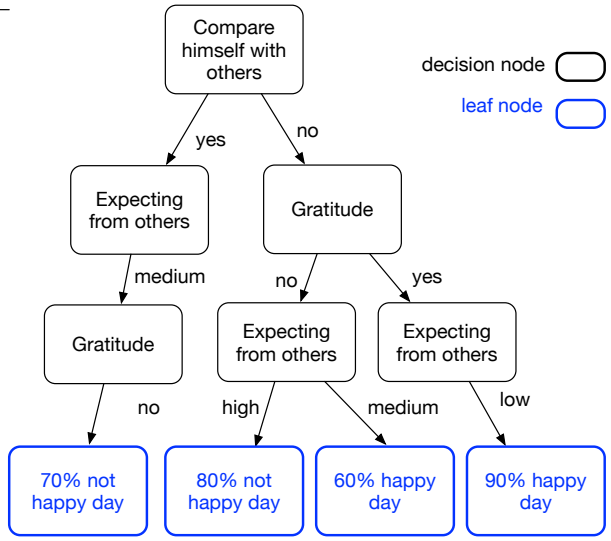


Figure 9-13: A sample dataset along with a simple decision tree, we like to predict the happiness status of the test dataset. On the right side, the algorithm constructs an example tree, and by tracing this tree, the algorithm can make the decision about the test dataset (the ? marks of the blue record in the table).

prediction model that assigns the label (happy / not happy) before he ends his day based on the data he gives to the algorithm.

After the training phase, the decision tree receives a vector of features (the test record in the table of Figure 9-13, which is presented in blue color), and the decision tree starts to navigate its branches until it identifies the missing labels ('?', in the table of Figure 9-13).

Decision trees have three important characteristics:

(i) Decision trees are operated based on the policy of finding the best feature to split the training data into subsets until there is no possible split into a smaller subset. When no further split is possible, the decision tree has been constructed. Next, the test dataset will use this tree to label its records.

(ii) The leaf nodes in decision trees present the 'outcome' of the path from the root to that node. All non-leaf nodes are called 'decision nodes' because the algorithm uses them to decide on navigating in which branch reaches a leaf node at the end. The decision tree of Figure 9-13 shows leaf nodes in blue and decision nodes in black.

(iii) Decision trees are explainable, making them very popular algorithms for real-world applications. Decision tree results are a set of if-then-else rules, making them explainable, which is not the case for many machine learning algorithms, including neural networks. If you

encounter a fancy word like *explainable AI (xAI)*, it means the machine learning algorithm or model is explainable and not BlackBox.

Please fasten your seat belt. We are about to land on the big planet of decision trees and learn four vanilla decision trees, followed by gradient boosting decision trees.

Iterative Dichotomiser 3 (ID3)

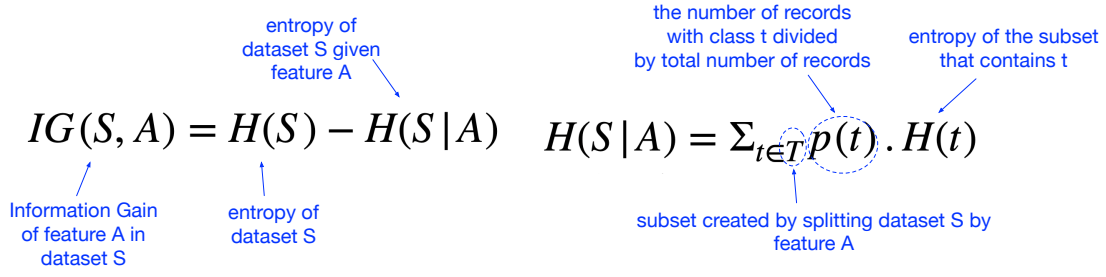
ID3 is one of the simple decision trees that iteratively dichotomizes (divides into two groups) the dataset at each step. Recall that we have explained that the process of decision tree creation is to split the training data into smaller subsets. Looking at the tree in Figure 9-13, we have selected "Compare himself with others" as the root node, but is it the best feature to select? Probably not, and we need a mechanism to choose the best feature to split the data. By splitting the data, we mean choosing the feature that can separate records based on the target label more accurately than other labels. In other words, choose the best separative feature.

ID3 uses *Information Gain (IG)* to select the best feature. Information gain is based on the concept of *entropy*. Entropy is a number that quantifies the uncertainty or disorder (check Chapter 3 to recall it). Entropy specifies (in number) how well a given feature separates the dataset records based on the label. Assuming n is the number of possible labels (classes) in the training set, S denotes the train dataset, p_i denotes the probability of having class i (the number of records with class label i divided by the total number of dataset records). In other words, p_i is the proportion of elements in S that belong to class i. The entropy $H(S)$ is computed as follows:

$$H(S) = -\sum_{i=1}^{n} p_i log_2(p_i)$$

In other words, $H(S)$ refers to the average amount of information required to identify the class label of a new record.

Each column in the table of Figure 9-13 presents features of the decision tree, and we can write information gain for the feature A (a column of data) as: $IG(S, A) = H(S) - H(S|A)$.



Here, IG presents the differences between the original information, i.e., $H(S)$ and the required information, i.e., $H(S|A)$. Now that we understand IG and entropy in the context of ID3, we can learn step by step how the ID3 algorithm works.

As the first step, the algorithm should find the feature (column) with the highest split against one of the possible values of the labels. To find this feature, it measures the IG for every feature

(column) of the dataset. In the second step, it splits the training dataset *S* based on the feature that has the highest IG. Next, in the third step, if all rows in a tree node belong to the same class, then the algorithm marks this node as a leaf node and does not split it any further. Otherwise, it repeats the process from the first step until all nodes are converted into a leaf node, and no path remains from root to decision nodes without ending in a leaf node.

To better understand the ID3 algorithm, let's explain it with the example we have used in the table of Figure 9-13. As the first step, we calculate the entropy of the target column that we would like to predict; in our case, it is the "Happy" column. We have seven yes (✓) and five no (X) for this column. Its entropy is calculated as:

$$H(S) = -(7/12).log_2(7/12) - (5/12).log_2(5/12) = 0.96$$

Now we have the entropy of "Happy", and we should find which column has the largest split on "Happy", so we calculate the IG value for each column (feature). We have the following IGs for each feature (column in the table of Figure 9-13). The sign # is used to refer to the number of occurrences.

"Gratitude": yes, #rows: 6, #happy: 5, #not happy:1

$$H(S_{yes}) = -(5/6).log_2(5/6) - (1/6).log_2(1/6) = 0.65$$

"Gratitude": no, #rows: 6, #happy: 2, #not happy:4

$$H(S_{no}) = -(2/6).log_2(2/6) - (4/6).log_2(4/6) = 0.918$$

The IG of "Gratitude" will be calculated as follows:

$$IG(S, Gratitude) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) =$$

$$0.96 - (6/12)(0.65) - (6/12)(0.918) = 0.176 \leftarrow \textbf{\textit{IG of gratitude}}$$

"Compare himself with others (Compare)": yes, #rows: 4, #happy:4, #not happy:0

$$H(S_{yes}) = -(4/4).log_2(4/4) - (0/4).log_2(0/4) = 0$$

"Compare himself with others": no, #rows: 8, #happy:3, # not happy:5

$$H(S_{no}) = -(3/8).log_2(3/8) - (5/8).log_2(5/8) = 0.95$$

The IG of "Compare himself with others" will be calculated as follows:

$$IG(S, Compare) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - 0 - (8/12)(0.95) = 0.32$$

$$0.96 - 0 - (8/12)(0.95) = 0.32 \leftarrow \textbf{\textit{IG of Comparing himself with others}}$$

"Expecting from others (Exp)": high, number of rows: 4, #happy:0, #not happy: 4

$$H(S_{high}) = -(0/4).log_2(0/4) - (4/4).log_2(4/4) = 0$$

"Expecting from others": medium, #rows: 5, #happy:3, #not happy: 2

$$H(S_{medium}) = -(3/5).log_2(3/5) - (2/5).log_2(2/5) = 0.97$$

"Expecting from others": low, #rows: 5, #happy:4, #not happy: 1

$$H(S_{low}) = -(4/5).log_2(4/5) - (1/5).log_2(1/5) = 0.72$$

Gratitude	Compare himself with others	Expecting from others	Happy
X	X	high	X
X	X	high	X
X	X	high	X
✓	X	high	X

Gratitude	Compare himself with others	Expecting from others	Happy
✓	✓	low	✓
✓	X	low	✓
✓	✓	low	✓

Gratitude	Compare himself with others	Expecting from others	Happy
✓	X	medium	✓
X	✓	medium	✓
✓	X	medium	✓
X	✓	medium	✓
X	X	medium	X

This dataset will be used to extend the tree of

Table 9-4: The result of splitting the original dataset in the table of Figure 9-13 based on the "Expecting from Others" column. Now, we have three datasets, but the two on the top are leaf nodes because we cannot further split them.

The IG of "Expecting from others" will be calculated as follows:

$$IG(S, Exp.) = H(S) - (S_{low}/S)H(S_{low}) - (S_{medium}/S)H(S_{medium}) - (S_{high}/S)H(S_{high}) =$$

$$0.96 - (5/12)(0.72) - (5/12)0.97 - 0 = 0.25 \leftarrow \textbf{\textit{IG of Expecting from others}}$$

We understand that the highest information gain (0.32) belongs to "Compare himself with others". This means that the training dataset can be split on this feature because it provides the best split in comparison to other features (Comp., Exp.). Therefore, we start to construct the tree as it is shown on the right side of Figure 9-13.

We do not explain the tree construction further. The rest will be done the same until no decision node remains, and we end up with all branches in a leaf node. If the split is based on another feature, such as "Expecting from others", we will have something such as Table 9-4, in which two tables on the top do not require any more split because the label for the Happy column is specified.

Chi-square Automatic Interaction Detector (CHAID)

Back in Chapter 3, we explained that the Chi-square test can be used to identify the relationship between two categorical variables, and here, we benefit from that attribute to split the tree.

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy
Gratitude	X	2	4	6	$\frac{(6 \times 7)}{12}$ 3.5	$\frac{(6 \times 5)}{12}$ 2.5
Gratitude	✓	5	1	6	$\frac{(6 \times 7)}{12}$ 3.5	$\frac{(6 \times 5)}{12}$ 2.5
Total		7	5	12		

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy
Compare himself with others	X	3	5	8	4.66	3.33
Compare himself with others	✓	4	0	4	2.33	1.66
Total		7	5	12		

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy
Expectation from others	high	0	4	4	$\frac{(4 \times 7)}{12}$ 2.33	$\frac{(4 \times 5)}{12}$ 1.66
Expectation from others	medium	4	1	5	$\frac{(5 \times 7)}{12}$ 2.91	$\frac{(5 \times 5)}{12}$ 2.08
Expectation from others	low	3	0	3	$\frac{(3 \times 7)}{12}$ 1.75	$\frac{(3 \times 5)}{12}$ 1.25
	Total	7	5	12		

Table 9-5: Chi-square calculation for the each feature.

Recall that the following equation calculates the Chi-Square value, in which O stayed for observed, and E stayed for expected values.

$$\chi^2 = \Sigma \frac{(O - E)^2}{E}$$

CHAID (Chi-squared Automatic Interaction Detector) primarily uses the Chi-square test for categorical variables to determine the best splits. For numerical variables, they often use the F-test to assess the significance of the splits. However, CHAID is less commonly used for purely numerical variables than categorical variables.

To calculate the splits in CHAID, the algorithm compares the Chi-square value of each feature (each feature is a column in tabular data) with the target label column. The feature with the highest Chi-square value is chosen for the split. In other words, first, the algorithm calculates the Chi-square for each feature. Then, it chooses the feature with the highest Chi-square and splits the table. This process continues until all remaining features can no longer be split, and thus, they are all considered to be the leaf nodes (no decision node remained).

Using data from the table in Figure 9-13 example, we need to calculate the Chi-square for each feature, including (Gratitude, Comp., and Exp.). These calculations are presented in Table 9-5.

We use the result of Chi-squares for each feature as follows and calculate the overall Chi-square for each feature.

Gratitude:

- Gratitude:No, Happy:Yes → $(2 - 3.5)^2/3.5 = 0.643$
- Gratitude:Yes, Happy:Yes → $(5 - 3.5)^2/3.5 = 0.643$
- Gratitude:No, Happy:No → $(4 - 2.5)^2/2.5 = 0.9$
- Gratitude:Yes, Happy:No → $(1 - 2.5)^2/2.5 = 0.9$

Total Chi-Square = 0.643 + 0.643 + 0.9 + 0.9 = 3.086

Compare himself with others:

- Compare: No, Happy:Yes → $(3 - 4.66)^2/4.66 = 5.91$
- Compare: Yes, Happy:Yes → $(4 - 2.33)^2/2.33 = 1.197$
- Compare: No, Happy:No → $(5 - 3.33)^2/3.33 = 0.838$
- Compare:Yes, Happy:No → $(0 - 1.66)^2/1.66 = 1.66$

Total Chi-Square = 5.91 + 1.197 + 0.838 + 1.66 = 9.605

Expectations from others:

- Expectation: High, Happy: Yes → $(0 - 2.33)^2/2.33 = 2.33$
- Expectation: High, Happy: No → $(4 - 1.66)^2/1.66 = 3.298$
- Expectation: Medium, Happy: Yes → $(4 - 2.91)^2/2.91 = 0.408$
- Expectation: Medium, Happy: No → $(1 - 2.08)^2/2.08 = 0.561$
- Expectation: Low, Happy: Yes → $(3 - 1.75)^2/1.75 = 0.893$
- Expectation: Low, Happy: No → $(0 - 1.25)^2/1.25 = 1.25$

Total Chi-Square = 2.33 + 3.297 + 0.408 + 0.561 + 0.893 + 1.25 = 8.739

Based on these results, the best feature to split the table is "Compare himself with others", because it has the largest Chi-square.

The CHAID algorithm creates a tree with the "Compare himself with others" as the root node. In this split, the algorithm deals with two subsets of the dataset; one subset has only "yes", and the other subset has only "no". These steps of spiting the dataset continue until all nodes are leaf nodes. You can do the rest independently, and it does not make sense to explain the rest of the algorithm in detail. Save your brain energy for more algorithms; still, two more decision tree algorithms remain, and then we will switch to boosting and bagging, which are extremely important algorithms and not easy to understand.

C4.5

C4.5 is developed by the author of the ID3 algorithm, and it is an extension of the ID3 algorithm. There is an open-source implementation of this algorithm called J48, which is the same algorithm but with a different name. C4.5 operates very similar to ID3, but it can handle missing data and continuous data, which are not very efficiently handled by ID3. The ID3 algorithm calculates the information gain to split the data, but C4.5 calculates the *gain ratio* instead. The

gain ratio is the information gain of a feature divided by split information. Therefore, instead of information gain, C4.5 calculates the GainRatio for all features and decides on the split based on the highest gain ratio. The gain ratio is computed as follows:

$$GainRatio(S, A) = \frac{IG(S, A)}{Split(S, A)}$$

Split of information is the sum of entropies that are calculated for each feature, as it is written as follows:

$$Split(S, A) = -\Sigma_{v \in values(A)} \frac{S_v}{S} log_2 \frac{S_v}{S}.$$

Here, S presents the dataset, and A presents the candidate feature, v is all possible values of A, S_v is the size of the subset of the dataset where A has v value.

Gratitude	Compare himself with others	Expecting from others	Happy
X	X	0.8	X
✓	✓	0.2	✓
✓	x	0.5	✓
✓	X	0.2	✓
X	✓	0.5	✓
X	X	0.8	X
✓	X	0.5	X
X	X	0.9	X
X	✓	0.4	✓
✓	✓	0.2	✓
X	X	0.5	✓
✓	X	0.8	X

→

Gratitude	Compare himself with others	Expecting from others	Happy
✓	✓	0.2	✓
✓	✓	0.2	✓
✓	X	0.2	✓
X	✓	0.4	✓
X	✓	0.5	✓
✓	X	0.5	X
X	X	0.5	✓
✓	x	0.5	✓
✓	X	0.8	X
X	X	0.8	X
X	X	0.8	X
X	X	0.9	X

Table 9-6: (left) The original dataset, (right) the dataset is ordered based on the "Excepting from others" which is continuous. Every time a single feature is selected for split information calculation the dataset should be ordered based on that feature.

Table 9-6 shows the table of Figure 9-13 with continuous data. In this example, we consider "Expecting from others" as continuous data. To implement the C4.5 decision tree on this data, first, we need to calculate the entropy of Happy for Table 9-6, as follows (7 happy and 5 not happy): $H(S) = -(7/12).log_2(7/12) - (5/12).log_2(5/12) = 0.96$.

Now that we have the entropy of “Happy", we should find which column has the largest split on the "Happy" column to calculate IG for each column (feature).

The following presents the gain ratio calculation for each feature. The sign # is used to refer to the frequency.

"Gratitude": yes, #rows: 6, #happy: 4, #not happy:2

$$H(S_{yes}) = -(4/6).log_2(4/6) - (2/6).log_2(2/6) = 0.918$$

"Gratitude": no, #rows: 6, #happy: 2, #not happy:4

$$H(S_{no}) = -(3/6).log_2(3/6) - (3/6).log_2(3/6) = 1$$

The IG[6] of "Gratitude" will be calculated as follows:

$$IG(S, Gratitude) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - (6/12).0.918 - (6/12).1 = 0.24$$

Next, the Split and then Gain Ratio of "Gratitude" will be calculated as follows:

$$Split(S, Gratitude) = -(6/12).log_2(6/12) - (6/12).log_2(6/12) = 1$$

$$GainRatio(S, Gratitude) = \frac{IG(S, Gradtitude)}{Split(S, Gratitude)} = 0.024/1 = 0.24$$

The IG of "Compare himself with others" will be calculated as follows:

$$IG(S, Compare) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - 0 - (8/12)0.954 = 0.324$$

Next, the Split and then Gain Ratio of "Compare himself with others" will be calculated as follows:

$$Split(S, Compare) = -(8/12).log_2(8/12) - (4/12).log_2(4/12) = 0.9$$

$$GainRatio(S, Compare) = 0.324/0.9 = 0.36$$

Now, we need to calculate the Gain Ratio for "Expecting from others", but it is a continuous attribute, and thus, we need to identify the maximum Gain Ratio for the value that can do the best split. Therefore, for each value (0.2,0.4,0.5, 0.8), we should calculate the Gain Ratio.

"Expecting from others" $<= 0.2$, #rows: 3, #happy: 3, #not happy:0

$$H(exp <= 0.2) = -(3/3)log_2(3/3) - 0 = 0$$

"Expecting from others" > 0.2, #rows: 9, #happy: 4, #not happy:5

$$H(exp > 0.2) = -(5/9)log_2(5/9) - (4/9)log_2(4/9) = 1$$

Based on the above results, the Gains ratio for Exp. for 0.2 will be calculated as follows.

$$IG(S, Exp\ for\ 0.2) = 0.96 - (3/12).0 - (9/12).1 = 0.21$$

$$Split(S, Exp.for\ 0.2) = -(3/12)log_2(3/12) - (9/12)log_2(9/12) = 0.8$$

$$GainRatio(S, Exp\ for\ 0.2) = 0.26$$

"Expecting from others" <= 0.4, #rows: 4, #happy: 4, #not happy:0

$$H(exp <= 0.4) = 0 - (3/3)log_2(3/3) = 0$$

---

[6] We do not write them here again, the IG of "Gratitude" and "Compare himself with others" come from the ID3 explanation.

"Expecting from others" > 0.4, #rows: 8, #happy: 3, #not happy:5

$H(exp > 0.4) = (3/8)log_2(3/8) - (5/8)log_2(5/8) = 0.94$

$IG(S, Exp\ for\ 0.4) = 0.96 - (4/12).0 - (8/12)\ .\ (0.94) = 0.33$

$Split(S, Exp\ .\ for\ 0.4) = -(4/12)log_2(4/12) - (8/12)log_2(8/12) = 0.9$

$GainRatio(S, Exp\ for\ 0.4) = 0.36$

Since your brain and our brain are about to explode, we do not continue further. However, the algorithm will do the rest for other values of the "Expecting from others" as well, including 0.5. 0.8 and 0.9. Then, it compares all gain ratios (all from "Expecting from others" plus gratitude and "Compare himself."). For example, assume $GainRatio(S, Exp\ for\ 0.8) = 0.67$ receives the highest gain ratio. Therefore, the decision tree starts with something like Figure 9-14, and continues until all nodes are leaf nodes.

Classification and Regression Trees (CART)

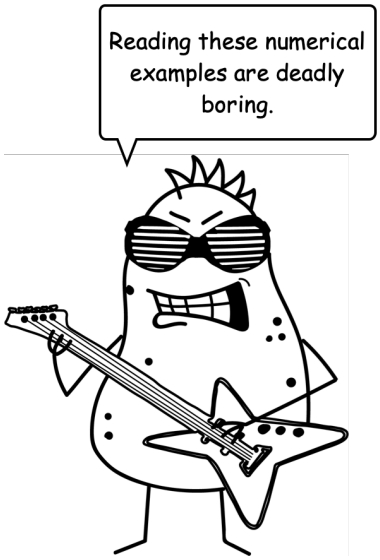
The CART algorithm is another popular decision tree algorithm. It constructs a binary tree (each node has only two branches), which can also handle continuous data appropriately. The CART algorithm uses a recursive approach similar to other decision trees to split the dataset into smaller units. CART uses *Gini index (Gini split* or *Gini Impurity)* to determine the best feature to split. Gini split quantifies the impurity of each feature. The Gini index is computed as $Gini = 1 - \Sigma_{i=1}^{n}(p_i)^2$, where p_i is the proportion of the samples that belong to class i in a particular node.

the total number of labels

frequency of records with label *i*

$$Gini = 1 - \sum_{i=1}^{n} (P_i)^2$$

The Gini index value varies between values 0 and 1, where 0 expresses the purity of the feature, i.e., all values belong to a specified class, or only one class exists for this feature, and 1 indicates the random distribution of elements across various classes. The algorithm calculates the Gini index for each feature and chooses the lowest one for the split.



It can handle continuous data in a similar fashion as C 4.5; check the example we explained for "Expecting from others" in the C 4.5 algorithm description.

Let's calculate the Gini index for the example presented in Figure 9-13. Similar to other algorithms, the algorithm

calculates the Gini index for each feature (column), and then it selects the feature with the lowest Gini index for the split.

Gini index of "Gratitude":

$$Gini(gratitude = yes) = 1 - (5/6)^2 - (1/6)^2 = 0.27$$
$$Gini(gratitude = no) = 1 - (4/6)^2 - (2/6)^2 = 0.44$$
$$Gini(gratitude) = 0.44(6/12) + 0.27(6/12) = 0.335$$

Gini index of "Compare himself with others":

$$Gini(Compare = yes) = 1 - (4/4)^2 - (0/4)^2 = 0$$
$$Gini(Compare = no) = 1 - (3/8)^2 - (5/8)^2 = 0.47$$
$$Gini(Compare) = 0(4/12) + 0.47(8/12) = 0.313$$

Gini index of "Expecting from others":

$$Gini(Exp. = low) = 1 - (3/3)^2 - (0/3)^2 = 0$$
$$Gini(Exp. = medium) = 1 - (4/5)^2 - (1/5)^2 = 0.32$$
$$Gini(Exp. = high) = 1 - (4/4)^2 - (0/4)^2 = 0$$
$$Gini(Exp. = high) = 0.32 + 0 + 0 = 0.32$$

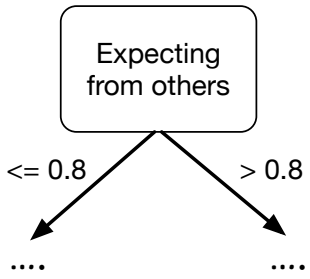


Figure 9-14: C4.5 decision tree result example "Expecting from others" for 0.8 has the highest gains ratio.

$Gratiude = 0.085$, has the lowest Gini index. Therefore, the tree starts the first split by using this feature; then, it continues constructing the branches with the same approach for each subset (one subset includes gratitude = yes, and the other subset includes gratitude = no).

Decision Tree Challenges and the Need for Pruning.

While working with decision trees, one might ask when we need to stop creating the tree. One answer is to define a minimum number of records under a leaf node; if the dataset remaining for a node has a size less or equal to this minimum number of records, then the algorithm should stop splitting the dataset and stop going deeper into that branch.

The second approach is to specify a maximum depth for a tree. Larger tree depths lead to more complex models that can capture detailed patterns but risk overfitting. Conversely, shallow trees might fail to capture essential patterns in the data, which can lead to underfitting. Thus, finding a balance between tree depth and complexity is crucial for building an effective decision tree model. Allowing each leaf to hold only one data point often leads to overfitting because the tree becomes too detailed, fitting the noise in the training data, which can decrease its predictive performance on new data.

If we plot the accuracy of the test and train

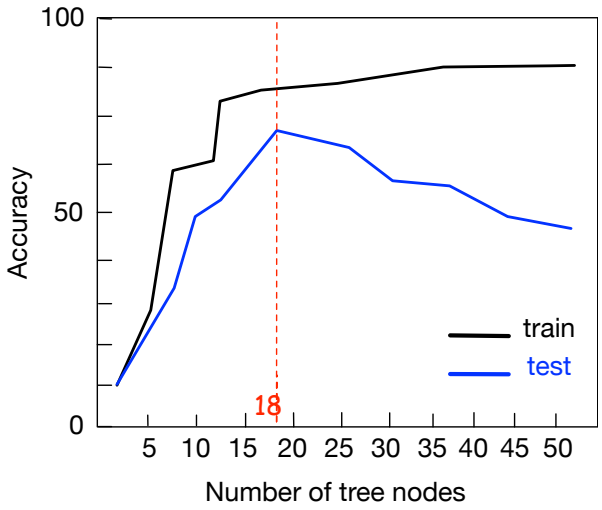


Figure 9-15: The overfitting problem in a tree. We can see that after the number of nodes increases to more than 18, test data accuracy decreases significantly, but training is still performing well.

datasets, we might end up having Figure 9-15, which is a demonstration of overfitting in a decision tree. This figure shows that after the 18th node in the tree, the test dataset accuracy starts to decrease. However, while working on the train set, we went to 50 nodes, but the accuracy didn't decrease. In other words, this problem shows that we should cut the tree before it becomes too specific for the train dataset (overfitting), and in this example, we should stop at the 18th node.

A third approach, which is a popular one, is to separate the "train" dataset into two parts: *train*, and *validation* sets (we have explained it in Chapter 1). We construct the tree by using the train set. Then, we test the tree using the validation set. Next, we remove each node and again measure the tree's accuracy (now a pruned tree) on the validation set. If the accuracy of the validation set improves, we do not incorporate the recently removed node anymore and will continue working with the pruned tree. Otherwise, if the accuracy decreases, we add the node back. Next, we remove another node, and we repeat this node removal accuracy test on the validation set until no further node removal improves the accuracy, and we stop removing any new nodes. Pruning stops when removing further nodes no longer improves validation set accuracy. In the end, we evaluate the tree with the data that it has not encountered before from the Test set.

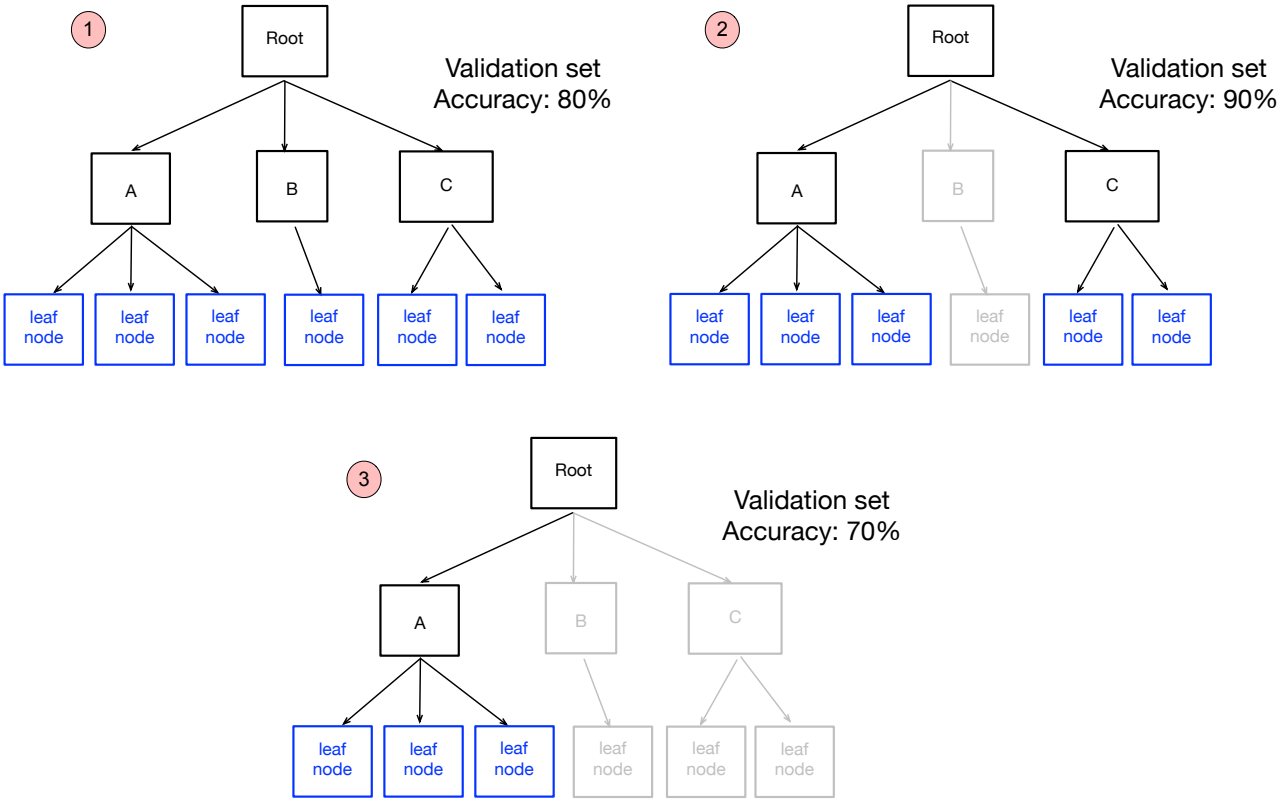


Figure 9-16: An example of pruning the original tree (1) and, as a result, removing branch B improves the accuracy (2), but removing another branch, branch C decreases the accuracy, so the algorithm returns (2) as the final result of pruning.

This process is similar to what we have explained about Sequential Backward Selection (SBS), which we have explained in Chapter 6. Note that while we are talking about nodes in this section, in the process of tree pruning, we do not remove leaf nodes, we only remove decision nodes.

To better understand pruning, look at the toy example in Figure 9-16 (1). Using the training set, the algorithm constructs the decision tree, and we experiment with its accuracy on the validation set. The accuracy is 80%. Then, we removed branch B and again tested the accuracy, it improved to 90%. That is good, so we keep the new tree, as shown in Figure 9-16 (2). From the new tree, we remove another branch, which is branch C. Now, the accuracy decreases to 70%, which is worse, as shown in Figure 9-16 (c). Therefore, we return to the pruned tree in Figure 9-16 (2) because it has the highest accuracy.

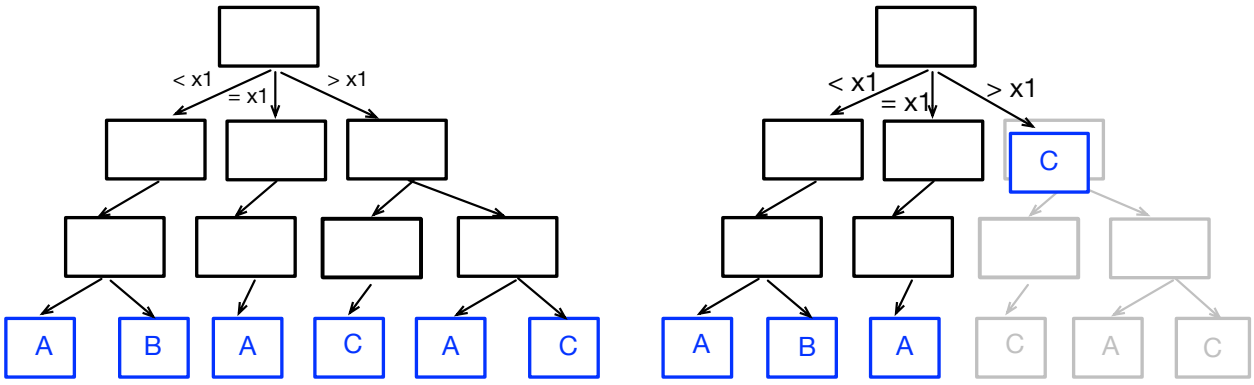


Figure 9-17: (Left) Original tree (Right) the pruned version of the tree, by substituting any branches under > x1 as C leaf node. Because most of the leaf nodes under this branch was C (two C and one A) before pruning.

The fourth approach is to prune a tree, which is fairly simple. It starts to remove a branch and substitute it with the most frequent leaf node. For example, take a look at Figure 9-17. Here, we have a branch $> X1$, and most of its leaf values are *C*. Therefore, the pruning process removes this branch (and its sub-branches) and substitutes one leaf with a leaf node with the value of C. In other words, it checks the value of the new record (test set). If it is $> X1$, then it assigns the *C* label to that record.

Decision Trees Computational Complexity

Assuming a tree has *m* depth, and it is constructed based on *n* number of data points, the computational complexity of running a tree is *O(m)* and always $m < n$. The complexity of the decision tree increases exponentially as the tree goes deeper. Géron [Géron '19] explained that balanced decision trees have the computational complexity of $O(log_2(m))$ for the test set and $O(n \times log_2(m))$ for the train set.

Perhaps you might think that decision trees are similar to rule-based classifiers. That is correct; decision trees generate rules, but instead of feeding the rules ourselves, they discover rules on their own.

Table 9-7 summarizes the decision trees we explained here and their usage. If you intend to decide on a decision tree to use, it is better to test them on your dataset and choose the one that has the highest accuracy and/or resource efficiency.

Algorithm	Split Function	Usage / Characterisitcs
ID3	Information Gain	It is used for categorical data and not numerical data.
CHAID	Chi-Square	It can produce multiple branches, and since it uses Chi-Square test, it is good for descriptive analysis.
C4.5	Gain Ratio	it constructs a tree based on rule sets. It can handle missing data and can be used for both continuous and discrete data.
CART	Gini Split	it constructs data based on numerical splitting; it is binary but allows more than one choice via one vs others labeling policy. It is not very useful when the number of classes is large.

Table 9-7: Decision tree algorithms, their splitting method and usage / characteristics.

NOTES:

* Decision trees can be used for continuous, categorical, and discrete datasets. Besides, they are a good choice for datasets that include missing variables, and we do not need to invest heavily in removing or reconstructing missing data. Besides, decision trees are not good at handling duplicates, and it is better to remove duplicates and then feed the dataset into a decision tree for classification. Decision trees can handle duplicates in the sense that they will still function, but duplicate data can bias the tree's structure toward the over-represented examples.
* We can say that the decision tree is nothing but a set of rules that assign the label to the training data point based on that rule. It is a useful feature of the decision tree, and decision tree algorithms are explainable. At the time of writing this chapter (the first time in 2020), there is a movement to have explainable algorithms, i.e., *xAI*. This is due to the fact that black box algorithms, such as deep learning algorithms, are not explainable and, thus, not reliable for many applications, especially medical applications that make sensitive decisions.
* A problem in the decision tree is repetitions. Repetition occurs when an attribute is tested along the tree more than once, e.g., at level *n*, *salary > 100k,* and then again at level *n+3, salary > 100k*. This problem can also be mitigated by some condition checks (If-Then command).
* Another problem that happens in decision trees is replication. At some level, a duplicate subtree might be created, and considering the fact that a tree is usually loaded into the memory,

this is very inefficient. This problem can also be mitigated by some conditional checks, i.e., rule-based classifiers or condition checks (If-Then command).

* ID3, C4.5, and CART algorithms use greedy search methods. Greedy algorithms perform heuristic searches, and they might get stuck in local optima and not reach the global optima (Check Chapter 8 to recall global optima). As we have explained before, greedy algorithms are the algorithms that make the best short-sighted decisions. For example, we have the following two paths to get from A to D. *Path 1: A 3 steps to B, 99 steps to C, 999 steps to D*. *Path 2: A 5 steps to B, 6 steps to C, 5 steps to D*. Path 1 is longer than Path 2, but the greedy algorithm, choose Path 1, because the 3 steps are less than 5 steps at the beginning (short-sighted vision).
* In some literature specifying the depth of a tree as a threshold or the number of branches as the threshold for a decision, the tree is called tree regularization. We prefer to be parsimonious with these terms, but they are not wrong.

Ensemble Learning Methods

If you think you have already got rid of the decision trees, please accept our sincere condolence; we still have a long way to go with decision trees. All classification methods we have described until now are useful. However, in a real-world setting, we usually do not use one method, and most of the time (not always), we use ensemble learning or ensemble methods.

Ensemble learning refers to using more than one classification algorithm to perform the classification task, i.e., deciding on labels of the test dataset. Classification algorithms that we have explained until now are called weak learners or vanilla learners in the context of ensemble learning models. They are not weak, but combining them builds a stronger classifier, making them more accurate, and this is the objective of ensemble learning.

Some ensemble learning methods resemble human behavior while seeking a consultation. Assume you ask a person's opinion about a product. What that person said is probably correct, but if you ask for the opinion of more people, you get a better understanding of that particular product. Ensemble learners do the same in the context of machine learning, and they lower error rates and reduce the risk of overfitting. However, it is impossible to always use ensemble learning methods due to their resource utilization; they are not good for low-power or devices with limited computational capabilities.

There are three types of ensemble learning: Bagging [Breiman '96], Boosting [Schapire '90], and Stacking [Wolpert '92]. In the following, we describe these ensemble learning methods. Next, we focus on state-of-the-art classifiers that are based on gradient boosting decision trees.

Bootstrap Aggregating (Bagging)

Bootstrap Aggregating (Bagging) methods start by choosing *more than one random subset* from the training set and building a model for each subset (Figure 9-18). As a result, we have a set of models, and in the end, they are combined together. Based on majority voting or averaging the

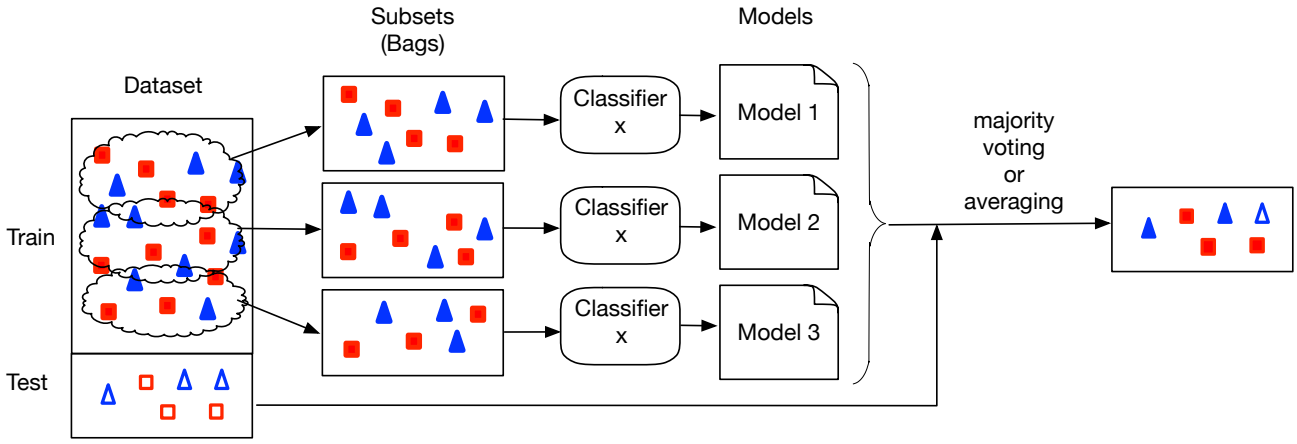


Figure 9-18: A toy example that presents how the bagging algorithm works. In the context of ensemble learning, a single classifier is called a weak learner. Note that all classifiers should be the same algorithm.

candidate labels, which are provided by each classifier, the final label will be assigned. Bagging reduces the variance because of the combination of different models.

Figure 9-18 visualizes how bagging is working. Here, we have a dataset, and as the first step, the algorithm starts with the original dataset and creates multiple subsets through a process known as *bootstrap sampling*. Bootstrap sampling involves repeatedly drawing samples from the original dataset, with each draw returning the data point back to the dataset, allowing it to be selected again. This means that after a data point is selected, it can be chosen again in subsequent samples. As a result of this sampling method, each bootstrap sample is likely to contain duplicates of some data points while missing others. On average, each bootstrap sample will include about 63% of the unique data points from the original dataset. These subsets are referred to as bootstrap samples or *bags of data*.

Then, as we can see in Figure 9-18, each of these subsets will be fed into a classifier. As a result, we will have three models, each from the same classifier algorithm. Afterward, it uses the majority voting or averaging from models to determine the label of new data points or test set data points.

Despite the usefulness of vanilla decision trees, they suffer from high variance. It means that if we separate the training dataset into two parts and train a decision tree for each part, the two parts could have two very different decision trees. This shows the vulnerability of the decision tree to variance. We can use a Bagging method (e.g., Random Forest) to reduce the risk of variance. The result of bagging usually has higher accuracy than each algorithm alone. The input parameters that a bagging algorithm receives include the number of subsets, the number of data points in each subset, and the classification algorithm.

Bagging is a *bootstrapping* method. In the English language, bootstrapping is an expression that metaphorically describes the act of achieving something without external help, often overcoming seemingly impossible odds. In the modern era, it is referred to processes that rely on self-

resources and not outside resources. In the context of statistics, bootstrapping is a metric of sampling that uses random sampling with replacement and assigns an accuracy score to sampled data.

Boosting

Boosting is another type of ensemble learning method in which each weak learner (a decision tree) is associated with a weight. To better understand the intuition of weights in boosting, let's talk about the example we described earlier. We explained that to buy a product, we could consult with several other recommenders. In the context of boosting, we assign weight to each recommendation based on the expertise of the recommender. If the recommender is an expert, the weight is high, and the non-expert recommender gets a low weight. These weights affect the average of opinions we get for a product.

Boosting is a sequential process that applies a classification algorithm to the entire train dataset and then makes a model. Next, it collects the errors (mislabeled data points) of the previous model and makes a smaller dataset from errors. Then, it uses the same classification algorithm (similar to Bagging) on the dataset that includes errors and creates another model. This process continues in several iterations until a maximum number of iterations (hyperparameter) is reached. Each Boosting model depends on the previous model. In other words, each new model focuses on the errors of the previous round. The models that perform better will receive higher weights, and the model that performs weaker will receive lower weights.

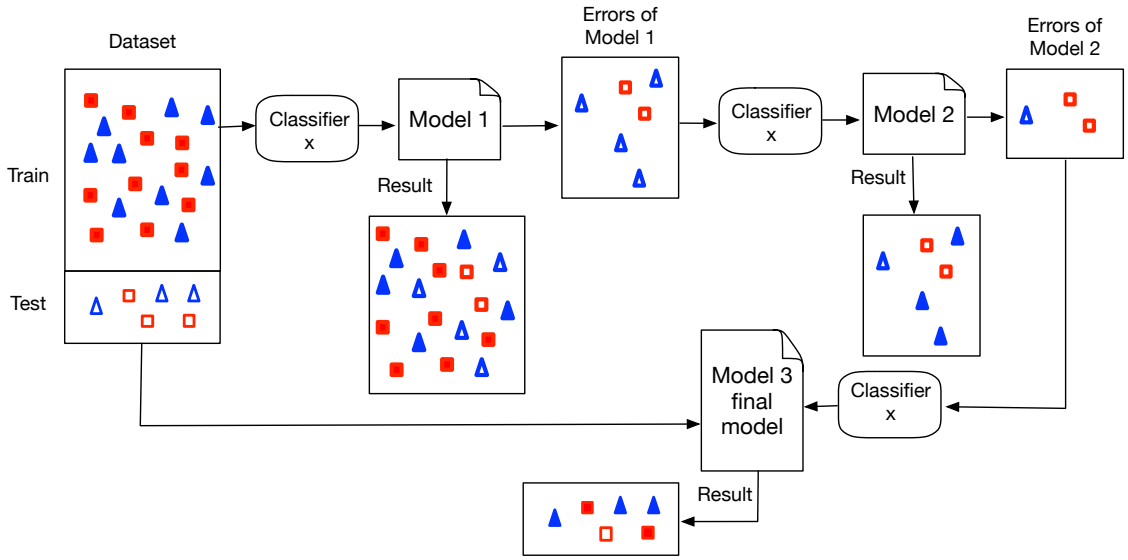


Figure 9-19: A toy example that presents how the boosting algorithm works. Note that the test dataset uses the final model, which is the strongest model.

Figure 9-19 shows the process flow of the Boosting algorithm. The objective is to label the unlabeled data, and in each round, a new dataset is constructed from the model's errors and fed into the next model as an input dataset.

Boosting algorithms usually receive three hyperparameters: (i) the number of decision trees to construct, (ii) the shrinkage parameter (which is a type of regularization and it is used to slow down the learning and thus reduce overfitting), and (iii) the number of splits in each tree. In general, Boosting decreases the bias (bagging decreases the variance). However, it provides better accuracy than bagging, but it also tends to overfit the training data as well. Therefore, hyperparameter tuning is required to avoid overfitting.

Stacking

Unlike Bagging and Boosting, Stacking applies 'different' weak learners in parallel and combines their results (base models) by training a *meta-model*. Meta-model or meta-learner aims to minimize the overall prediction error of weak learners. Previously described ensemble learning methods, i.e., Bagging and Boosting, are homogenous (they use one weak learner), but stacking is heterogeneous (different classification algorithms), and the output of each model will be used to train a meta-model. Usually, meta-model classifiers are simple regression algorithms, such as linear regression for regression tasks or logistic regression for classification tasks. However, we could use other algorithms as well.

The main advantage of stacking is diversity among different models, which allows different patterns to be extracted from the underlying dataset. Both Bagging and Boosting are usually used for decision trees, but stacking uses a variety of weak learners, which makes it powerful to benefit from the diversity of different classifiers.

Figure 9-20 visualizes the process of stacking. Usually, it has two steps. The first step, which we call level 0, involves using a mixture of classifiers. The next step, i.e., level 1, includes constructing a meta-model classifier that can accurately classify the test dataset.

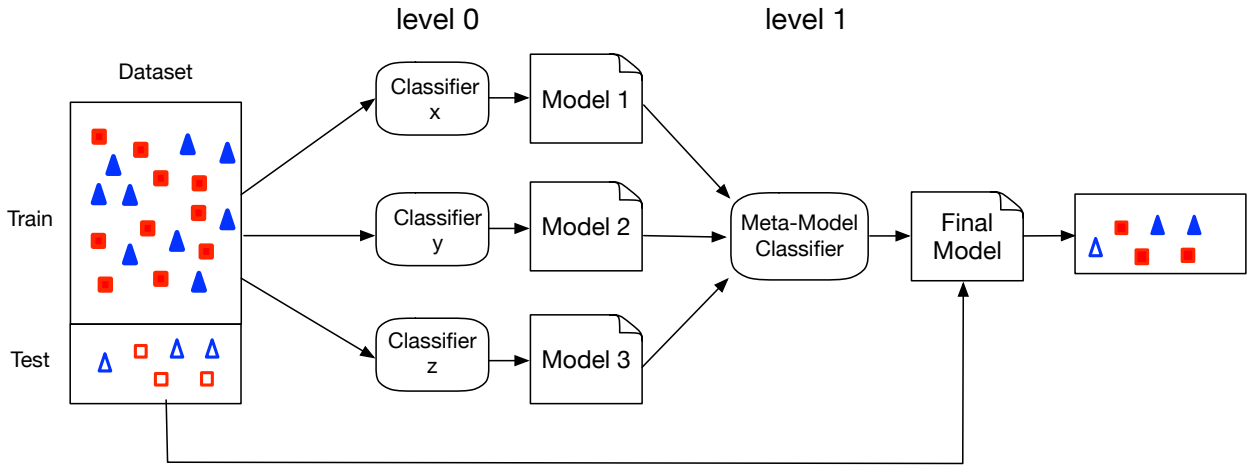


Figure 9-20: A toy example that presents how the stacking algorithm works. The first classifiers are called level 0, and the second classifier is called level 1.

Random Forest

Random forest [Breiman '05] is the most popular Bagging algorithm that uses decision trees. Decision trees are very good because they are explainable, but they suffer from weak accuracy. The random forest algorithm resolves the accuracy problem of decision trees by constructing multiple decision trees during the training phase. Random forests can reduce the variance of decision trees by averaging their predictions from different subsets of the data.

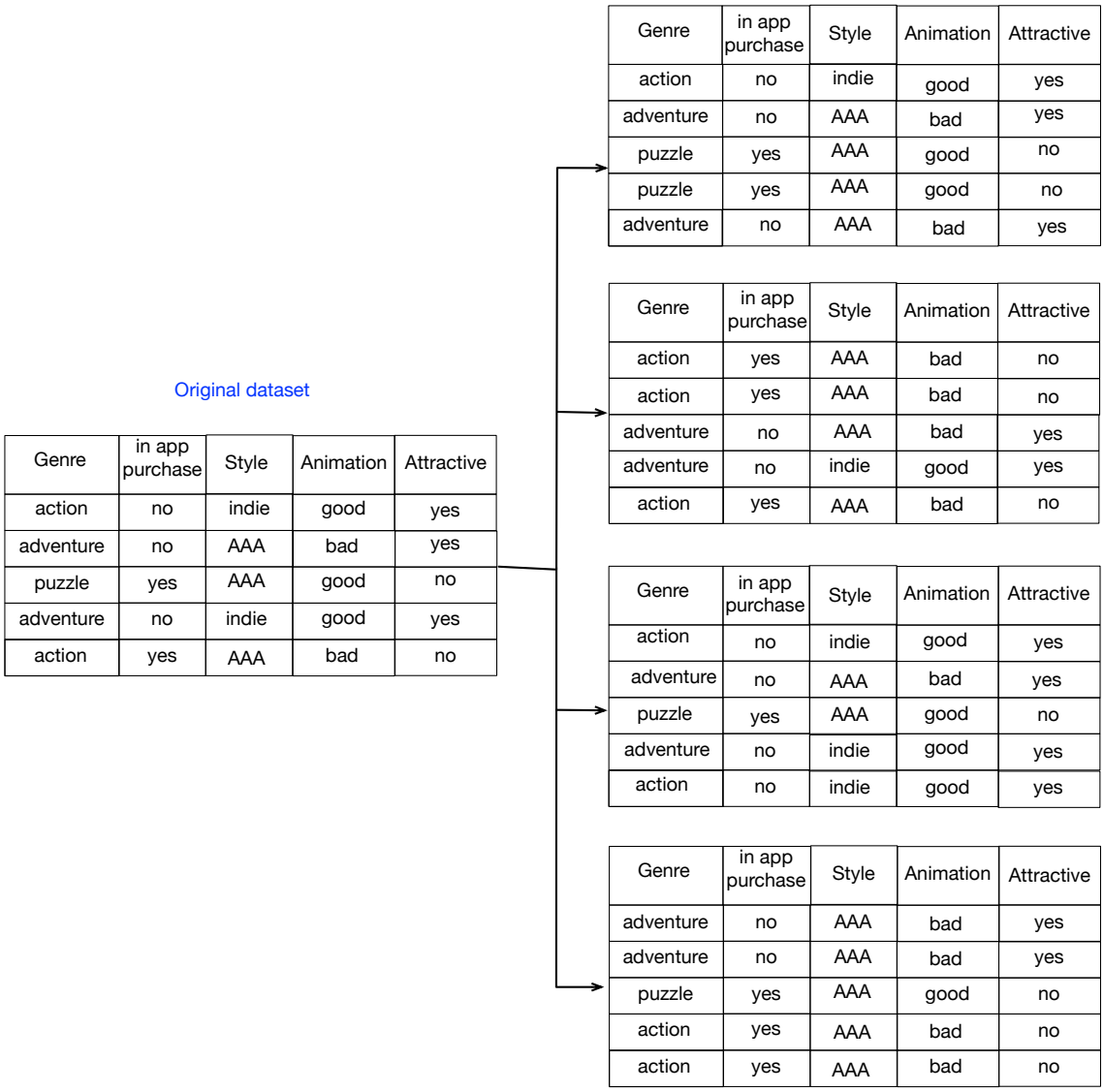


Figure 9-21: From the original dataset, a set of bootstrap datasets (subsets) are created. The size of the subsets is similar to the original dataset, but each of them contains about 70% of the original data. The rest of their data points are duplicates.

To understand the algorithm, we follow our sacred tradition and begin with an example. Assume you finished this book and since a big worry of your life, which is learning artificial intelligence and machine learning concepts, has been resolved. Now, you have started to play some games. A

recommendation algorithm intends to suggest games to play. The history of your game playing is shown as a table on the left side of Figure 9-21. It is a dataset of some games, and you have labeled them as attractive to you or not.

(i) As the first step, the random forest algorithm starts by creating subsets from the original dataset, similar to other Bagging algorithms; each subset includes only 63% of the original dataset samples, as it has been shown in Figure 9-21. These subset datasets are called *Bootstrap datasets*, and the algorithm randomly selects some records from the original dataset to construct them. Since only 63% of the original dataset existed in each bootstrap (subset) dataset, the part of the dataset that does not exist in the bootstrap dataset is called the *Out-Of-Bag (OOB) dataset*, i.e., $100 - 63 = 37\,\%$.

(ii) In the second step, assuming we have a d number of features, The random forest selects k ($k < d$) number of features from each subset, and it uses these k features to calculate the split. In traditional decision trees, we use all features (d), but here, we use only k features from d possible features. Usually, the random forest uses $k = \sqrt{d}$ equation to decide on the k value. For example, in Figure 9-22 $d = 4$ ('Attractive' column is the label, and we didn't count it as a feature), thus we assume k = 2. For each dataset, we select k random number of features. Selected features are shown in blue color in Figure 9-22. For example, if $d = 8$, we get the closest fit, which is $k = 3$. In other words, unlike vanilla decision trees, we only use a random number of features (columns) and not all of them. This contrasts with decision trees, where the best feature for splitting is chosen from all available features rather than a subset. Here, however, the split is selected from a smaller set of features, not all of them.

A question might arise: why did we select k features from the d number of features? The answer is that if one single feature contributes significantly to a prediction, it will be used in the top split of all decision trees. By changing the split of trees, random forest creates highly uncorrelated trees, thus reducing the overall model variance and preventing overfitting.

(iii) In the third step, the test dataset (or test record) will be fed into all trees, and then its labels will be decided by averaging the labels of the other decision trees. It will lead to having a low variance while getting a high accuracy.

At the end of the training phase, we have a set of uncorrelated decision trees. For the inferences, we easily feed them into the set of decision trees, and each of these trees assigns a label to the new data point. The algorithms perform a majority voting and decide the final label based on votes, as shown in Figure 9-23.

The Random forest has two hyperparameters to configure, k and m (in addition to the decision tree algorithm that is usually fixed). k is the number of features to select for comparison and split

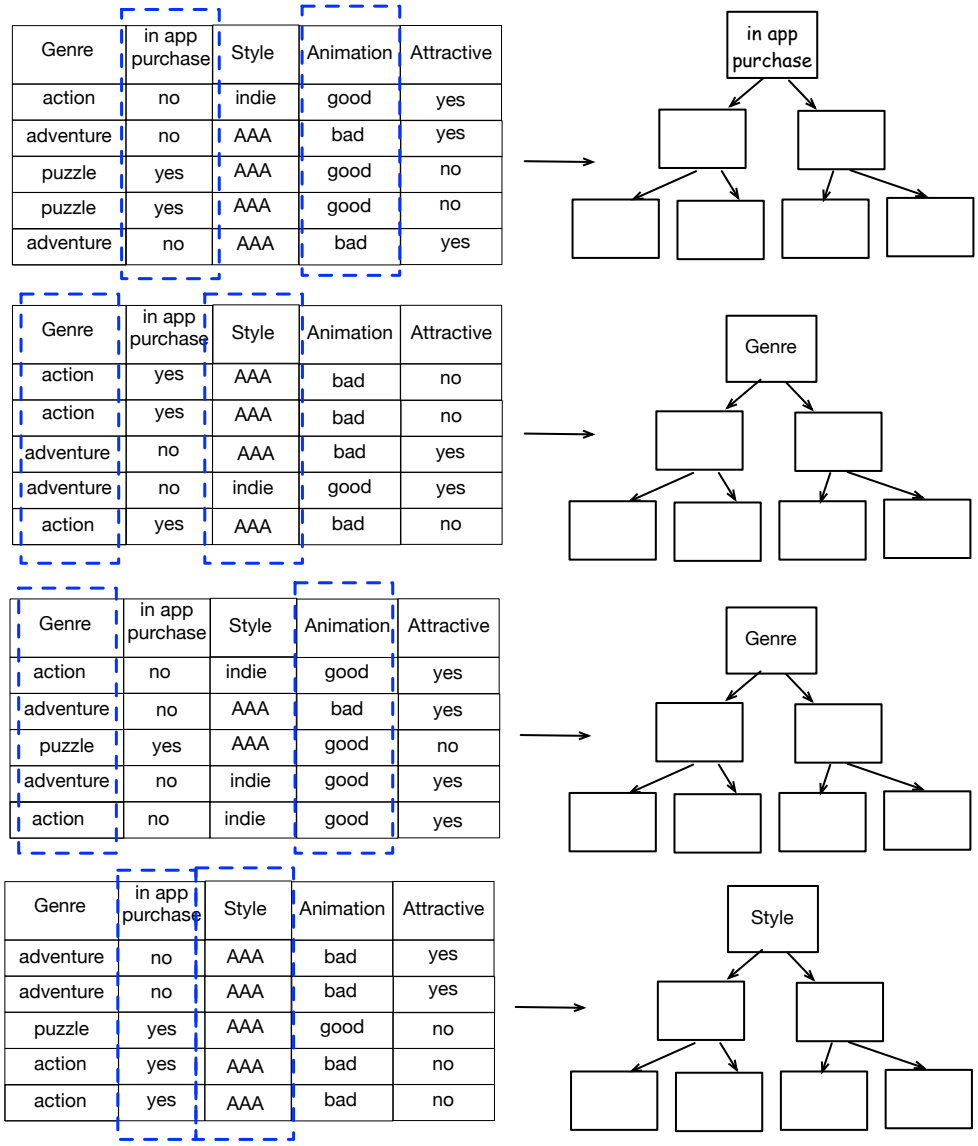


Figure 9-22: *k* number of features from each select is selected (marked in blue dots), and the split for each decision tree is selected based on analyzing these k features, unlike decision trees, which use all features.

selection ($k = 2$ in our example), and m is the number of subsets created from the original dataset ($m = 4$ subsets in our example).

We have explained that while creating the Bootstrap dataset, the records or data points that are not included in the subsets are called Out-of-Bag (OOB) samples. For example, if the original dataset is composed of $[a, b, c, d]$ and one of the Bagged datasets is composed of $[a, b, d]$, we can say $[c]$ is an OOB sample. Similar to the Bootstrap samples, each OOB sample set could be passed through its related decision tree algorithm, constructed without looking at OOB samples, and the prediction error of OOB can be calculated. This error is called the *Out-of-Bag error*. This

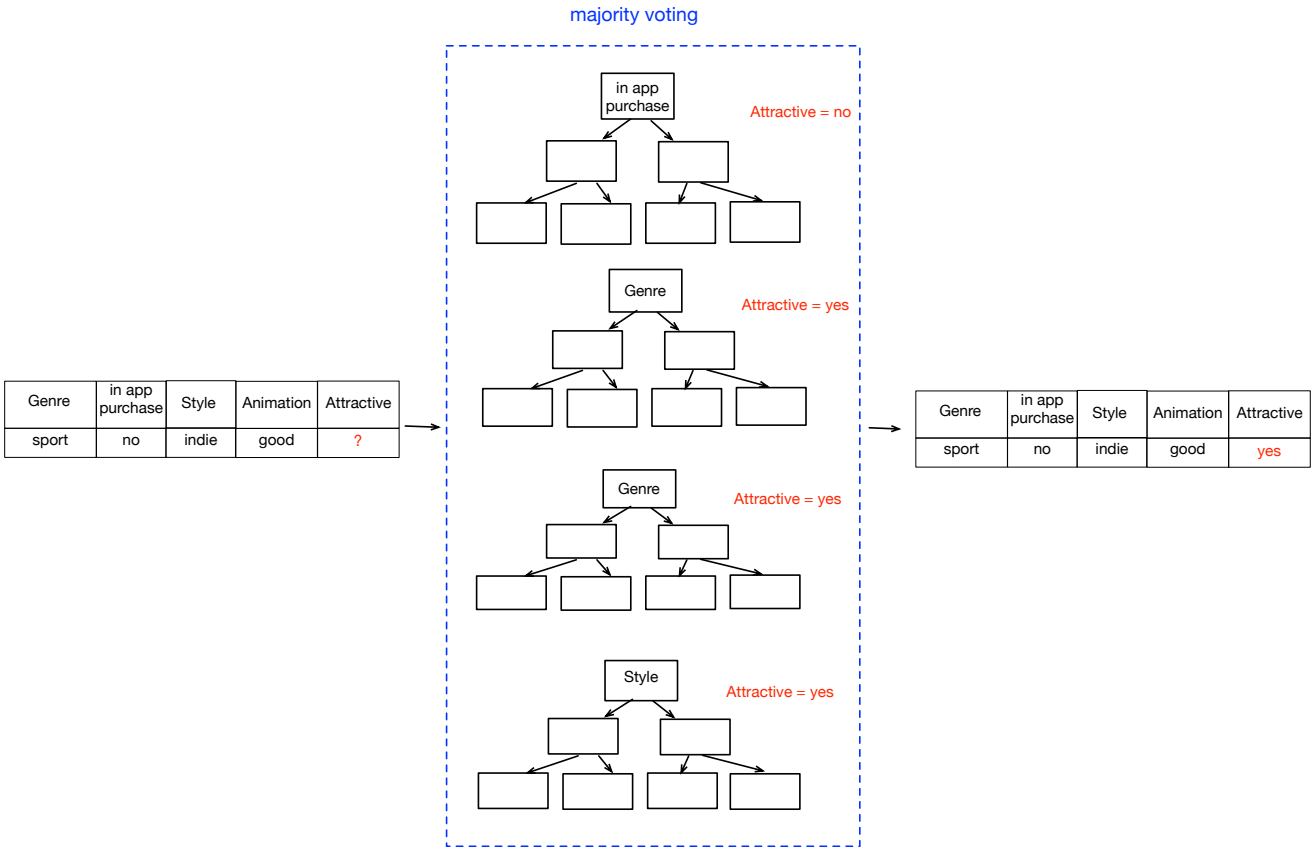


Figure 9-23: A test data which has only one record is fed into all decision trees. Each of them is assigned a label for the "Attractive" feature and at the end based on majority voting the final label will be assigned.

error could also be used for the validation of the dataset (in addition to the test dataset). It means that we are not only relying on a test dataset for validation, but we can also use OOB datasets as well. As a result, we will have a very good estimate of the Random forest model. This is especially useful if we have a small training dataset.

Assuming m is the tree depth and n is the number of data points, we explained the computational complexity of the tree is $O(n \times log_2(m))$. The Random Forest creates m' numbers of decision trees, so the computational complexity of training is $O(m' \times n \times log_2(m))$ and testing will be $O(m' \times log_2(m))$.

Adaptive Boosting (AdaBoost)

AdaBoost [Freund '95] is an ensemble learning algorithm based on Boosting that uses decision trees as its weak learners. To learn AdaBoost, we should first learn some concepts, which we describe below.

- Similar to other boosting algorithms, AdaBoost combines multiple weak learners into a single strong learner. It focuses on errors in each iteration and improves the classification result by assigning higher weights to errors and lower weights to correctly classified data points.

- Similar to Random Forest, it uses a set of decision trees. However, since these trees are very short in depth (usually only one level deep), they are called *stumps*.
- Each data point (e.g., a record in table format) is associated with weight, and at the beginning, this weight is equal for each data point. The sum of weight at each iteration/level should equal one. Assuming we have *n* number of data points, the first stumps have their weight calculated as follows: $weight = 1/n$. If, in an iteration, the data is classified incorrectly, then its weight increases. Otherwise, its weight decreases (for correctly classified data points) because the sum of weights should be equal to one at each iteration/level.
- For each stump after the first level, the *error rate* is calculated as the sum of the weights of the misclassified instances divided by the total sum of weights. For example, if we have 5 data points, each with an initial weight of 1/5, and 3 of them are misclassified, the error rate would be 3/5 , assuming all weights are still equal.
- The result of an AdaBoost algorithm is a set of classifiers (stumps), each associated with a weight.

Standing and Clapping Hand (C1)	Move too many Body parts (C2)	Keep Smile (C3)	Good Dance Performance (Actual)	Good Dance Performance (Predicted)
No	No	No	No	No
No	Yes	Yes	No	No
No	Yes	No	Yes	Yes
No	No	Yes	Yes	Yes
No	Yes	No	No	No
Yes	Yes	Yes	No	Yes

Table 9-8: A sample dataset of dancing, the prediction label will be "Good Dancer".

Now that we have learned these preliminary concepts let's use an example to explain this algorithm. Dancing is a part of the culture in many regions, especially in Africa, Asia (East, South, and West), South and Latin America, etc. People gather together in ceremonies and parties and dance together.

Assume that after you finish this book, you are in charge of an AI corporation building dancing robots for parties. Your robots will learn to dance by looking at people dancing at parties. By looking at the people, the robot constructs the dataset presented in Table 9-8 along with our labels (Good Dance Performance). The output or prediction variable we are looking for labels is "Good Dance Performance". For the sake of simplicity, we consider all features binary.

Adaboost algorithm performs the prediction in the following steps:

(i) The algorithm assigns equal weight to each record, which is *1/n* of the total *n* records (See Table of Figure 9-24). Then, it uses each feature alone and creates a decision tree with a depth

C1	C2	C3	Actual	Predicted	Level 0 Weight
No	No	No	No	No	1/6
No	Yes	Yes	No	No	1/6
No	Yes	No	Yes	Yes	1/6
No	No	Yes	Yes	Yes	1/6
No	Yes	No	No	No	1/6
Yes	Yes	Yes	No	Yes	1/6

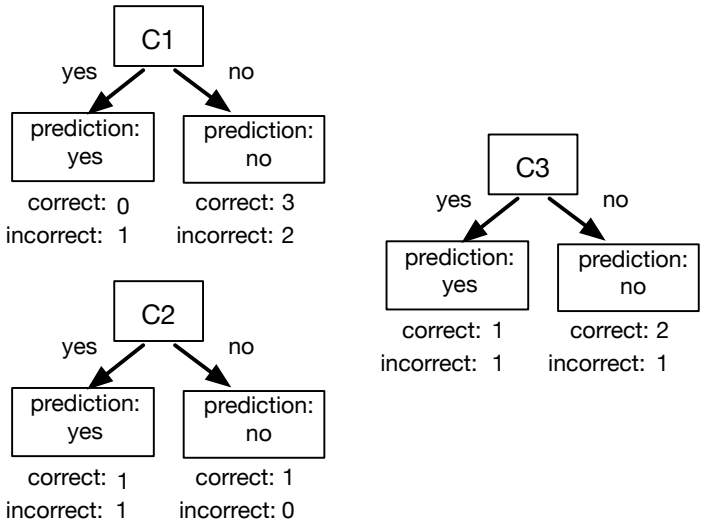


Figure 9-24: First each record receives an equal weight. For each feature a stump is created, the number of its correct vs incorrect instances is written at the bottom of each stump.

equal to one for each feature. In our example, we have three columns as features, and thus, we have three stumps. Keep in mind that the number of stumps is equal to the number of features. For example, if we have *m* features, AdaBoost will create *m* stumps.

(ii) Entropy, Gini index, or another tree split method is used for each stump to find the best stump that can split the data. We do not write them for the sake of brevity, but let's say the lowest entropy belongs to C3. Therefore, the AdaBoost algorithm selects the C3 stump.

(iii) Now, for the selected stump (C3), the algorithm should calculate the α_t, known as *stump performance score*, *significance,* or *stage value* of the stump by using the following equation. Here, *t* presents the index of the current iteration.

$$\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$$

In this equation ϵ_t is the total error, which is the sum of weights for the misclassified records in the stump by total records in that stump, which is as follows:

$$\epsilon_t = \sum \text{(misclassied records in the stump / total records)}.$$

Based on Figure 9-24, in stump C3, we have one incorrect record, and thus $\epsilon_t = 2/6$. Now, we can calculate the α_t for stump C3 as follows:

$$\alpha_t = \frac{1}{2} ln(\frac{1-(2/6)}{(2/6)}) = 0.3465$$

(iv) Now that we have α_t of this stump, we can add it to the final model. The final model $H(x)$ is calculated as the sum of stump performance times each stump:

$$H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$$

Therefore, we have $H(x) = 0.3465$ ($Stump(C_3)$) as the result of the first level (*Stump (C3)* is a function).

(v) The next level starts by updating weights. The following equation describes how the algorithms calculate the new weights (weight at level *i*).

$$w_i = \begin{cases} w_{i-1} \times e^{-\alpha_t} & for\ correctly\ classified\ record \\ w_{i-1} \times e^{\alpha_t} & for\ incorrectly\ classified\ record \end{cases}$$

Based on this equation, we calculate the weights of the second level and add them to the table, as shown in Table 9-10. For example, for the last record, the prediction says 'Yes', but the actual data says 'No'. In this case, assume that two records are incorrectly classified by Stump C3 (the last record), which its new weight will be $1/6 \times e^{0.3465} = 0.235$. Others were correctly classified, and they will be $1/6 \times e^{-0.3465} = 0.117$. These numbers help us to identify values for the "Weight Level 2" column.

Nevertheless, the sum of weights should be equal to one, but they are not; therefore, the algorithm normalizes them by summing them all ($5 \times 0.117 + 1 \times 0.235 = 0.82$) and dividing each weight by the sum. This helps us to populate the "Level 2 weight (normalized)" column in Table 9-9.

C1	C2	C3	Actual	Predicted	Level 1 weight	Level 2 weight	Level 2 weight (normalized)
No	No	No	No	No	1/6	0.117	0.14
No	Yes	Yes	No	No	1/6	0.117	0.14
No	Yes	No	Yes	Yes	1/6	0.117	0.14
No	No	Yes	Yes	Yes	1/6	0.117	0.14
No	Yes	No	No	No	1/6	0.117	0.14
Yes	Yes	Yes	No	Yes	1/6	0.235	0.264

Table 9-9: A sample dataset of dancing, with the old and new weights.

(vi) Now, a new dataset with the same size as the original dataset will be created based on the coefficients of the most recent weights (Level 2 weights). In other words, new dataset records have their distribution based on weights. Since we have 0.286 ~ 30%, we can say 30% includes this weight error, and in the new table, there will repeated at 30% of records. Therefore, we will have something like the table in Figure 9-25. The blue records in this table are the ones that are substituted by the record that has a weight of 0.286, and other records (black ones) remain unchanged. Randomly, we selected one record to substitute it, and as a result, the blue colors are 30% of the entire dataset.

C1	C2	C3	Actual	Predicted
No	No	No	No	No
No	Yes	Yes	No	No
No	Yes	No	Yes	Yes
No	Yes	No	No	Yes
No	Yes	No	No	No
No	Yes	No	No	Yes

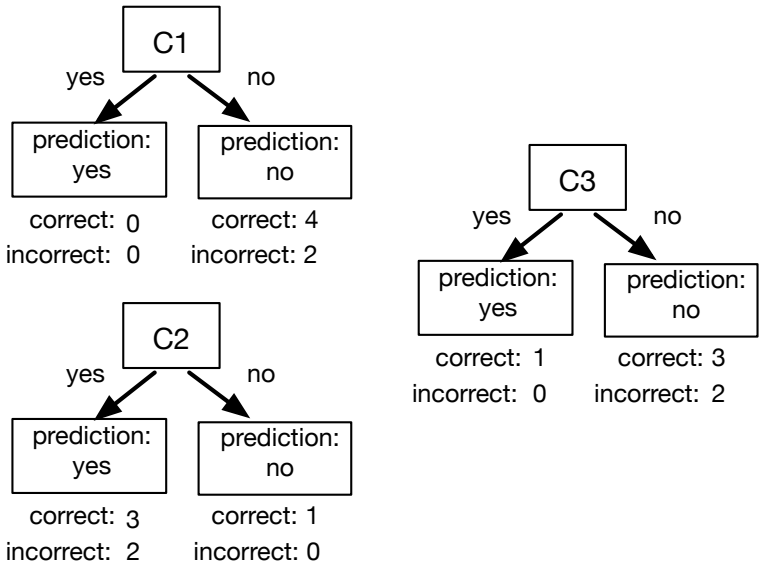


Figure 9-25: A new dataset created based on the normalized level 2 weights. Blue ones are because of weight 0.087, and because of that the algorithm creates multiple copies of that record. Then, based on the new dataset, again for each feature one stump will be created.

(vii) Next, the algorithm repeats from step (i) and creates one stump per feature from the Table in Figure 9-25. There will be three stumps, as it is shown in Figure 9-25.

The process from step (ii) will be repeated. Let's say Entropy, Gini index, or another tree split is used and select stump C1. At the end of step (iii), after the stage value (α_t) has been identified, the model can be refined. For example, assuming in this iteration, C1 has the lowest entropy (in this example, all have the lowest entropy, but we assume C1 has the lowest one), and we suppose we get its $\alpha_t = 0.622$, the final prediction model will be written as follows:

$$H(x) = 0.804(Stump(C_3)) + 0.622(Stump(C_1)$$

By looking at the result, i,e, $H(x)$, we can summarize that AdaBoost is a combination of several stumps (weak learners), and each stump is AdaBoost adjusts the weights of the training instances based on the errors made by the previous stump.

AdaBoost receives several hyperparameters. One hyperparameter is called the "number of iterations", *T*. This iterative process will be repeated *T* times. In other words, the algorithm iterates until the number of iterations is reached. Also, some AdaBoost implementations receive the "number of trees (stumps)" as an input parameter as well. Obviously, the number of trees should be lower than the number of features, and the algorithm selects them randomly. To identify the optimal number of stumps, we need to perform hyperparameter tuning. Some implementations also allow specifying a bit deeper than one-level trees. Therefore, another hyperparameter of AdaBoost could be the "depth of tree". Another parameter is the "learning rate", and it usually has a predefined value (e.g., 0.1). It could be used to shrink the contribution of each weak model in the final model, but it is not required in all implementations.

Assuming m is the number of stumps, n is the number of data points, and p is the number of features (columns) the computational complexity of AdaBoost is $O(m \times n \times p)$ for training and $O(m \times p)$ for testing.

NOTES:

* Bagging is robust to noise and outliers; boosting can indeed be sensitive to noise and outliers because it tries to correct misclassifications in each round, potentially leading to overfitting on the noise.
* Decision trees are the most common algorithm used in Bagging. Because they have high variance, bagging is helpful in reducing the variance of classification algorithms.
* Since boosting focuses on misclassified data points, the risk of overfitting increases.
* The term Bootstrapping is used in statistics as well. Bootstrapping is a statistical method used to estimate the population's statistical characteristics (e.g., distribution) by sampling a dataset with replacement.
* Random Forest enables us to measure the importance of each feature based on its contribution to tree split (e.g., impurity), which makes it an attractive algorithm to use for feature selection as well. In particular, per feature, we can calculate the impurity of the tree and then rank features based on their impurity to select the best feature that can assist us in the prediction or labeling task.
* While using the Random Forest algorithm, it is possible to make tree creation more random by using a random threshold for splitting trees. As a result, such a set of trees (forest) is called an *extremely randomized forest*.
* While Random Forest is generally robust against overfitting due to its averaging method, AdaBoost can provide high accuracy but might be prone to overfitting if the noise is present.

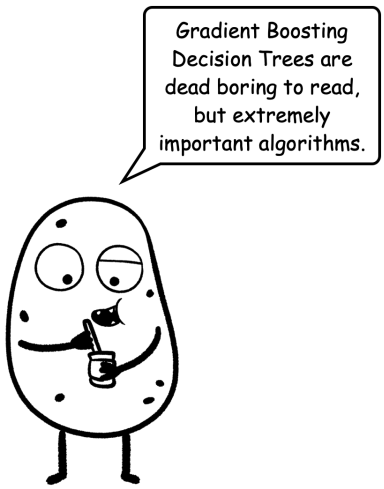
Gradient Boosting Decision Tree

At the time of writing this chapter, there are online communities, such as Kaggle's[7] online machine-learning competitions. Most of the awards go to the participants who use Gradient Boosting Decision Tree (GBDT) algorithms. GBDT algorithms have higher accuracy than traditional classification algorithms and, for tabular data, perform better than Deep learning algorithms (we will explain them in upcoming Chapters 10, 11, and 12). Nevertheless, unlike deep learning algorithms, they do not need a large dataset to operate. Therefore, if the dataset is not large enough and we can experiment with gradient boosting algorithms, it is a wise decision.

We start this section by explaining the basic form of the gradient boosting algorithm [Friedman '01, Mason '99]. Then, we will describe the three state-of-the-art algorithms that leverage the gradient boosting approach.

[7] https://mlcontests.com

GBDT algorithm [Friedman '01, Mason '99] is boosting (ensemble) algorithms that operate by using a gradient cost function. The boosting process of this algorithm is *a numerical optimization problem to optimize*.



A gradient boosting algorithm is composed of these components: (1) a loss function to be optimized, (2) a sequence of decision trees (weak learners) to make predictions, and (3) an additive model that adds decision trees to minimize the loss function. It is called an additive model because one new decision tree will be added to the sequence at each iteration, and existing decision trees remain unchanged (frozen) in the model. This additive model uses a gradient (check Chapter 8) to minimize the loss score. In Chapter 8, we have explained that gradient descent is used to minimize a set of parameters, but in the context of a gradient boosting algorithm, the gradient is applied to minimize residual errors in decision trees.

The gradient boost can be used for both regression and classification. To keep the clarity of the algorithm, we cheat a bit in this chapter, which is about classification, and explain its regression here.

Regression with GBDT

To implement a regression with a simplified version of the gradient boosting, the algorithm performs the following steps.

(1) First, it starts with averaging the data we intend to predict. Next, the 'residual' will be calculated by the loss function. A simplified version of residual (pseudo residual) error is computed as $actual - predicted$. We will also use the simplified pseudo residual in our examples.

(2) Then, a new decision tree (e.g., CART) is constructed based on differences between pseudo residuals and predicted values.

Next, by comparing the initial prediction and pseudo residual, a new prediction is estimated, but the tree is multiplied by a learning rate ($\alpha = 0.1$) to scale its contribution to the final prediction model and reduce the risk of overfitting.

(3) Step 1 (without average calculation) and Step 2 will be repeated until adding more new trees does not reduce the pseudo residual values or a specific number of trees reaches.

Gender	Race	Salary
Male	White	$200k
Female	Asian	$160k
Female	White	$170k
Male	Black	$150k
Female	Asian	$150k
Male	White	$200k

Table 9-10: Salary of the SocialLove company based on race and gender for employees with equal qualification.

The maximum number of trees will be given as a hyperparameter. Then, the additive model is constructed by summing up all constructed decision trees.

It is better to understand its details with an example. Assume we have access to the historical data of SocialLove, a corporation that is a pioneer in Internet social media. SocialLove[8] claims that they respect diversity and treat all their employees fairly. A friend of ours who is an AI engineer is willing to join SocialLove. We have acquired access to some internal data, and we could use it to predict the salary in that corporation based on their race and gender. Our algorithm uses gender and race from historical data to recommend a salary for a new candidate. The historical data that is used for our prediction is presented in Table 9-10.

In the first step, the algorithm calculates the average of the target feature used for prediction (i.e., average salary) and then calculates the residual error for this feature. The residual (Residual 1) and target prediction (Predicted Salary) are shown in the table on the left side of Figure 9-26. Next, the algorithm creates the first decision tree, *DT1*, based on residuals. *DT1* leaf nodes are residuals, and its decision nodes are input features (Gender and Race).

The input of the decision tree is "Gender" and "Race", and the output is residuals ("Residual 1" in Figure 9-26). Since each leaf node has more than one residual, the algorithm calculates the average of residuals and substitutes it as a leaf node value. Check the bottom table in Figure 9-26 to see the result of the average substitution for *DT1*.

Now, the first decision tree is constructed, and each leaf node (light blue nodes in Figure 9-26) has one value. We observe that some residuals are close to the predicted value. For example, the average of residuals in a leaf node is -10, which is a small difference from the original value. This is a sign of overfitting (we have high variance and low bias) and it should be resolved. To

---

[8] At the time of writing this section, in 2021, requests for gender and race fairness reached tech companies, and ethics among the AI community raised many discussions. The data of this example are not real, but this unfortunate phenomenon exists everywhere, and to show our respect for people who struggle against this attitude, we use this example here.

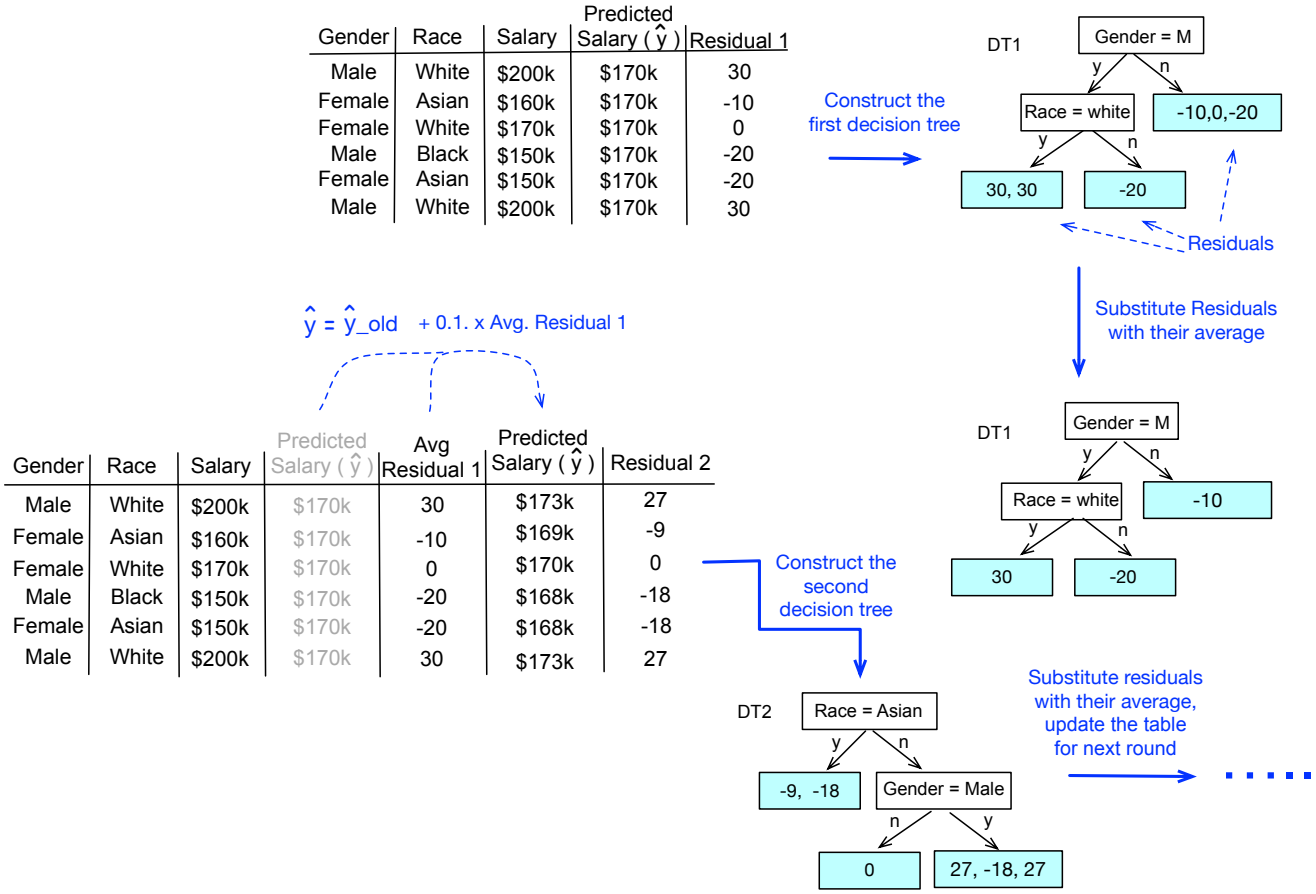


Figure 9-26: Flow of the GBDT for regression. The same information from Table 9-12, along with the predicted salary, has been added, and residuals are calculated. Since it is the first round, the predicted salary is just an average of all salaries. After residuals have been calculated, they will be substituted in the table for the next round. In the next round based on residuals, a new decision table is constructed, and then a new table with an updated prediction and residual will be created. This process continues until we reach the maximum number of trees (given as a hyperparameter).

ensure that the overfitting is handled, the algorithm adds a learning rate ($\alpha = 0.1$) multiplied by residuals to the predicted value. It uses the following equation: $\hat{y}_{new} = \hat{y}_{old} + \alpha \,.\, Residual$. Multiplication by a learning rate reduces the contribution of the decision tree in the final model, and thus, more decision trees will improve the accuracy of the final model. This is an important step to reduce the risk of overfitting. In the next iteration (step 3), we use the DT1 and calculate the predicted value as: $H(salary) = \hat{y} + 0.1(Residual\ 1)$.

Now, we can see that residuals have decreased slightly. See the second table in Figure 9-26. Again, the algorithm uses these new residual values (Residual 2) to construct a new decision tree, DT2, similar to what we have explained for DT1. The initial result is shown in the DT2 of Figure 9-26, and the described process repeats continuously, i.e., again, a new residual (Residual 3) is calculated, and the predicted value will be updated.

At the end, the prediction model will be written as follows:

$H(salary) = \hat{y}_{old} + 0.1(Residual\ 1) + 0.1(Residual\ 2) + \ldots.$

Technically, we can formalize the prediction model as follows.

$$H(x) = H_0(x) + \alpha \times (H_1(x)) + \alpha \times (H_2(x)) + \ldots = \sum_{i=1}^{n} \alpha \,.\, H_i(x)$$

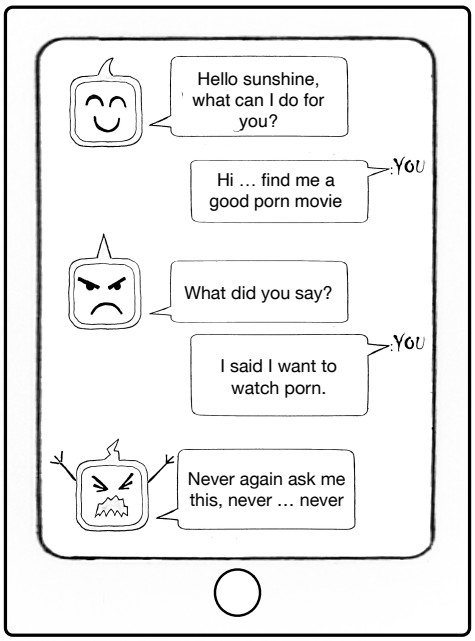
Here, $H_i(x)$ presents the residual of the model *i.* Each time we add a tree to the model, the residual values will get smaller. This process of tree creation and new residual calculation continues until the maximum number of decision trees is reached or the residual values do not change. Then, $H(x)$ is the final model and used to label new data.

When new test data arrives to estimate its salary, the algorithm finds the residual value of the right branch of decision trees (e.g., if it is male and not white, the algorithms substitute -20 for DT1, -18 for DT2, …), and substitute them in the $H(x)$ equation to predicts the salary of the new test data.

This example seems too simple; no optimization algorithm has been used, and there is nothing about gradient here. However, this simplified example helps us to understand the intuition behind using GBDT for regression.

Classification with GBDT

The gradient boosting algorithm is used for classification as well. It is fairly similar to regression one with some slight modifications, and it operates as follows:



(1) First, the algorithm starts calculating the average of the target prediction variable and then computing its *log(odds)* (check Chapter 8 to recall odds). In other words, the log(odds) is the initial prediction value for the classification task of this algorithm. Then, the Sigmoid function for binary class labeling or Softmax function for multi-class labeling is used to transform the log(odds) into a probability value[9]. This probability value will be used to decide about the initial labels. For example, if the probability is higher than 0.5, in a binary classification, then all labels will be positive, otherwise negative.

(2) Next, the residual will be calculated by the cost function. A simplified residual (pseudo residual) is

[9] In Chapter 8 we use the same approach for logistic regression and convert log (odds) into probability.

calculated ($actual - predicted$) in terms of probability (a number between 0 and 1).

(3) Afterward, a weak learner (e.g., ID3, CART, etc.) is used to construct a decision tree and predict residuals. The number of residuals is more than the number of leaves in the tree, and some leaves have more than one residual. A transformation should be implemented to assign

Length of Question (LQ)	Speak Loudly (SL)	Say Something Negative (SN)	Get Angry (GA)
8	Yes	Yes	Yes
8	No	No	No
9	Yes	No	Yes
14	No	Yes	Yes
17	Yes	Yes	Yes
5	No	No	No

Table 9-11: Previous reaction of the chatbot to the question we asked it.

Length of Question (LQ)	Speak Loudly (SL)	Say Something Negative (SN)	Get Angry (Actual)	Initial Prediction	Residual
8	Yes	Yes	Yes	0.66	1-0.66 = 0.34
8	No	No	No	0.66	0-0.66 = -0.66
9	Yes	No	Yes	0.66	1-0.66 = 0.34
14	No	Yes	Yes	0.66	1-0.66 = 0.34
17	Yes	Yes	Yes	0.66	1-0.66 = 0.34
5	No	No	No	0.66	0-0.66 = -0.66

Table 9-12: Previous reaction of the chatbot to the question we asked it.

each leaf a residual value. The following equation is the transformation to calculate the leaf value from its residuals.

$$leaf\_value = \frac{\sum_i Residual_i}{\sum P_{old} \times (1 - P_{old,i})}$$

Here, P_{old} refers to the previous predicted probability value. In the initialization phase, all predicted probabilities are the same, which we already know is wrong, but it is only the initial prediction. After this transformation, the tree is ready, and each of the tree's leaf nodes includes one value.

(4) The recently constructed tree will be multiplied by the learning rate and added to the additive model, i.e., $H(x)$. This tree will be used to calculate the new predicted probability, i.e., Sigmoid of log(odds).

(5) The process from steps (1) to (4) will continue until adding more new trees does not reduce the residual values or a specific number of trees reaches. The maximum number of trees will be

given as a hyperparameter. Then, the final model is constructed by summing up all the constructed decision trees.

Table 9-11 presents an example; here, we try to create a chatbot that is getting angry when we are talking about something harmful. We are in love with our chatbot, and we really hesitate to make it angry. Since the mood of the chatbot changes based on the questions, we developed a predictive model to avoid asking the chatbot questions that result in anger.

We would use the GBDT classification model to predict whether the chatbot gets angry or not based on the given question.

(1) The first step, as we have explained before, is to make an initial prediction and then calculate its log(odds). In Table 9-12, column GA, there are four 'yes' and two 'no'. In total, we will have six states. Its log(odds) is calculated as follows:

$log(\frac{4}{2}) = 0.69$, which we round it to 0.7.

After the log(odds) have been computed, it is converted into probability by using the Sigmoid function as follows:

$$initial\_prediction = \frac{e^{log(odds)}}{1 + e^{log(odds)}} = 0.66$$

The value of the initial_prediction $0.66 > 0.5$, thus, states that all sentences are predicted to make the chatbot angry (we assign 1 to get angry and 0 not to get angry). Clearly, this approach is not correct, but it is the initial prediction, and the algorithm improves its prediction in each iteration.

In this example, we choose to perform binary classification for the sake of simplicity, but though softmax, we can cover more than two labels.

(2) Now, the probability value of the initial prediction is used to calculate residuals, and the residual is the difference between actual values (1 for yes and 0 for no) and the initial prediction. The result of the residual is presented in Table 9-12.

(3) At this stage, residuals are ready, and the algorithm can build its first decision tree. It builds the decision tree to predict residuals using LQ, SL, and SN. Let's assume the algorithm constructs a tree like the one presented on the left side of Figure 9-27.

Since some leaves have more than one residual value, we should use the described equation for leaf values to get only one value for each leaf.

Afterward, we perform the transformation for every leaf (even the one that has only one value) by using the described equations as follows:

$$\frac{-0.66 + 0.34 + -0.66}{(0.66 \times (1 - 0.66)) + (0.66 \times (1 - 0.66)) + (0.66 \times (1 - 0.66))} = \frac{-0.98}{0.67} = -1.46$$

$$\frac{0.34 + 0.34}{(0.66 \times (1 - 0.66)) + 0.66 \times (1 - 0.66)} = \frac{0.68}{0.45} = 1.51$$

$$\frac{0.34}{0.66 \times (1 - 0.66)} = \frac{0.34}{0.22} = 1.54$$

By substituting these values, the first weak learner table will look like the right side of Figure 9-27.

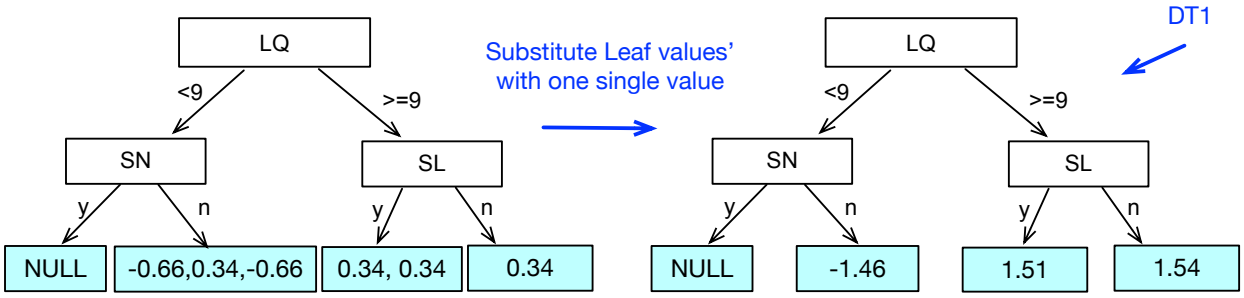


Figure 9-27: A sample decision tree is constructed (by hand to simplifying its readability) from the data in Table 9-14. Leaf nodes assigned residuals as value, but then the described equation of step (2) is used to substitute leaf nodes with more than one variable with new variable.

(4) Now that the first decision tree is ready, let's call it DT1, and it is added to the additive model. In particular, the result mode (additive model) will summate the initial prediction and learning rate (e.g., 0.1) times the decision tree.

$$model = 0.66 + 0.1 \times DT1$$

(5) Based on the additive model, the algorithm calculates the log(odds) of each record and substitutes it as the prediction column.

For example, for the first record in Table 9-12, we have *LQ<9* and *SN=No*. By substituting them in the model, we have $model = 0.66 + 0.1 \times 0 = 0.66$. For the fourth record, we have *LQ>=9* and *SL=No,* and thus $model = 0.66 + 0.1 \times 1.54 = 0.814$. These log(odds) values will be used by the Sigmoid function to calculate the new prediction value and construct a table similar to Table 9-12. The initial prediction value will be substituted by the results Sigmoid of log (odds). Then, from step (2), the process iteratively continues until a threshold for a maximum number of trees is reached or the loss score does not improve. The final model will be something like the following: $model = 0.66 + 0.1 \times DT1 + 0.1 \times DT2 + 0.1 \times DT3 + \ldots$

When a new record is added to the table, the algorithm identifies its label (either get angry or not get angry) by substituting its values in the model. Now that we understand a simplified version of the algorithm, we can write it pseudo-code, which could help us understand the algorithm in more technical detail. In the examples described, both regression and classification, we try not to describe its math. Nevertheless, while formalizing GBDT, we describe its math.

The algorithm gets the input dataset, i.e., $\{(x_i, y_i)\}_{i=1}^{n}$, and a differentiable[10] loss function, i.e., $L(y_i, \gamma)$, as an input. The loss function should be differentiable because it uses the chain rule of derivatives (check Chapter 8 to recall these mathematical terms). x_i presents the features for the given input value (one row of data without the prediction column), y_i presents the output, and n is the number of data points in the training set. The loss function L evaluates how well we can predict y_i, and it is computed as follows (for a regression problem):

$$L = 1/2(predicted - actual)^2 .$$

The gradient boosting algorithm can be written as follows. While reading the pseudocode, please read the blue text as comments on top of each command.

The input data is the training dataset ($\{(x_i, y_i)\}_{i=1}^{n}$), x is the input variable, and our objective is to predict y, given the differentiable loss function. $L(y_i, F(x))$ Is a differentiable loss function, and $F(x)$ is the prediction function

Input data: $\{(x_i, y_i)\}_{i=1}^{n}$, $L(y_i, F(x))$

At the beginning, the algorithm initializes the model ($H_0(x)$) with a constant value. $L(y_i, \gamma)$ is the cost function that should be minimized. y is the prediction variable and γ is the log(odds) value. In this context, *arg min* means we must find the *log(odds)* that minimizes the sum. This minimization can be implemented with the Newton method or gradient descent.

$$H_0(x) = arg\min_{\gamma} \Sigma_{i=1}^{n} L(y_i, \gamma)$$

For m =1 to M # M is the number of trees to be constructed, usually set to 100.

The following line computes residual for 'n' training set. i in $r_{i,m}$ stays for the sample number, and m for the number of trees. The residual ($r_{i,m}$) is the derivative of the loss function with respect to the predicted value.

$$r_{i,m} = -\left[\frac{\partial L(y_i, H(x_i))}{\partial H(x_i)}\right]_{H(x)=H_{m-1}(x)} \quad for \quad i = 1,...,n$$

it uses the 'train' dataset to construct a weak learner ($H_m(x)$). $H_m(x)$ fits the predicted value to the residual value. Meaning, it builds a tree to predict pseudo-residuals ($\{(x_i, r_{i,m})\}_{i=1}^{n}$) instead of prediction value ($\{(x_i, y_i)\}_{i=1}^{n}$). Therefore, $Fit(H_m(x), r_{i,m})$ trains the mth weak learner to predict these residuals (the errors made by the previous predictions).

$$H_m(x) \leftarrow Fit(H_m(x), r_{i,m})$$

Each leaf from the constructed tree $H_m(x)$ has k number of residuals to determine the output values for each leaf. The output (predicted) value is a γ (gamma), a.k.a. "multiplier" (or log(odds) in classification case). γ is the minimum of the cost function. It resolves an optimization problem with a derivative of the cost function and chain rule. In other words, this equation minimizes the optimization function.

In the case of classification case, instead of $i = 1$ in the summation, we will have $x_i \in R_{ij}$, which R_{ij} refers to all residual values in that leaf.

$$\gamma_m = arg\min_{\gamma} \sum_{i=1}^{k} L(y_i, H_{m-1}(x_i) + \gamma)$$

[the algorithm continues on the next page]

[10] A function that its derivative existed at each point is referred to as a differentiable function.

```
        # At the end, the model will be updated and making a new prediction for each
        sample by adding it to the previous prediction. α is the learning rate.
        H_m(x) = H_{m-1}(x) + α × γ_m
return H_m(x)
```

For some mysterious reason, we did not find much discussion about the computational complexity of the GBDT algorithm. A paper by Si et al. [Si '17] reports that considering N is the number of data points and L is the number of labels (size of output space). At least $O(NL)$ time and memory are required to build GBDT trees. The residual density grows after each iteration, and will become a matrix of size $N \times L$.

Finally, we are done with the gradient boosting algorithm and other algorithms. Take out your brain from the fryer and put it in an ice bucket because the remainder of this chapter will explain some state-of-the-art classification algorithms.

NOTES:

* In the described examples, we have explained that no loss function has been used, and we calculate a pseudo-residual, not a real residual, which is calculated by the loss function.
* In the described examples, we have simplified the weak learner (tree) construction and do not use any of the standard tree construction algorithms for the sake of simplicity.
* Usually, a gradient boost used for classification uses trees with 8 to 32 leaves.
* The pseudo-code for the algorithm for both regression and classification is the same, except for a few differences while calculating γ_m.
* In the classification case, to calculate γ_m instead of derivate, the algorithm can take a second-order Taylor polynomial and approximate the minimum value.

eXtreme Gradient Boosting (XGBoost)

As we have explained previously, at the time of writing this book, XGBoost [Chen '15] is among the best algorithms in terms of their accuracy, especially when deep learning algorithms do not meet the demands of the users. XGBoost is a derivation of a GBDT algorithm, which has employed lots of optimization techniques, and its cost function has a regularization (check Chapter 8 to recall regularization) along with the cost function as well.

XGBoost optimization methods include using an approximate greedy algorithm, parallel learning, and a weighted quantile approach to identify the threshold value of split faster. In addition, the algorithm handles the sparsity of data efficiently, using caching (bringing data to CPU cache memory instead of the hard disk) and sharding (a method to facilitate disk access), which results in a better execution time by dealing with missing values. We do not explain the details of optimization methods to preserve your brain cells to understand the core of this very accurate but heavy algorithm.

Vanilla Decision Tree

Gradient Boosting Decision Trees

Unlike the GBDT method, XGBoost uses a specific type of decision tree, i.e., XGBoost tree. XGBoost has many hyperparameters, and we describe the important ones here, i.e. η, γ, "minimum child weight", "maximum depth", "subsample", λ and α. The hyperparameter η (default = 0.3) is a learning rate and shrinks the model weight at each iteration to make it robust against overfitting. A node in XGBoost split if the split leads to a reduction in a loss function. γ (default = 0) is a threshold to specify the minimum loss reduction that is required to perform the split. "Minimum child weight" specifies the minimum sum of weights of samples required to be in each child node. "maximum depth" (default = 6) specifies the maximum depth of a tree (deep trees are prone to overfitting). "subsample" parameter specifies the ratio of sampling for the training dataset. For example, subsample = 0.5 means at every iteration, the algorithm randomly samples 50% of training data before growing a tree. λ is a L_2 regularization parameter and α is a L_1 regularizer on weights (leaf values). There are several other hyperparameters, we do not explain them, but you can check them from XGBoost documentation.

XGBoost can be used for both classification and regression. The loss function for classification and regression differs, but both cases use second-order Taylor approximation (Check Chapter 8 for Taylor series). XGBoost cost function is a gradient loss function plus a regularization, which is very similar to ridge regression (Check Chapter 8).

Assuming x_i is the input and, $\hat{y}_i$ is the output, $\hat{y}_i = \Sigma_{k=1}^{K} f_k(x_i)$, which is a combination of K number of decision trees. The objective function of the GBDT algorithm can be formalized as $L(y_i, \hat{y}_i)$. Additionally, GBDT can have a regularizer Ω to reduce the risk of overfitting. The following equation presents the objective function of a GBDT algorithm with a regularizer:

$$\Sigma_{i=1}^{n} L(y_i, \hat{y}_i) + \Sigma_{k=1}^{K} \Omega(f_k)$$

In this equation, $L(y_i, \hat{y}_i)$ is the gradient cost function, f_k presents the kth decision tree, and $\Sigma_{k=1}^{K}\Omega(f_k)$ is the regularizer.

XGBoost is a type of GBDT with a specific regularizer, and the regularizer in XGBoost for the kth tree is calculated as follows:

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda \Sigma_{j=1}^{T} w_j^2$$

T is the number of leaves (terminal nodes), γ is a user-defined penalty, and the rest is the L_2 norm or Ridge penalty (check Chapter 8 regularization section), which encourages tree leaves to have smaller weights.

XGBoost algorithm operates as follows:

(1) First, similar to the GBDT, the algorithm calculates all initial residuals. It starts with a base prediction (like the average of the target values for regression), represented by a single node or leaf. Then, each successive tree in the iteration process aims to correct the residuals left by the previous trees.

(2) The algorithm tries different splits on residuals for each feature. Then, for each candidate tree, based on the split in residuals and previous prediction value, it calculates a *similarity weight*. For classification, it uses the square summation of all residuals divided by the summation of previous probabilities (P_{old-i}):

$$Similarity\ Weight = \frac{\sum Residuals^2}{\sum_i [P_{old-i} \times (1 - P_{old-i})] + \lambda}$$

For regression, it uses the following equation:

$$Similarity\ Weight = \frac{\sum Residuals^2}{Number\ \ of\ \ Residuals + \lambda}$$

(3) Next, the *Gain* for each newly constructed tree is calculated to determine the best split of data for each feature (each column except the prediction column).

Gain = Left tree (similarity weight) + Right (similarity weight) - Root (similarity weight)

The tree with the best Gain will remain, and other trees will be removed.

(4) If the difference between Gain and γ (Gain - γ) is positive, nothing changes; if it is negative, then the algorithm prunes the tree, removes that branch, and again subtracts γ from the next Gain value way up to the tree.

(5) The algorithm calculates the "output value" for the leaves as follows:

Output value = Sum of residuals / Number of residuals + λ

(6) As the first decision tree is created, the algorithm calculates the *log(odds)* of the initial prediction value and the output value of the related branch in the decision tree to make a new prediction. The result is a new predicted probability, and this value will be used (instead of the initial predicted probability) to adjust residuals and continue from Step 2 until the maximum number of trees is reached or residual values do not improve.

After the model is ready to perform prediction, it acts similarly to the gradient boosting algorithm. In particular, for new test data, the algorithm calculates the log(odds) of the previous

prediction value plus learning rate (η) times the log(odds) of output for each XGBoost decision tree. Let's be honest together: how can we expect you to memorize the above explanation without an example? To understand this complex algorithm, we follow the holly traditions and learn it with an example. We have the data of a few readers who have read this chapter and experimented with the described algorithms by writing some codes from scratch. We would like to predict whether a new reader who read this chapter a couple of times, with some implementations of these algorithms, has already learned the described algorithms or not. Table 9-13 shows the dataset we have for training. For testing, we give a new record with specified RC and EC, and the algorithm predicts the LA.

Number of time this chapter is read (RC)	Experiment by Coding (EC)	Learn the Algorithm (LA)
2	Yes	Yes
1	No	No
2	Yes	No
2	No	No
2	Yes	Yes
1	No	No

Table 9-13: Sample dataset used by XGBoost.

In this example, for ease of understanding in describing the algorithm and calculating its math, we do not calculate α, and we also assume λ, γ, "minimum child weight", and other hyperparameters are all set to zero.

(1) To implement Step 1 of the XGBoost algorithm, for 'yes' in LA, we assign '1', and for 'no', we assign '0' (for simplicity, we use binary classification). Similar to the GBDT, the initial prediction could be the average of their possible values, and it is $(0 + 1)/2 = 0.5$, or something similar. During iterations in all GBDT algorithms, the initial values will be changed, and it is unimportant how we initialize it. Initial pseudo-residuals are calculated as the differences between observed and predicted values. The predicted value is equal to the initial prediction, in the beginning, which is also called *the base model,* because it participates in constructing the final model. By substituting initial residuals in Table 9-13, we get the table presented on the left side of Figure 9-28. At this moment, look at the table there and not the tree on the right side.

After the residuals have been calculated, the XGBoost algorithm fits a series of binary decision trees to predict the residuals of the target variable. However, it has not yet been decided which column should be used for the split and the algorithm experiment for all columns. A binary table for RC is shown in Figure 9-28. As described, these types of trees will be created for all other columns (features) as well. Besides, we could make a more complex tree and make more

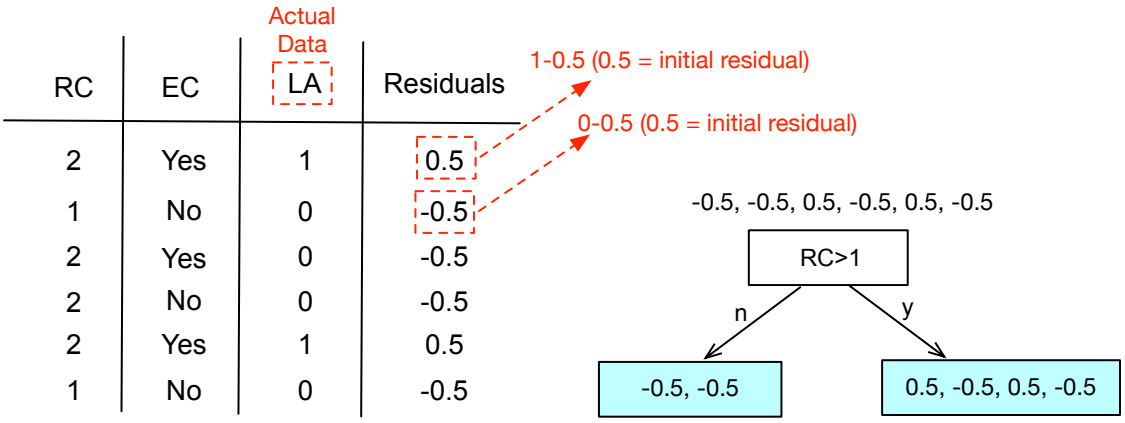


Figure 9-28: Table 9-15 is used to calculate the residuals, we intend to do a classification and since it is binary we substitute yes and no with 1 and 0 in LA column. On the right side a binary tree is constructed and we assume the RC>1 as a split.

branches, but for the sake of simplicity, we keep it very small. Why did we select $RC > 1$ as the main branch? Because we simply average RC values, and it is 1.5, and rounded to 1.

(2) In the second step, "similarity weight" for each leave node is calculated. For the sake of simplicity, we assume $\lambda = 0$. Then, $P_{old-i} = 0.5$ because it was our initial prediction probability. Therefore, the similarity weight for the leaf of the tree in Figure 9-28 is calculated as follows:

$$\frac{(-0.5 + -0.5)^2}{0.5(1-0.5) + 0.5(1-0.5) + 0} = \frac{1}{0.5} = 2$$

Respectively, the similarity weight for the leaf on the right side is calculated as follows:

$$\frac{(0.5 + -0.5 + 0.5 - 0.5)^2}{0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0} = \frac{0}{1} = 0$$

We also compute the similarity weight for the root node (which is a node that includes all residuals) as follows:

$$\frac{(-0.5 + -0.5 + 0.5 + -0.5 + 0.5 - 0.5)^2}{0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0.5(1-0.5) + 0} = \frac{1}{1.5} = 0.66$$

Now, similarity weights for all tree nodes get calculated. Also, the algorithm calculates the "gain" for this feature (RC). Therefore, we will have: $2 + 0 - 0.66 = 1.34$. Figure 9-29 writes similarity weights on top of each node on a tree.

(3) The algorithm performs the same process and calculates the gain for other features (columns), which are used to predict the LA (the target data). The only remaining feature is EC. Its tree, similarity score, and gain score are presented in Figure 9-30.

EC gain is 2.67, and it is higher than RC gain, which is 1.34. Therefore, the algorithm chooses EC to perform the split on the root node.

Note, if there are three different labels for one feature (our example has two labels only, and we don't have this situation), the algorithm experiments with different possible combinations for binary tree construction (XGBoost split trees are always binary trees), and at it selects the tree

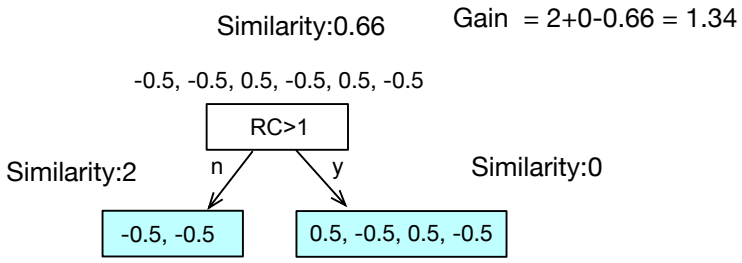


Figure 9-29: Similarity score of each node will be used to calculate the Gain score for RC.

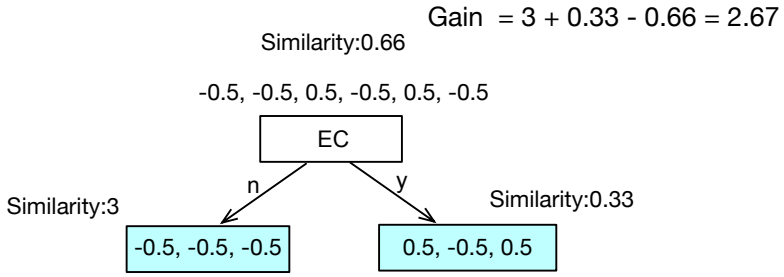


Figure 9-30: Similarity score of each node will be used to calculate the Gain score for EC.

that has the highest gain score. For example, if we have Red (R), Green (G), and Blue (B), once it makes a binary tree on one branch R and two other branches G and B, and next it calculates their gain. Then, another tree is constructed that has G as one branch; the other branch includes R and B, and it calculates their gain. Afterward, it chooses the best tree with the highest gain. However, experimenting with too many trees is infeasible and uses approximation, which we do not explain in more detail.

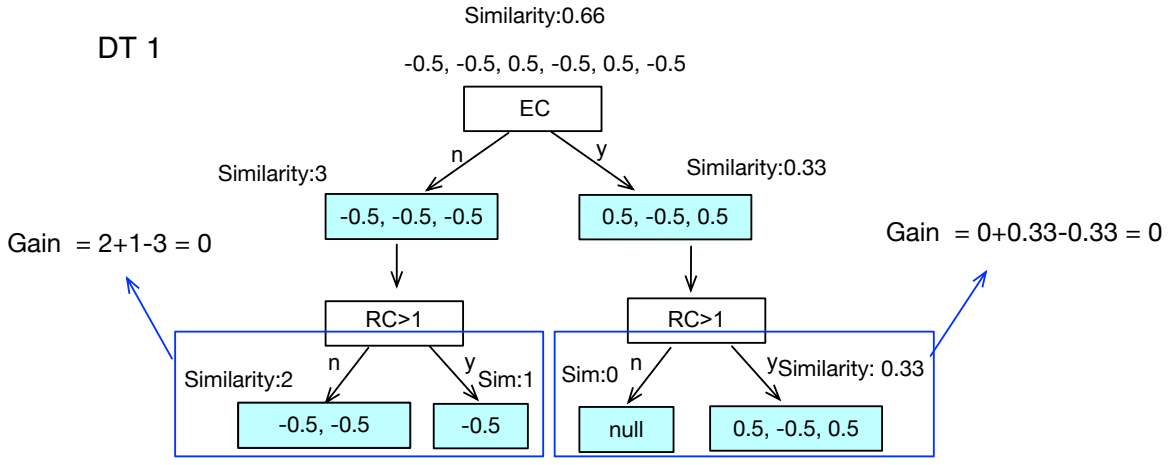


Figure 9-31: A split, and gain scores have been calculated for other branches.

(4) We have explained that we set γ it to zero, and thus, in this example, we skip pruning the tree; if we incorporate γ it removes the branches, which subtracts their gain from γ and results in a negative number.

To see the example of how the gain of a branch is computed, check the example presented in Figure 9-31. Both Gains are zero, and thus, one of the trees is randomly selected. At this stage, we can say that one decision tree is ready. Let's call it *DT1* (Figure 9-31).

(5) Now, the output value for each leaf should be calculated by the equation we have described (*Output value = Sum of residuals / Number of residuals* + λ). Recall that we set λ to zero.

The followings show the calculation of the output for each branch :

$$EC = n, RC = n :$$
$$output : \frac{-0.5 + -0.5}{2} + 0 = -0.5$$

$$EC = n, RC = y :$$
$$output : \frac{-0.5}{1} = -0.5$$

$$EC = y, RC = n :$$
$$output : 0$$

$$EC = y, RC = n :$$
$$output : \frac{0.5 + -0.5 + 0.5}{3} + 0 = 0.16$$

Figure 9-32 is Figure 9-31, but it presents the output result for each branch.

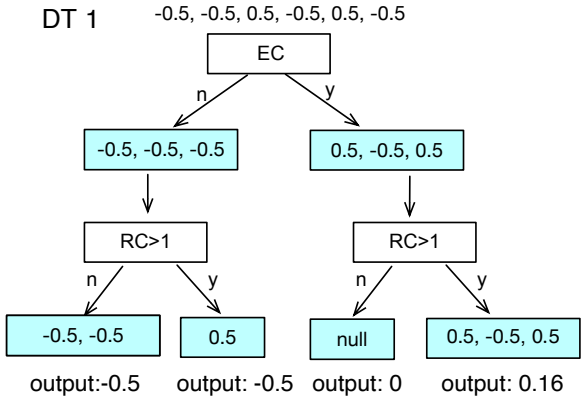


Figure 9-32: Output for each leaf has been calculated and added to the bottom of leaves.

(6) Now, the algorithm iterates from step 1 to construct the new prediction probabilities and substitute the initial one. The *log(odds)* of initial probability, which is equal to 0.5, will be 0 we donot write them in Table 9-14. The new log(odds) of predicted probability will be $0 + 0.3 \times log(odds)\ of\ DT1_{output}$ (0.3 is the recommended learning rate η). For each record, we can substitute the values into the $0 + 0.3 \times log(odds)\ of\ DT1_{output}$ equation and give us the result. To calculate the predicted probability from log(odds), a Sigmoid function is used as follows:

$$Predicted\ Probability = \frac{e^{log(odds)\ of\ p}}{1 + e^{log(odds)\ of\ p}}$$

Then, the result will be a new predicted probability, and its differences with LA will be used to construct new residuals (see Table 9-14). The new residuals are usually smaller than the residuals in the previous rounds.

To decide on the LA label of the new record, the new record data will be substituted in the model (i.e., *initial prediction* + *DT1* + *DT2* + ...), and its LA label will be based on the result of the model.

RC	EC	LA	Old Residuals	New Log(Odds) of predicted probabilities	New Probabilities	New Residuals
2	Yes	1	0.5	0.3 x 0.16	0.045	0.96
1	No	0	-0.5	0.3 x -0.5	-0.17	0.17
2	Yes	0	-0.5	0.3 x 0.16	0.045	-0.96
2	No	0	-0.5	0	0.5	-0.5
2	Yes	1	0.5	0.3 x 0.16	0.045	0.96
1	No	0	-0.5	0	0.5	-0.5

$$\frac{e^{log(0.3\times0.16)}}{1 + e^{log(0.3\times0.16)}} = 0.045$$

Table 9-14: New prediction probabilities and their residuals, in each iteration residuals will get smaller. which are constructed based on old residuals. For sake of simplicity we use absolute value and neglect negative residuals.

The XGBoost for regression operates very similar to classification, except for its similarity weight calculation, which we have explained before. Therefore, we skip its explanation with a detailed example. Before finalizing this section, remember that the only difference between classification and regression of XGBoost is their loss function.

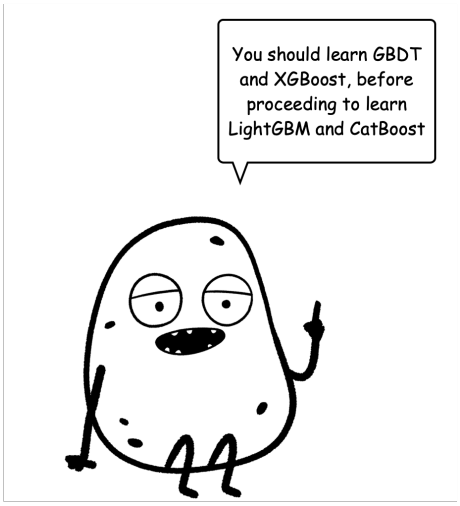
Assuming K is the total number of trees, d is the maximum depth of trees and $||x_0||$ is the total number of training data points with non-missing data, and n is the total number of data points, the computational complexity of XGBoost for training is $O(Kd||x_0|| + ||x_0||log\ n)$. The $||x_0||log\ n$ part belongs to the preprocessing phase.

In October 2023, XGBoost 2.0 was released. It has some improvements in its implementation, including native support for GPU, improved memory management, and probably the most important one is multi-target trees. The multi-target trees allow XGBoost to build one tree for all targets (instead of many trees). This helps make smaller models and prevents overfitting.

LightGBM

Later, after XGBoost and its tremendous success in Kaggle competitions, back in 2017, another interesting algorithm was released by Ke et al. [Ke '17]. One of the problems in XGBoost was the need to scan many possible combinations to decide on the best split. LightGBM, another GBDT type, tries to mitigate this by using Gradient-Based-One-Side Sampling (GOSS) and reducing the number of features by Exclusive Feature Bundling (EFB).

GOSS operates based on the fact that different gradients of training data points have a different impact on the information gain (the result of the algorithm). In particular, data points with larger gradients that are under-trained data points contribute more to the result of prediction (i.e., information gain). Respectively, data points with small gradients are not important for the algorithm, and GOSS randomly samples a subset of data points with small gradients. This process decreases the role of gradient in learning. On the other hand, data points that have a large gradient are important for the learning phase of the algorithm, and they will be kept, e.g., top 30% of gradients.



Usually, real-world datasets include lots of zeros; even one-hot encoding makes sparse features because there are too many zeros added to the feature list. The EFB uses the sparsity that exists in most real-world datasets. Features that never take non-zero values together are referred to as *Exclusive features*. EFB merges exclusive features into a single feature (similar to the feature reduction techniques we have described back in Chapter 6). In other words, EFB reduces the problem of feature reduction to a graph coloring problem and uses a greedy algorithm[11] to remove features. In particular, all features will be a vertex in the graph, and if they are not mutually exclusive[12], the algorithm adds one edge between them. The weight of edges corresponds to conflicts between features.

---

[11] Greedy algorithms refer to heuristic algorithms (check Chapter 6) in which each step chooses a local optimum. In the end, it might not end up being the best optimum solution, but each step alone chooses the best optimum that is available for the algorithm.

[12] Mutual exclusion means that two processes can not exist in the same state. Remember that, assume a rest-room which usually two people can not enter one rest-room.

To understand the characteristics of EFB, take a look at Table 9-15. There we have ‘Feature 1’, ‘Feature 2’, and ‘Feature 3’. LightGBM can not merge ‘Feature 1’ and ‘Feature 2’ together because in the third row, ‘Feature 1’=3 and ‘Feature 2’=2. To make a bundle, at least one of them should be zero. However, it can merge ‘Feature 2’ and ‘Feature 3’ together because, in each record, at least one of them is zero. There is a γ parameter that can tolerate small conflicts (e.g., a few records have non-zero values, but both features are still merged) in each bundle, but we did not use it in our example.

While constructing bundles, one problem is that the bundled feature should not be in the same range as the original feature. To solve this issue, the algorithm adds a constant value to bring them to a different range. For example, here, we add 10 to each non-zero value of the bundling feature.

Feature 1	Feature 2	Feature 3
0	2	0
1	0	1
3	2	0
0	0	3
4	0	0
0	2	0

Feature 1	Feature New
0	2+0
1	0+1
3	2+0
0	0+3
4	0+0
0	2+0

Feature 1	Feature New
0	2+0+10=12
1	0+1+10=11
3	2+0+10 =12
0	0+3+10=13
4	0+0
0	2+0+10=12

Table 9-15: (left) original table of data, which has three features, (center) feature 2 and feature 3 could be merged together, because at least one of them is zero, (right) to avoid having the number of bundled feature in the same range as original data, a constant value will be added to non-zero columns.

Assuming n is the total number of data points, k is the number of features, and m is the number of bundles, the EFB algorithm of lightGBM reduces the $O(nk)$ complexity $O(mn)$. Nevertheless, the computational complexity of GOSS (linear as well) and GBT will be added to this value.

CatBoost

CatBoost [Prokhorenkova ’18] is a third popular GBDT-based algorithm introduced after LightGBM in 2018. CatBoost described that previous GBDT works suffer from the *prediction shift* problem, and it can mitigate this problem.

The prediction shift problem occurs when the model is trained on a dataset whose distribution has shifted or is different from the distribution of the actual data on which predictions are to be made. This discrepancy can lead to inaccuracies because the model is learning from a dataset that does not represent the true underlying distribution of the target population.

CatBoost introduces two improvements to the GBDT algorithm: (i) a specific method to encode categorical features, i.e., "Ordered Target Statistic" (Ordered-TS), and (ii) "Order Boosting", which is a permutation-driven alternative to the classical boosting algorithm.

To implement the ordered target statistic, the CatBoost uses target encoding (check Chapter 6). However, we have explained in Chapter 6 that target encoding is prone to data leakage. CatBoost resolves this issue by first applying a permutation (rearranging the order of data) on data, which relies on a specific ordering principle. After ordering the data, the target value of every feature is calculated from the rows before (previous observations or historical observations). It uses the following equation (a simplified version of the original version in the paper) to apply target encoding while resolving the data leakage problem.

$$Ordered - TS = \frac{current - count + (\alpha \times p)}{total - feature - count + \alpha}$$

In this equation, *current-count* specifies the sum of values for the feature we are applying target encoding, but not all values; it applies the sum until the current data point. p stays for the prior, it is a constant value, and it is used to smooth the result of target encoding. A common value for p is the average of the target value in the dataset, e.g., 0.5. α (which sets larger than zero) is a weight parameter used to ensure not dividing by zero and specify the weight on prior. *total-feature-count* specifies the total number of the current feature values, excluding the current row.

To understand how Ordered-TS works, take a look at Table 9-16. Here, we assume p as a constant and equal to 0.5 and α is 1. Table 9-16 (a) presents features and their values. It is a result of binary classification, and thus, each row has a value of either 0 or 1. In Table 9-16 (b), the average of each feature is calculated, and in Table 9-16 (c), these features have been substituted on the original table. Table 9-16 (c) presents the target encoding, prone to data leakage.

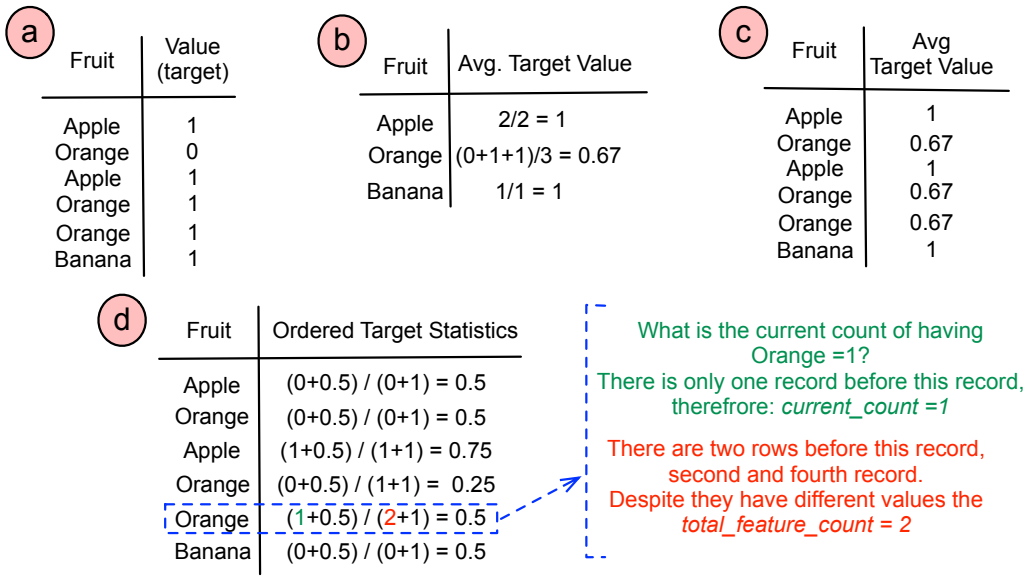


Table 9-16: An example how CatBoost calculate the Ordered Target Encoding.

The authors proposed using the described equation and resolving the data leakage by ordering. Take a look at Table 9-16 (d), which is the ordered target encoding value. Note that *current_count* and *total_feaute_count* are both calculated by looking only at previous records, not the current one and overall table. In Table 9-16 (d), we describe the justification of the sixth row. As another example, let's take a look at the value of the third row, where Apple=1, in Table 9-16 (c). Table 9-16 (d) shows the *current_count* of this row is equal to 1, because there is one row before this row, i.e., the first row that has Apple=1, and the *total_feature_count* of the Apple feature is also 1, because there is only one row that includes Apple before this row, i.e., the first row.

The second novelty of CatBoost is its Order Boosting. Order Boosting does not use the whole dataset to calculate the residuals at every iteration of GBDT. In classical GBDT, multiple trees are constructed to fit the entire dataset. Using the entire dataset might lead to overfitting, which CatBoost referred to it as a "prediction shift". CatBoost tries to resolve this issue using a tree that is constructed using a subset of the dataset, and it calculates the residuals using the data points that it did not see before (from another subset of the dataset). To understand this feature, let's say we have a 1D dataset of 10 data points, i.e. $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$. The CatBoost applies a random permutation on these data, and after applying the permutation, the data points are ordered as follows: $\{x_4, x_6, x_3, x_1, x_8, x_2, x_7, x_9, x_5, x_{10}\}$. A model that is trained on $\{x_4\}$ will be used to determine the label for x_6, and its residual error will be calculated (*actual label - predicted label*). Respectively, a model that is trained on $\{x_4, x_6\}$ will be used to determine the label for $\{x_3\}$, and its residual error will be calculated. It means the model which is trained on $\{x_4, x_6, x_3, x_1, x_8\}$ has never been seen x_2 before, but it uses x_2 to determine its residual.

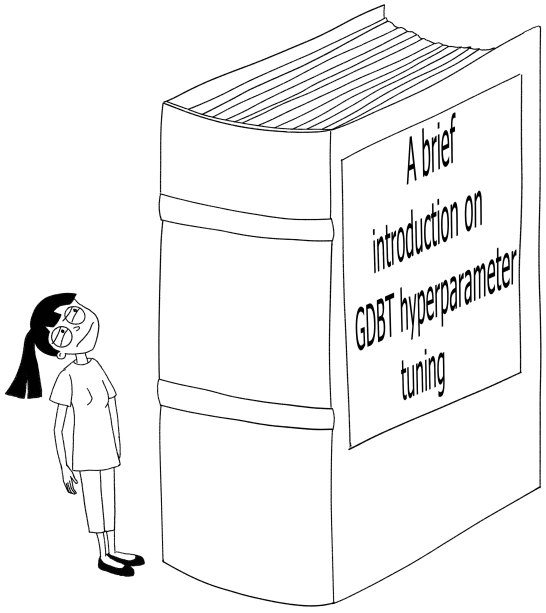
This approach resolves the prediction shift, but by using this approach for every single data point, one decision tree needs to be constructed, which leads to quadratic time and memory use, and thus, it is computationally very expensive. To resolve this issue, the authors proposed the following solution: assuming we have n data points, the authors proposed constructing decision trees for data points that are located at 2^j, where $j = 1,2,4,8,...log_2(n)$. This means the first model is trained on the first data point. The second model is trained on the first and second data points. Respectively, the third model is trained on the first four data points, the fourth model is trained on the first eight data points, etc. In the end, instead of having n trees, we have $log_2(n)$ trees.

Assuming s is the number of permutations and n is the number of data points. $N_{TS,t}$ is the number of TS to be calculated at iteration t, and C is the set of candidate splits to be considered at the given iteration b_j^t is the ith leaf value at iteration t. Authors claimed the computational complexity of calculating the gradient is $O(sn)$, building the tree T is $O(|C|.n)$, calculating all b_j^t is $O(n)$, updating a model is $O(sn)$, and calculating the Ordered-TS is $O(N_{TS,t}.n)$.

It seems all linear, but, summing them together makes CatBoost a very computationally expensive algorithm. In an experiment, Keshavarz et al. [Keshavarz '20] compared the

computational complexity of these GBDT algorithms, and CatBoost is the most resource-intensive one.

If you are still alive after learning these three hard-core GBDT models, let's do a brief review of what we have explained. XGBoost implements several optimizations, but its salient difference is the use of regularization and XGBoost tree instead of weak learners on top of the GBDT. LightGBM is a GBDT that provides two additional features, GOSS and EFB. Catboost is a GBDT used for categorical data, and it provides two additional features, i.e., Ordered Target Statistics and Ordered Boosting.



NOTES:

* Except for LightGBM, other GBDT methods grow trees by increasing their depth (leve-wise tree growth). It means they identify the best node to split, and from that node, the tree will split down to branches. However, LightGBM grows trees by increasing the leaves in the related node (leaf-wise growth) to reduce the loss score and keep the other leaves in the tree intact.
* If there is no GPU available and you are using a local machine for training, usually LightGBM consumes less amount of resources than XGBoost and CatBoost, which operate well with GPU.
* These three boosting algorithms are state-of-the-art algorithms that can work accurately on tabular data and when we have small data.
* Despite all GBDT algorithms being good at operating with a small number of data points and tabular data, their resource utilization is very high. This makes them infeasible to be used on battery-powered small devices such as smartwatches.

How to Select the Best Classification Model?

We have learned several classification algorithms in this chapter and logistic regression in the previous chapter. Now, assume we have experimented with a set of algorithms, and then we would like to decide which models perform better. The first approach that can be used for model selection relies solely on the model's accuracy. In particular, by comparing the cost (false-negative, false-positive) with the benefit (true-positive, true-negative) or using the ROC curve of different models (check Chapter 8 to recall what the ROC curve is) or other evaluation metrics. What if the accuracy differences appeared by chance? To have a better estimate of whether there

is a real difference between the two models, we should use the *significance test* (check Chapter 3 if you can't recall). For example, we performed a 10-fold cross-validation, and it resulted in ten different error rates, one for each model. We can use the t-test to compare the mean and variance of these error rates set (each model has 10 error rates) and check whether there is a significant difference between the error rates of these models.

Nevertheless, sometimes the accuracy is not as important as execution time. Thus, execution time is another parameter to decide about a model from a set of models. For applications that require making decisions in real-time and close to the real-time execution time, it plays a crucial role. In some cases, we are dealing with limited resources, such as algorithms that run on any battery-powered devices, from fitness trackers to electric vehicles; in these cases, resource utilization of the algorithm also plays a crucial role in deciding the best classification model.

Summary

This chapter starts by describing rule-based classification. The rule-based classifier is not really a machine learning algorithm because we define rules by hand, and somehow, manually, we build the model. Next, we have described some common classification methods, including the Bayesian network, kNN (by using LSH, Voronoi tessellation, and KD-Tree), SVM, and four different decision trees, including ID3, CHAID, C4.5, and CART. The differences in these decision trees are based on their tree split policy, and to review them, you can check Table 9-8.

Ensemble Learning Method	Description	Algorithm Examples
Stacking	It is a combination of weak learners that are combined to build the final model.	Combination of Weak Learners
Bagging	It chooses a random subset from the training set and builds a model for each subset. Each model assigns a label, and the final label will be assigned based on majority voting or averaging the candidate labels, provided by each classifier.	Random Forest
Boosting	Boosting is a sequential process that starts by applying a classification algorithm to the entire train set and makes a model. Next, it collects the errors of the previous model and makes a smaller dataset from errors. Then, it uses the same classification algorithm (similar to Bagging) on the dataset that includes errors and creates another model. This process continues in several iterations until a maximum number of iterations (hyperparameter) is reached.	AdaBoost Gradient Boosting Decision Tree XGBoost LightGBM CatBoost

Table 9-17: Summary of ensemble learning algorithms.

Afterwards, we have explained the ensemble learning algorithm, which uses the described algorithms as weak learners and combines them for a better learning result. Table 9-17 summarizes three ensemble approaches.

Bagging is used with decision trees as classifiers because decision trees are prone to high variance, and bagging resolves the high variance problem. The Random Forest algorithm is the most popular bagging algorithm. First, it creates subsets from the original dataset, then selects several features from each subset, and it uses these features to calculate the split. Afterward, the test dataset will be fed into all trees, and then its labels will be decided by averaging the labels of the other decision trees. Random forests are not as good at solving regression problems as they are at sorting because they do not give a continuous output. AdaBoost is a boosting algorithm that uses a set of small decision trees. It increases the weight of hard-to-classify data points and decreases the weight of easy-to-classify data points. Then, it combines multiple weak learners into a single strong learner. The result of an AdaBoost algorithm is a set of classifiers (stumps), each associated with a weight.

The Gradient Boosting Decision Tree (GBDT) algorithm is an ensemble algorithm that operates using a gradient cost function. The boosting process of this algorithm is a numerical optimization problem. We have explained that a gradient boosting algorithm is composed of (i) a loss function to be optimized, (ii) a sequence of decision trees (weak learners) to make predictions, and (iii) an additive model that adds decision trees to minimize the loss function. Similar to other Boosting algorithms, GBDT uses a residual (error) to calculate the best split for the next step, and finding the residual and splitting is the task of its cost function. Three algorithms are built on top of GBDT, including XGBoost, LightGBM, and CatBoost. At the time of writing this Chapter, these algorithms are state-of-the-art algorithms to use on tabular data whose number of data points is limited.

Further Reading or Watching

* There is a good book for basic algorithms, Mastering Machine Learning Algorithm by Jason Brownlee [Brownlee ’16]. It includes more examples of basic supervised machine learning algorithms. We use this book to write the Naive Bayesian section of this chapter.
* Victor Lavrenko (https://www.youtube.com/user/victorlavrenko/videos) has a good video series on teaching LSH. We have used his explanation to construct our explanation as well, and also good videos to explain the decision tree.
* There are many explanations of Kernel function and even books written about kernels [Scholkopf ’01]. Nevertheless, the slides for Matt Gormley from Carnegie Mellon University are one of the few that we find useful and help us learn the motivation behind using kernel functions. https://www.cs.cmu.edu/~mgormley/courses/10601-s17/slides/lecture12-svm.pdf. Besides, Josh Starmer online videos on teaching the mathematics behind kernel functions are clear and helpful, like most of his other videos.

* Sefik Ilkin Serengil provides a detailed explanation of vanilla decision trees with numerical examples. If you would like to see more examples for each decision tree algorithm, check his home page: https://sefiks.com.

* Motaz Saad provides a good visual explanation of Bagging, Boosting, and Stacking, which inspired us to develop the visualization for this model https://mksaad.wordpress.com/2019/12/21/stacking-vs-bagging-vs-boosting

* While constructing examples and explanations for AdaBoost we benefit from online videos of Krish Naik, Cory Maklin, and Bhavesh Bhatt.

* Bradley Boehmke & Brandon Greenwell have an excellent book released online [Boehmke '20], Hands-On Machine Learning with R, and you can read a detailed explanation about Gradient Boosting in their book: https://bradleyboehmke.github.io/HOML/gbm.html

* We have used Josh Starmer explanations (https://statquest.org) to construct the gradient boosting and XGBoost algorithm of this chapter as well.

* To learn LightGBM and CATBoost from other resources, you can check "Machine Learning University" of Amazon as well: https://aws.amazon.com/machine-learning/mlu. They provide a simplified explanation of these two algorithms, and we use them to describe LightGBM and CatBoost.

* If you are trying to learn the mathematical concepts behind Gradient Boosting in more detail and feel the description provided in this chapter is not enough, you can check the explanation of Terence Parr and Jeremy Howard: https://explained.ai/gradient-boosting.