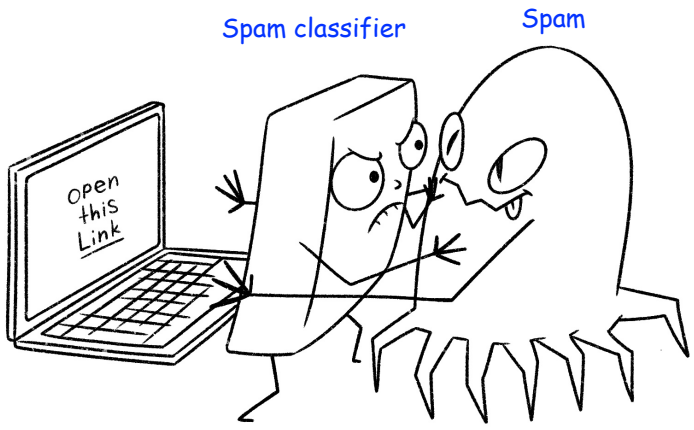
Chapter 9: Classification

Welcome to another very popular category of machine learning algorithms, which is classification algorithms. In the previous chapter, we explained regressions, we use them for quantitative information (a sexy name for datasets that have numerical relations). However, most of the real-world machine learning datasets include qualitative information (non-numerical data as well as numerical).

Classification algorithms are used to classify and separate the data objects of the dataset based on their labels. For example, an algorithm is analyzing 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 is trying to identify fake accounts from real users on social media. It is not limited to binary decisions. As a non-binary example, we can feed a dataset of handwritten digits into a classification algorithm and it can recognize the digit.

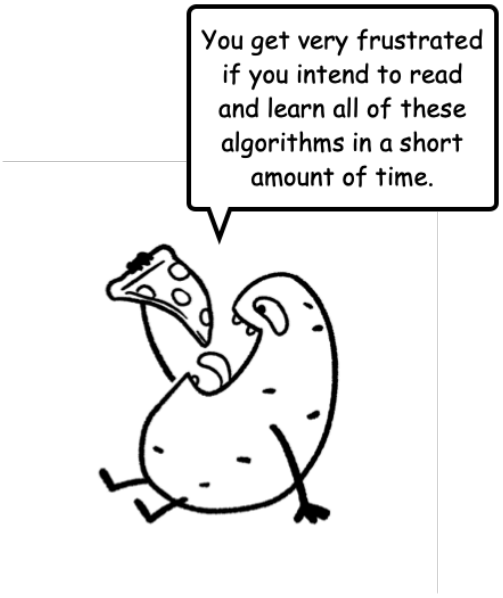


Classification algorithms are supervised learning algorithms and work based on the assumption to separate the dataset into at least two subsets, a **test** set, and a **train** set. First, we run a classification algorithm on the **train set**, which builds a model based on the given data. The model could be a set of "if-then-else rules" or mathematical equations. 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. The ground truth dataset is a small part of the train set that is annotated accurately by human experts and thus building it is labor intensive and expensive process.

There are two types of classifications, one is non-exclusive classifications, 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 one class only.
Three 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), LDA is good for binary classification. In this chapter, we start by describing 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, Adaboost, and then we finalize this chapter with gradient boost classifier, and its new derivations including, XGBoost, lightGBM and Catboost.
Note that gradient boosting algorithms of this chapter are not easy to learn and you might need to read it more than once, and also check other resources to learn them. We have tried our best to make them easy to understand, but keep in mind they are complex in their nature.

Rule Based Classifier

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



A rule based classifier is **a set of IF-THEN** that is used 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 following rules.

- IF (*does not recommend this book*) THEN (*will go hell*).
- IF (*does not pay for this book*) AND (*read 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 inside 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:

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

$n_{covered}$ stays for a number of records or data points 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:

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

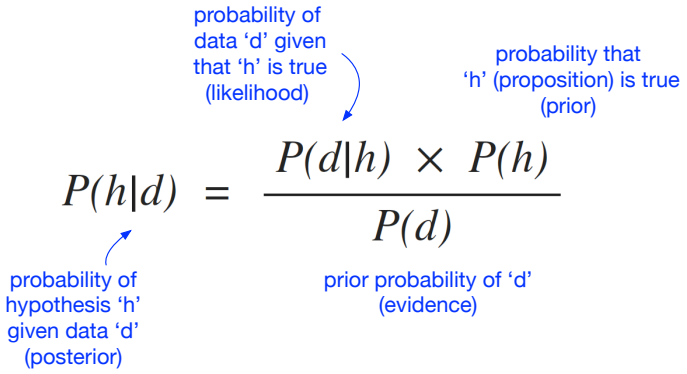
If a datapoint belongs to more than one class, the algorithm uses voting to decide about the label of that class.

Rules could be predefined by users and not learned from the training set. Also, you might say 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 the rules. Therefore it is not a good choice for a machine learning approach. Nevertheless, we can use it in different projects and enjoy its simplicity. For example, in a Game [XX] we define some rules to construct the game character's mood and body shape based on predefined rules. In particular, if the user is far away from its daily steps, then 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 (the old name of machine learning) is the Bayes theorem. It describes the probability of an event occurrence depends on the prior knowledge we have about that event or things that are related to that event[1]. For example, the probability of you getting into heaven is related to leaving a good review for this book online, and the probability of being a good human being is related to recommending it to your friends, the probability of getting obese is related to the diet, genetics, and so forth.

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



[1] Read Chapter 3 if you can’t recall the concept of probability, before continue reading the rest of this algorithm.

Bayes Theorem

We use 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, we frequently see the outcome, but we would try to have a guess what are the initial events that lead to this outcome.

In the context of machine learning, we are employing prior knowledge or evidence (d) to select the best hypothesis or proposition that predicts the future (h). This is the Bayes theorem and it is written as follows:
You might recall from Chapter 3 that we discussed the conditional probability of P(A|B) will be written as follows:

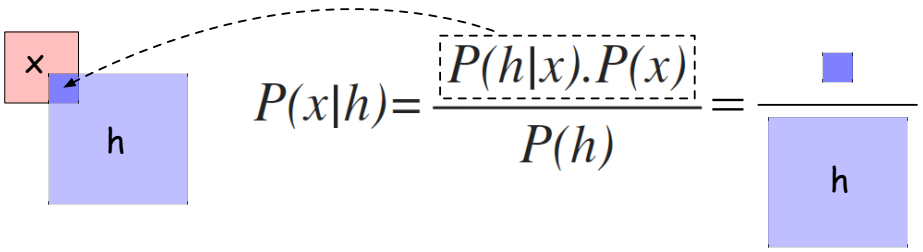
$$P(A \mid B) = \frac{P(A \cap B)}{P(A)} \text{ or } P(B \mid A) = \frac{P(B \cap A)}{P(B)}.$$

Naive Bayes assumes that all features are independent of each other. In other words, the Naiveness of this method makes an assumption 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 having A and B occur together, and it will be calculated as follows: $P(A \cap B) = P(A) \, . \, P(B)$.

Let's use an example to understand this theorem. A friend of yours told you most people who used essential oils, traditional medicine, herbal medicine, and other non-medicine methods (let's call them xmed), did not get flu this year. You are curious to see if his theory is right or wrong and let's think 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 that the probability of people who has used xmed and didn't get flu, i.e. $P(h \mid x)$. Now, we would like to know if his theory is correct, i.e. probability of people who use *xmed* given they are healthy $P(x \mid h)$. Take a look at Figure 9-1 which visualizes the Bayesian inferences and explains how this question can be answered.

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.

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 to the algorithm and available. In other words, from the training dataset, we would like to make inferences and this inference is called *Bayesian inference*.

Since $P(x)$ is known, we define our objective is to find the maximum value of $P(h\,|\,x)$ as follows: $max(P(h)\ or\ P(h\,|\,x) = P(x\,|\,h) \times p(h))$

Assuming we have a number of k possible classes (labels), $c_1, c_2, \ldots c_k$, and each of our data points has n number of features, i.e. $f_1, f_2, \ldots f_n$, then we can use the Naive Bayes classification algorithm and assign one of the available k labels to each of the data points. Therefore, based on the Bayes theorem we can write 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 written as the following equation:

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

A good example to learn Naive Bayes classification is to check if an email is a spam or not. Let's generalize the example to your life, because, after reading this book from beginning to end, you are going to be a superstar in data science. You are receiving too many emails in your private email and you have asked your secretary to remove all emails which have the word "prestigious" and come to your email address. The word "prestigious" is very common in spam emails, but could be a new client that you like to work with them, e.g. "we are a prestigious journal that publishes your scientific work". The traditional spam filters cannot filter your spam emails properly, 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 end up having the following probabilities: $P(S)$: Probability of an email that is spam. You have received 20 % of your email as spam.

$P(\sim S)$: Probability of an email that is important and not spam, $100 - 20 = 80\,\%$

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

$P(P\,|\sim S)$: Probability of an email has the term "prestigious", given that it is not spam, is 8% of not spammed emails include this term.

$P(P\,|\,S)$: 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.

$P(S\,|\,P)$: 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 $P(S|P)$ based on the following equation, which we described earlier:

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

This one is a very simple one, and we have only one feature. Lets' make the spam filter a bit more intelligent because your staff has realized that you still get some spam emails as well. So, he went to collect another 100 sample emails and analyze them. He feels that the term "family" similar to "prestigious", means no clear payment policy. For example, the person who is saying *"we are like a family in the work environment"* is looking to misuse your resources without proper payment. Let's use $P(F)$ to present the probability of an email including the word "family".

By analyzing those 100 emails manually your staff has created Table 9-1 a., which includes the

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 × 17/20 = 0.59	4/80 × 6/80 = 0.00375

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

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.

He has forgotten to analyze the occurrences of "family" and "prestigious" terms together. Based on the naive assumption events are independents, and thus we can assume $P(F \cap P) = P(F).P(P)$. Therefore, we can create Table 9-1 b. Simply by multiplying the probability by the all emails in each category to understand approximately the number of emails that include both "family" and "prestigious" and are spam, i.e. $20 \times 0.59 = 11.8 \sim 12$ and the approximate number of emails that include both "family" and "prestigious" and are not spam, are $80 \times 0.00375 = 0.3$ (which is close to zero).

Based on Table 9.1, we can have the following probabilities:
Probability of getting a spam email, $P(S) = 80/100$
Probability of getting a none spam email, $P(\sim S) = 20/100$

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 $P(P \mid \sim S) = 6/80$
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, $P(F \mid \sim S) = 4/80$
The probability of having a spam email, given it includes both words "family" and "prestigious" is as follows, $P(S \mid F \cap P) = 0.59$. The sign $\cap$ or $\wedge$ stayed for conjunction (and), and the results can be read from Table 9-1.


Thomas Bayes

Therefore, when an email arrives that includes these two keywords, the spam filter mark it as spam because with a 59% probability it is spam.
This is called "Naive Bayes" because the process of calculating the probability for each hypothesis is simplified. However, this does not mean that it is prediction capability is not strong, it is a very useful algorithm for many classification cases, we will describe a classification (predication) example shortly.

Gaussian Naive Bayes

The email example that we have explained here is dealing with discrete data. How about having continuous data and we intend to make a prediction model with Naive Bayes? If we assume *features are normally distributed* we can use the Gaussian Naive Bayes algorithm. If they are not normally distributed it is recommended to remove outliers to make them near normally distributed.

For example, let's assume there is a feature in spam email detection and it is the "*time of email arrival*", we call it "time" for the sake of brevity. We assume time is a continuous variable, for the continuous variable instead of probability we use their probability distribution function (PDF), which you can recall PDF (not the one from Adobe) by looking back at Chapter 3.
If our objective is to perform a prediction based on time, instead of writing $P(h \mid d)$ we could write. $P(spam \mid pdf(time))$

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.

Naive Bayes Prediction Example

Here we explain another prediction example in

another form. This example guarantees when you finish this section you are an expert in Naive Bayes and the soul of Thomas Bayes is praying for the sins you have performed or will perform in your machine learning career and not using Bayes theorem when it can answer you.

Assume you had a good friend, when you call him, he is always available for hanging out with you. Having a social interaction is very important for mental health. Nevertheless, watching a useless "TV series" or "playing online games" is also very delightful.

Every weekend, you check your favorite TV series web page, to see if the new episode of "how to waste my time" has arrived or not. Also, you open your phone and scroll the app market to see if there is a new game to download and play. You do this every weekend and you decide to use the Naive Bayes algorithm to be able to predict whether you will go to meet your friend on the weekend or stay home and spend your time on games or TV series.

Table 2 a. presents what you have recorded about your behaviors on previous weekends. Your friend would like to predict whether you are going to meet him the next weekend (based on the availability of a new game or new episode). Assuming the # sign will be used to mention the count of events and $\wedge$ use to say intersection (and), first we calculate the conditional probabilities as follows:

$$P(TV = yes) = 0.4, \quad P(Game = yes) = 0.6, \quad P(Friend = yes) = 0.5$$

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

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

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

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

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

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

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

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

Recall that earlier we 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 "meet friend" column. For example, for the four different combinations of "TV" and "Game" based on Table 9-2 we can write the followings:

TV = no, Game= no:

tv	game	meet friend	Predicted (meet friend)	Predicted (~ meet-friend)
no	no	no	0.24 ×	0.02
no	no	yes	0.24 ✔	0.02
no	yes	no	0.16 ×	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 ×	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 value are equal we can not make any judgement.

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

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 column) with the actual value (third column). We can calculate accuracy, precision, recall, and other metrics related to the accuracy, by comparison between the actual and predicted class results.

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 choose 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 $O(Nc)$.

NOTE:

* Since there is no parameter being used for Naive Bayes, there is no need to use an optimization method for finding a good fit of the parameter. Naive Bayes and similar models are referred to as parameter-free models.

* Keep in mind that Naive Bayesian can perform both binary and multi-class classification both.

* While dealing with the 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, which are models that describe conditional probability, e.g. $P(Y \mid X = ' something')$.

*k*th Nearest Neighbor

The algorithms we have explained, including regression algorithms and naive Bayes, were based on building a model. There are other types of algorithms that are still supervised algorithms but do not build a model to classify data. These algorithms use the whole dataset as the model and are referred to as **instance based learning.** A popular example of these algorithms is *k*th Nearest Neighbor (*k*NN) algorithm.

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 a need to learn mathematics we have described afterward. Now wake up, the real life is waiting for you. However, the good news is that *k*NN is very easy to understand and operates similarly to unsupervised learning algorithms. Therefore, if like the author of this book you like to run away from learning mathematics, enjoy learning *k*NN.

*k*NN operates based on the assumption that *similar data stay close together*. First, similar to all other supervised learning algorithms, we should label a few data points. Then, the algorithm employs a distance metric (e.g. Euclidean, Cosine, etc. check Chapter 4 to recall them) and assigns labels to the data points which do not have a label and are close to the labeled data. The distance functions that are used in *k*NN are similar to distance functions that are used in clustering algorithms, such as Euclidean distance.

This algorithm receives k as the input parameter, which specifies the number of neighbor data points. Assuming $k = 3$, check Figure 9-12 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 ones which are grey in Figure 9-1 b. do not have labels[2].

The kNN algorithm calculates the distance of each unlabeled data point into k nearest data points and decides about the label. For example, the algorithm chooses a data point that is marked with "?" in Figure 9-1 b to decide about its label (blue or red). Figure Figure 9-1 c shows that the are three data points close to this particular data point, two were red and one is blue. Therefore, the majority are red, and thus this data point will receive the red label as well (Figure 9-1 d). This

[2] To be honest, now it is three dimensional space, because another dimension or feature is blue/red color. It is better to refer to dimension as features, because having a dimension that includes either blue or red is not wrong, but doesn't make sense.

process continues until all data points received a label. Also, if a new data point is added to the dataset it will be treated as another unlabeled data point.

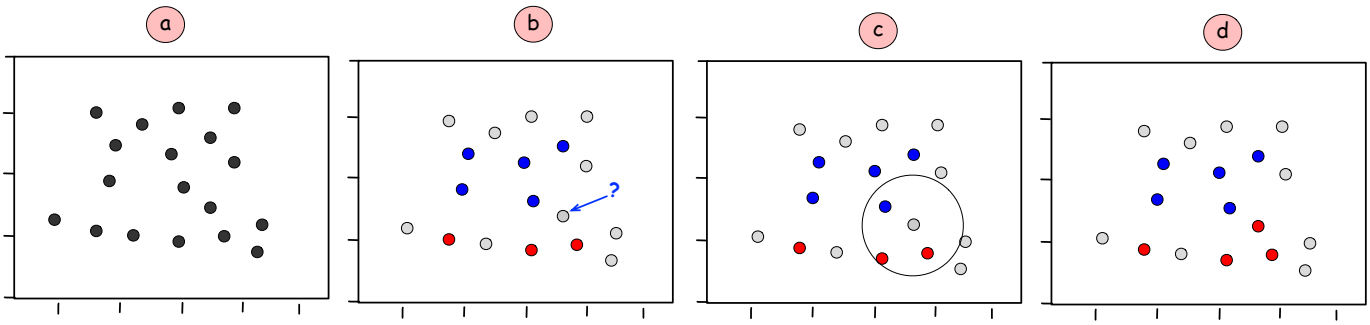


Figure 9-1: (a) The dataset without labels (b) few data points in the dataset has been labeled with blue and red color. (c) a datapoint have been selected that we need to know what label (red or blue) does this data point receive ?

Our lovely *k*NN belongs to the category of **lazy learning** algorithms because it stores all the data points and then uses them, which is not actively classifying data points. kNN has many useful 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.

A question arises now, how can we identify the optimal number of *k*? Similar to clustering techniques the best approach is to test with different values for *k* and choose the one that has the highest accuracy.

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 $O(kdn)$. The brute force kNN is very slow and therefore there are approaches that help kNN performs better. In the following, we explain three of these approaches, including Voronoi Tessellation, KD-Tree and Local Sensitive Hashing.

Voronoi Tessellation

Voronoi tessellation is the process to partition a 2D plane that includes several data points, into regions, in which each region presents the area that is closest to its related data point. Check Figure 9-2 a, we have data points on a 2D plane, and in Figure 9-2 b, the regions were drawn by a Voronoi Tessellation. The result is called the Voronoi diagram (Figure 9-2 b is a Voronoi diagram), and each of these regions holds one of the points in its center. If any other data point is added to a region, the closest point to the newly added data point is the center point of that region.

If a *k*NN algorithm sets $k = 1$ then 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. It improves the search time to *O(log n)*, but it is only good for low dimensional data (two dimensional data). For higher than two dimensional data, assuming d is the number of dimensions the computational complexity will be *O(d.n)*.

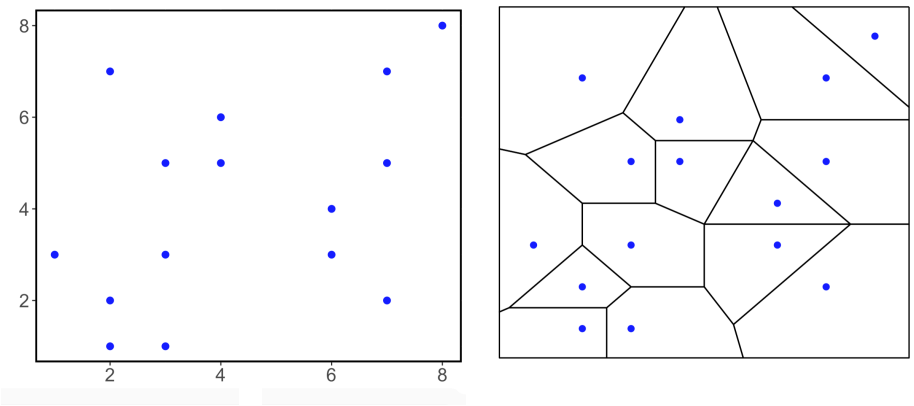


Figure 9-2: (left) Original dataset (b) Voronoi tessellation of the original dataset.

KD-Tree

If you recall from Chapter 6, we explained that one of the big challenges in search is to reduce the search space and trees are very useful to reduce the search space. K Dimensional Tree or KD-Tree is used for the kNN algorithm and efficiently distributes the search space to each point. Nevertheless, it is an approximation technique and we might miss some points which are the nearest neighbor.

Lets' 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 can plot the dataset as it has been shown in Figure 9-3 a. The process of KD-Tree partitioning starts by finding the *median* in one of the dimensions, e.g. here we have two dimensions so we choose x. 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 9-3 b. and you 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 are nodes with smaller x values, $\{(2,1), (1,3), (2,2), (3,1), (3,5)\}$.

Next, we switch to another dimension, i.e. y-axis, and get the median of y coordinates. Since our dataset, however, is partitioned into two segments we do once for each segment.

The values on the y-axis on the left segment of Figure 9-3 b are {1,1,2,3,5}, its median is '2'. Now, the next node to construct KD-Tree will be drawn on the y-axis at '2', the same process will be done on the right side as well (the median on the y-axis is 6). See Figure 9-3 c to check its results. Again as you can see from the tree on the right side of Figure 9-3 c the KD-tree algorithm goes into each branch and do the same (see Figure 9-3 d), until each branch has only one data point and the tree generation stops, as it is shown in Figure 9-3 e.

By having such a tree constructed, to search for the nearest neighbor the algorithm can navigate through the right branches instead of brute force search (check Chapter 5 if you can't recall the search on trees) and thus it can be used by the *k*NN algorithm to improve its search for the nearest neighbor.

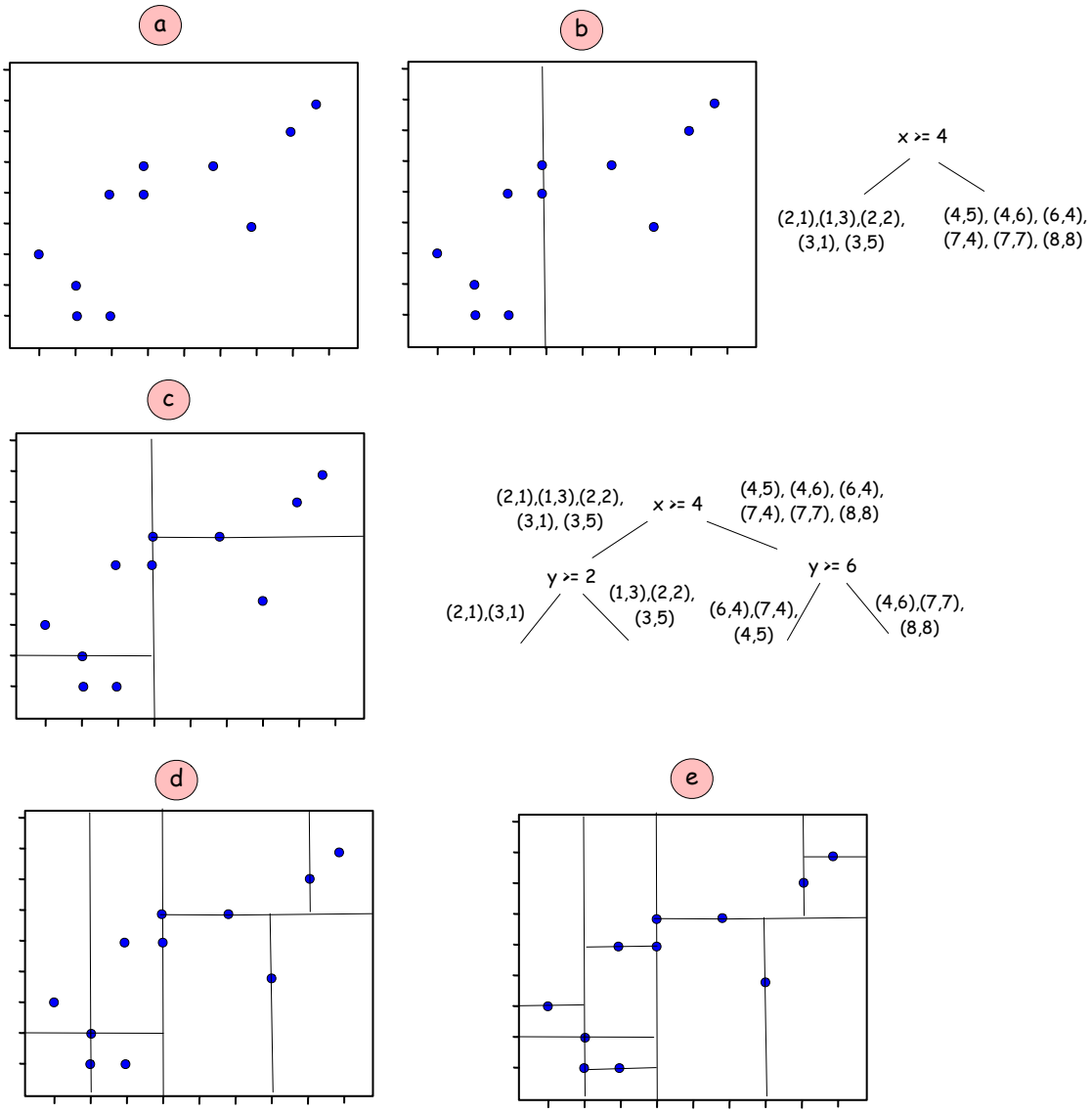


Figure 9-3: (a) The original dataset (b) the median has been chosen and the first branch of tree is constructed to separated the dataset into two sub datasets. (c) Now the dimension changed (Y axis) and the same process will be repeated (d) again the axis changed and the same process (e) no other data point left and each node in the final tree has only one data point.

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 point and k dimensions, the computational complexity of KD-Tree is *O (k.n log n)*.

Local Sensitive Hashing (LSH)

Local Sensitive Hashing (LSH) is widely in use in different applications including finding duplicate documents, search engines to find similar images, biologists use it to identify similar

gene expressions in a genome dataset, and audio and visual similarity detection which is used in authentication applications. Therefore, please switch to full attention mode while reading this section. Besides, usually in real-world cases, we have high dimensional data and to be able to use *k*NN usually LSH is used to implement kNN.

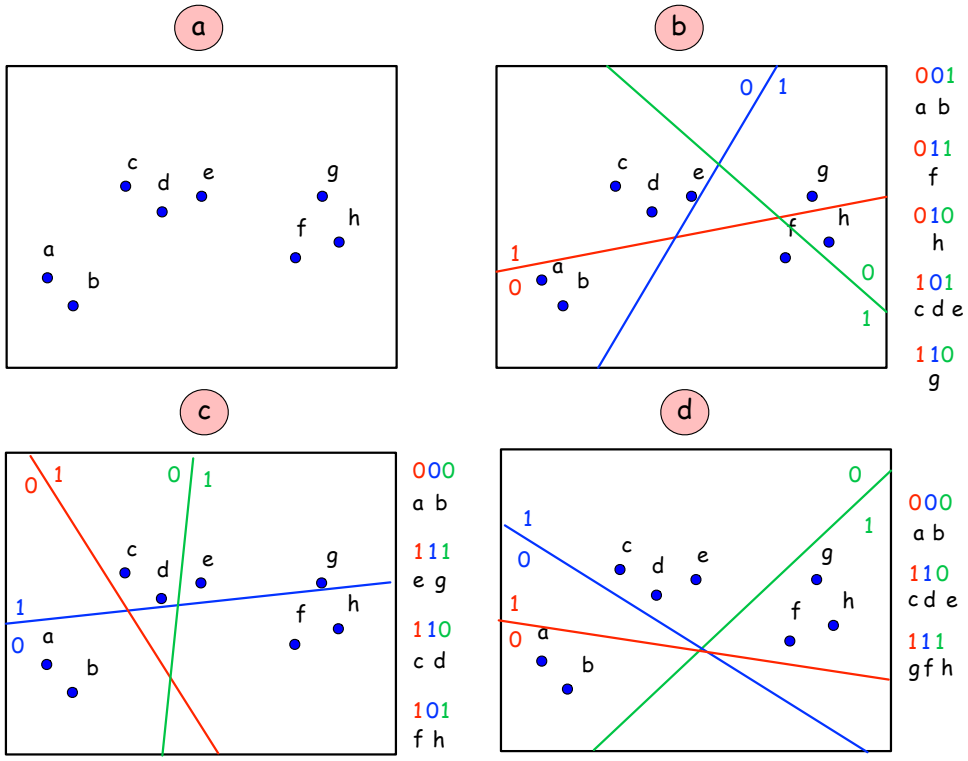


Figure 9-4: (a) The original dataset (b) Three lines drawn as a hash function to separate each data point (c) another three separator lines drawn and (d) for the third time another set of line drawn. Note the result of this hashing is written on the right side of each figure.

LSH hashes similar data objects into the same bucket. Somehow LSH is very similar to clustering and thus the *k*NN algorithm can use it to improve its performance by searching buckets, instead of comparing the new data object to all existing data objects. If you recall from Chapter 5, we explained that a good hash function should avoid a collision, but LSH works in the opposite and it tries to maximize collisions between similar data objects. In simple words, it tries to collect similar data objects into buckets.

LSH receives the number of *separator hyperplane* or *line*, 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 we end up having maximum 2^k regions (buckets) to search. The small regions that are generated by the separator lines are called buckets. Then, it separates the dataset into different regions (buckets) and when a new data object arrives it compares it only to the other data objects in that bucket. The algorithm runs this process *l* times.

For example, Figure 9-4 a. shows an original dataset that is converted into 7 buckets in Figure 9-4 b. Each new data point that arrives, will be compared only to its own bucket. This is good but we might miss some data points. For instance, in Figure 9-4 b. 'f' belongs to 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 buckets the search space *l*=3 times, as we can see in Figure 9-4 c and d.

Considering the result of this bucketing we can say *'a'* and *'b'* stays in the same region, *'c', 'd'* and *'e'* stays in the same region, and also *'f'* and *'h'* are in the same region. However, we can't judge for 'g' , with only *l*=3. We should make *l* larger to be able to identify neighbors of 'g'. When a new data object is received by the algorithm, *k*NN can use LSH and assign this data point into 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. same region). For example, if the hash code assigns region 000 to the newly arrived data object, based on Figure 9-4, 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.

NOTE:

* While deciding about a *k* in *k*NN, there is no optimization related algorithm existed for that. The best way to decide about *k* is to perform the parameter sensitivity analysis or hyper parameter tuning, i.e. experimenting and checking the result until we get the best parameter.

* It is recommended to rescale the data to have all features in the same numerical range, while using *k*NN algorithm.

* KD-Tree is also used to order the multidimensional space and facilitate search in the multidimensional environment.

* One of the biggest advantage of kNN is its non-linear decision boundary that can handle even classifying complex shapes. For example, take a look at Figure 9-5, which shows a complex classification required to separate the class of blue from red data objects.

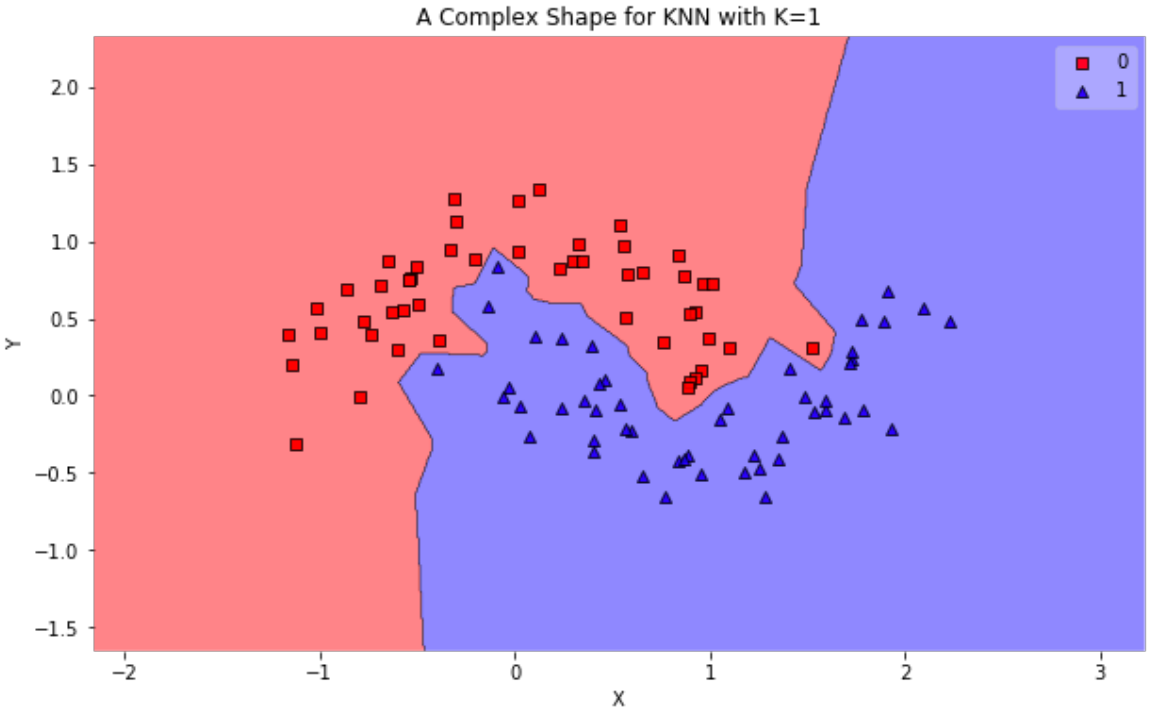


Figure 9-5: *k*NN algorithm that can classifies fairly complex structure as you can see from this picture. You can see from this picture that blue and red dots are interviewed but the kNN can successfully classifies them (look at red and blue region).

Support Vector Machine (SVM)

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

To explain SVM let's start with a very simple example and use SVM for binary classification. This example will give us intuition about the rationale of SVM. Then و we explain how a binary classifier extends into a multi-label classifier. Take a look at Figure 9-6 a. it shows fairly well separated datasets of blue and red dots. SVM can draw a line and separate them, lets randomly draw a line that separates these two sets from each other as it is shown in Figure 9-6 b. Now, we would like to test the algorithm, and let's assume we use one new data point as a test, the grey data points in Figure 9-6 c. You can see this data point stays in the blue region which seems not

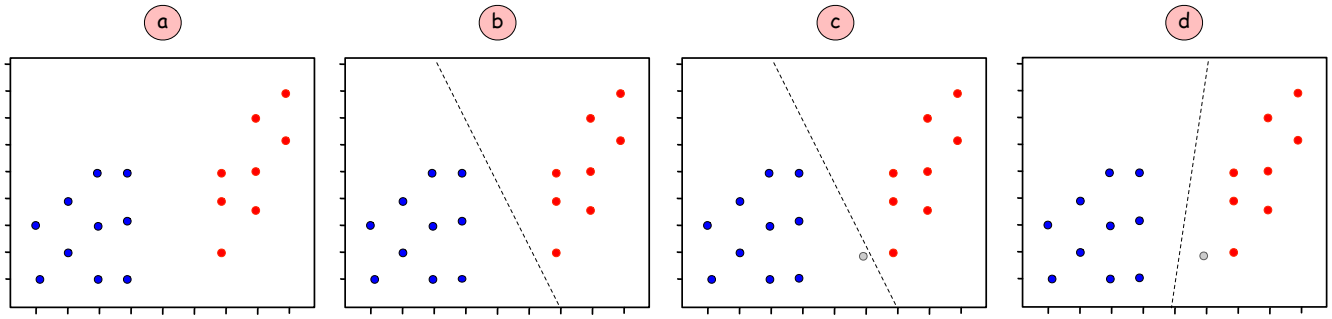


Figure 9-6: (a) sample data points that we intend to find a line to classify the space into the blue region and red region. (b) A random line has been drawn that it seems the space is well separated. (c) a new data point (from test set) is added and based on the drawn line if went into the blue region. However, it doesn't seem at the right place (d) another separator line is drawn and the data point fits into the red region.

correct, because it stays closer to the red data points. We draw another separator line in Figure 9-6 d and this line 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 do this separation properly.

Let us remind you that this data has three features and is drawn in 2D space, if we have more features and are drawn in 3D space, it will not be a line, it will be a **hyperplane** that separates regions. If we have 1D data points it will be a single data point. 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 can be written as $\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots \beta_p x_p = 0$. You might remember from Chapter 8 that β_0 is intercept and others are the slopes of the hyperplane. A vector of $\overrightarrow{\beta}$ is called a **weight vector** and some books use $\overrightarrow{w}$ to present it. Thus, in some literature 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: $\overrightarrow{w} \,.\, X + b = 0$

Assuming that we need to specify the new data points' label as blue or red (classify data points from the test set), we have a binary classifier, i.e. $y = -1, \quad y = 1$. Unlike, other binary classifiers instead of 0 and 1 in SVM we work with -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 formalizes it with the following equations:

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

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 $f(x^*) > 0$ then the new test data point x^* will get the label 1 (e.g. red) and if $f(x^*) \leq 0$, then x will assign 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 to assign to this data point (-1 or 1). Therefore, the output will be something like follows: $\{x_1 = -1, x_2 = 1, x_3 = 1, \ldots\}$

Now the question is how to draw this separator hyperplane? Or rephrasing it a bit sexier: how to identify weight vectors $\vec{\beta}$ or $\vec{w}$ of the hyperplane equation?

The answer is to use the **Maximal Margin Classifier**. Here, **margin** refers to the *distance between a data point and the separator hyperplane*. Therefore, the maximal margin classifier is using *a hyperplane that has the largest margins (largest distance between data points and hyperplane)*. Figure 9-7 a. shows another dataset with blue and red classes. It looks like the street, the centerline is called the **decision hyperplane** and the sidelines in parallel to it are

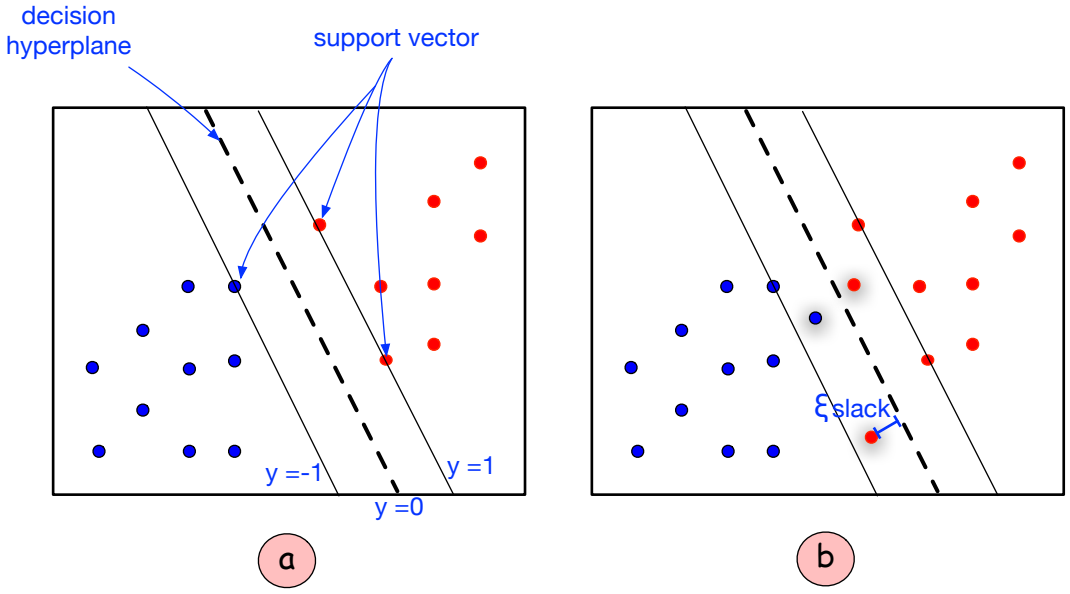


Figure 9-7: (a) sample data points that by using maximum margin classifier support vectors (points on the line construct support vector) and decision hyperplane were identified and highlighted. (b) three sample data points (they have shadow) that are stayed inside support vectors, and one data point stayed in the wrong side of the hyperplane (slack variable). A good margin classifier should reduce these issues as much as possible.

called **support vectors**. Figure 9-5 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 them.

The separator hyperplane will be identified with algorithms such as the *Sequential Minimal Optimization* algorithm (check Chapter 8 to recall what is optimization), which, we will drown under a huge pile of mathematics if we explain them here. We skip explaining the details of SVM optimization, and you might not need to learn it unless you would like to design an optimization algorithm or you are doing research on mathematical theories of optimization.

SVM optimization algorithm tries to find the hyperplane based on two objectives:
(i) it makes the maximum margin possible (making the street in Figure 9-7 wider) while limiting margin violations. Data points between vector spaces mean margin violation (see Figure 9-7 b). However, since this phenomenon is unavoidable in real-world datasets which are noisy, we tolerate it and call the classifier that tolerates these points a **soft margin classifier**.
(ii) it has a constraint and this constraint indicates 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. Otherwise, this hyperplane can stay far away on the other side of the world and thus makes a very large margin, but it doesn't make sense. For example, take a look at Figure 9-8, the margin is super large, but it does not classify the dataset.

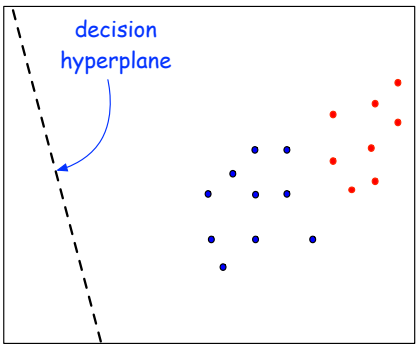


Figure 9-8: A decision hyperplane that has a very large separator margin, but it doesn't make sense. Because the separator constraint does not existed.

The loss function that is used by optimizations for maximum margin classification algorithms (such as SVM) is called **hinge loss**. Some implementation of SVM asks you to specify the 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 9-7 b. In reality, there are outliers and data points that can not stay in the correct class and stay 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 are staying between support vector lines or on the wrong side of the hyperplane are called **slack variables**[3]. Figure 9-7 b highlights three slack variables three between support vectors, and one of them stayed in the wrong side of the hyperplane. The objective function should make slack variables as small as possible, and reduce hard margin violations as much as possible.

There is a C hyperparameter that controls the magnitude of tolerance that is allowed on a slack variable, $C = 0$, which means that no slack variable is allowed and the larger the C means more slack variables are allowed. Figure 9-9 presents two different values for C. Setting $C=0$ means that no slack variable is allowed and the other one allows the slack variable. In other words,

[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: $x + 2y + \xi - 3 = 0$

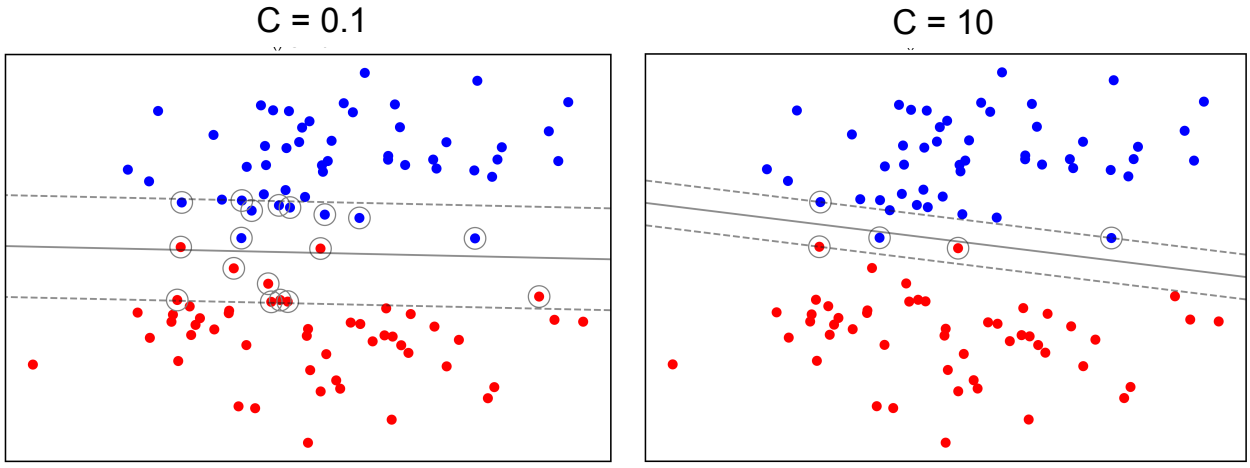
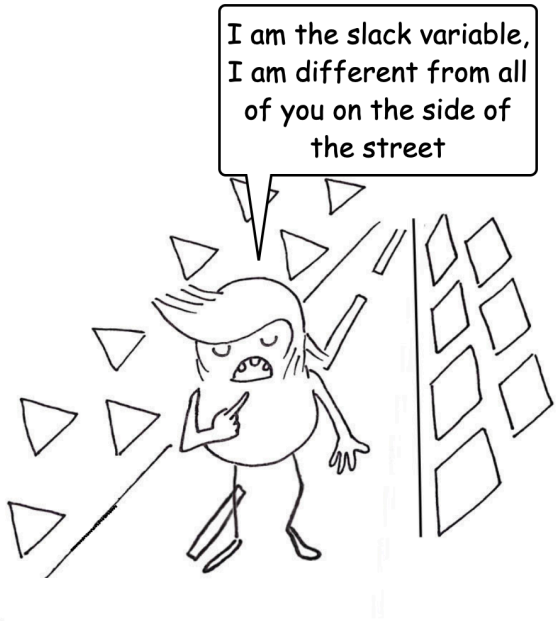


Figure 9-9: (a) A sample dataset and C=0 presents a very small margin. Therefore, the variance is low, but as you can see we have high bias. (b) A larger value has been set for C=10, and the bias is getting lower, but variance has increased.



setting a *C* hyperparameter is the bias-variance trade off. The higher value for *C* (larger street) allows a higher number of margin violations, which means high variance but low bias. On the other, hand smaller amount of *C* (smaller street) leads to having high bias and lower variance.

Handling non-linear data with Maximal Margin Classifier

Up until now all examples we have defined are easily separable with a line. Now, take a look at Figure 9-10 a, how can we use a maximum margin classifier and draw a hyperplane to classify it?

To handle non-linear data we need to 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 dimensionality

reduction is very useful, but in this scenario, we need to increase the dimension of the data to be able to classify it.

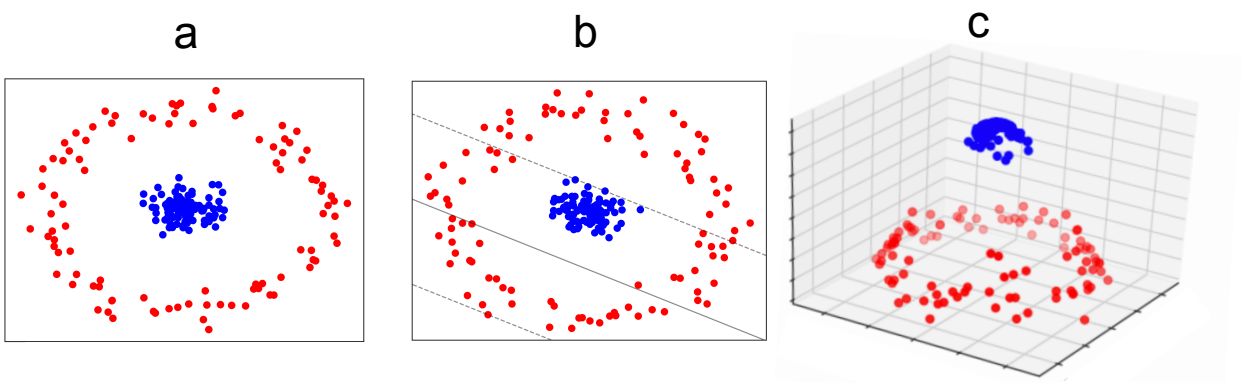


Figure 9-10: (a) A sample dataset that has two different labels. (b) The maximal margin classifier can not handle the data in two dimensions and you can see the maximal margin classifier line can not separate the data. (c) After applying the kernel function and brings the data into a higher dimension, a hyperplane can separate the blue from red dots.

To implement such a transformation for the data we use the **Kernel** function. Therefore, the kernel function is responsible to *transform a non-linear inseparable dataset into a linear and separable dataset.* For example, without applying a kernel function in Figure 9-10 a, we can see the maximal margin classifier lines cannot be classified as blue and red dots, as it has been 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.

The Support Vector Machine (SVM) algorithm uses a computationally efficient kernel function that transforms the input dataset into non-linear higher dimension data.

Kernel Functions

As we have explained before, the kernel function is used by 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 that brings our dataset into a higher dimensional space that was not separable with a linear hyperplane previously. After the transformation into the higher dimensional space, the SVM can separate them with a linear hyperplane (see Figure 9-10).

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

If the data points of the dataset have many features (X has many members), a simple polynomial transformation could make it fairly large representations, which will be X^2, what happens if still the algorithm can not separate them? We can increase the degree of polynomiality. For example, three degrees of polynomials, and we will end up having X^3 features, which are very complex, e.g., we have 10 features and 3000 data points, and $10^3 \times 3000 = 3{,}000{,}000$ are too many features to deal with and computationally very in-efficient. Meaning, that this polynomial transformation increases the computational complexity and we need a subtle way to handle resolve this problem.

In practice, since transforming data is a computationally expensive process, a kernel function *does not implement the transformation into a higher dimension*, but instead, uses a mathematical trick and *calculates the similarity between each data point in a higher dimensional space*, this trick is known as **kernel trick**.

In fact, a kernel function is a *similarity function* that measures the similarity between two sets of data to be able to assign the correct label to unlabeled data. Therefore, one feature set is the labeled data and the other feature set is the unlabeled data.

The kernel function can be represented as **dot products** or **inner product** of input labeled dataset and unlabeled input dataset, or simply two sets of information that we intend to compare. In other words, we can simply convert the high dimensional dot product to a simpler dot product between members of the feature vector. 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 $a \,.\, b = 3 \times 1 + 2 \times 4 + 4 \times 0 = 11$

The good thing with the dot product is that instead of computing the dot product between high dimensional features, we can calculate a dot product in low dimensional space and raise it to the power of polynomial. For example, assume we have a labeled train dataset $X = \{x_1, x_2, x_3\}$and we have an unlabeled test dataset $X' = \{x_1', x_2', x_3'\}$. Having two degrees of polynomiality their transformed version will be as follows: $\Phi(X) = \{x_1^2, x_1x_2, x_1x_3, x_2x_1, x_2^2, x_2x_3, x_3x_1, x_3x_2, x_3^2\}$ and $\Phi(X') = \{x_1'^2, x_1'x_2', x_1'x_3', x_2'x_1', x_2'^2, x_2'x_3', x_3'x_1', x_3'x_2', x_3'^2\}$. Now we would like to compare $\Phi(X)$ with $\Phi(X')$ to be able to label $\Phi(X')$ set from $\Phi(X)$ the labeled set. You see that we have too many features to compare between these two sets because from three dimensional space we moved into nine dimensional space.

---

[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 $F(x, y) = F(y, x)$, there is a function ϕ that can bring $F(x, y)$ into another dimension that $F(x, y) = \phi(x)^T \phi(y)$

By using their dot products we get the same result. Therefore, a kernel function can be written as follows: $K(X, X') = x_1 . x_1' + x_2 . x_2' + x_3 . x_3'$. Interestingly, if we do the math dot products of X and X' is equal to the dot products of $\Phi(X)$ and $\Phi(X')$, we can write $K(X, X') = < \Phi(X), \Phi(X') >$. We do not write the math in detail for you, because we are lazy and you can substitute numbers and compare them on your own.

Let's summarize how a kernel trick works: to avoid comparing the similarity between two datasets in the higher dimensional space, the kernel function uses a dot product in the lower dimensional space to compare two datasets together, and therefore does not compare them in the higher dimensional space.

Now we have understood the advantage of the kernel trick, congratulation, you learn another secret in mathematics. There are several popular kernel functions used for SVM. The popular ones are listed as follows.

Linear kernel: $K(X, X') = X . X'$, which is a simple dot product.

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

Gaussian Radial Bias Function (RBF) kernel: $K(X, X') = exp(\gamma \, | | X - X' | |^2)$, γ is a hyperparameter that can present the curvedness of the separation area, a γ gamma presents a more linear hyperplane. Larger gamma makes the hyperplane more curved.

Sigmoid kernel: $K(X, X') = tanh(\gamma X^T X' + r)$, r presents the polynomial coefficient and γ is a hyperparameter that presents the curvedness of the separation area (similar to the Gaussian RBF kernel). *tanh* is a hyperbolic tangent function.
RBF is the most popular kernel function for SVM algorithm. Take a look at Figure 9-11, which shows an SVM algorithm run with three different values for γ. Gaussian RBF kernel is a very popular kernel function for SVM. Aurelien Geron [Geron '19] states that γ acts as a regularization parameter. If the SVM model is overfitting reduce it, if the model is underfitting increase it.
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. In 2022, we hear some psycho recruiters ask for these Kernel equations at job interviews for data science.

When we are using SVM algorithm, usually we can specify the kernel function as a parameter of the SVM algorithm and hyperparameters of the kernel function. Now a question might arise, how does the kernel function decide about the polynomial degree? Or how polynomial parameters were determined? By using cross validation the algorithm performs hyperparameter tunning and some implementations automatically chose the best possible parameters for the selected kernel method. In some other implementations, we should give the parameter as an input variable. Therefore, this is something dependent on the SVM library or package we are using.

Usually, 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.

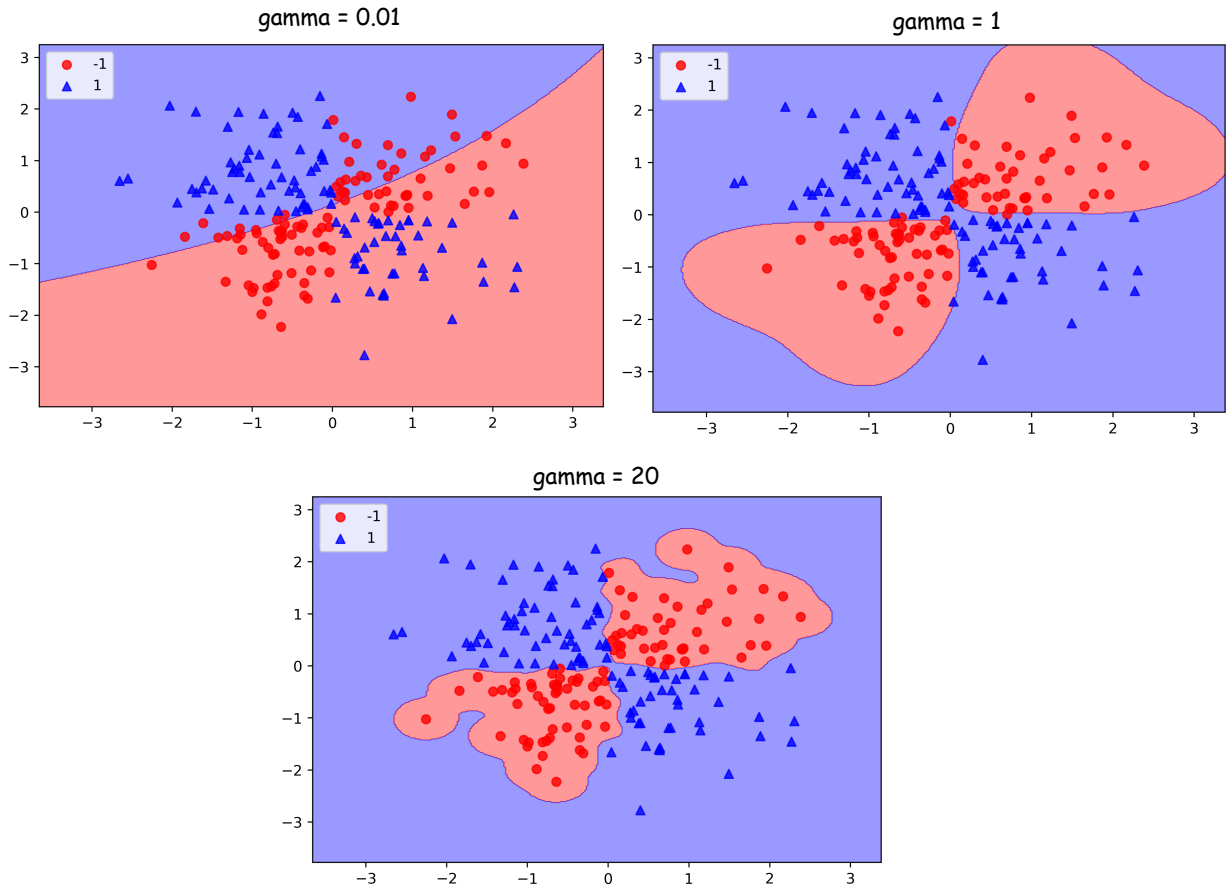


Figure 9-11: three different value gamma and hyper plane changes based on three different gamma parameters for RBF kernel function. The higher gamma makes the better separation, but setting gamma too high, like the most right figure makes the model prune to overfitting.

Multi-label classification for SVM

Binary classification is simple. What if we would like to run SVM classification and classifies a dataset based on four labels {blue, red, green, orange}? Once the SVM performs binary classification between blue and others {red, green, orange}. Then, it classifies between red and others, then between green and others and at last between orange and others. Each data point is assigned a label in each iteration, for example, data point x a gets three times green and once orange. This means at the end the x will receive green as its label. This style of handling multi-label 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. Based on the quadratic optimizer the following equation can be written for the maximal margin classifier.

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

In this equation α_i (the Lagrangian multiplier) and β_0 (bias) parameters will be determined automatically 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 does it receive).

SVM Computational Complexity

Calculating the computational complexity of SVM is very much 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, but keep in mind it is one of the algorithms that are highly accurate but with the cost of high computational complexity.

NOTE:

* Both kNN and SVM can handle nonlinear data structure and when a dataset is small, they are good options for classification.

* kNN is not good for higher dimensional data, SVM can also classify datasets whose features (dimensions) are higher than the number of data points. Therefore, it is more robust for handling high dimensional data.

* SVM is sensitive about scaling, and scaling the data will result in a change in SVM behavior. Therefore, if your dataset is changing frequently be careful while using SVM.

* The support vector classifier (SVC) algorithm is the support vector machine with a polynomial kernel, and its degree is $d = 1$.

* 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 one performs better or using Boosting, Bagging, and Stacking methods, which we will explain later in this chapter.

* If you remember we discussed that if the optimization has a convex shape, local minima is the global minima as well. SVM parameter optimizations is also a convex shape function.

* SVM algorithm is slow, but it is fairly robust against overfitting, which is a common problem in classification algorithms.

Decision Trees

Back in Chapter 5, we have explained that trees are used to reduce the search space and thus make the search algorithm faster and more efficient-resource.

Decision trees are a group of machine learning algorithms that can do both classification and regression while extracting human understandable rules as their model. They split the search space (dataset) into smaller spaces, and search the smaller space instead of the entire dataset, and because of this feature, usually, they are resource efficient. The first practical version of the decision tree was proposed in the early 80, i.e., CHAID [Kass '80]. A few years after CHAID, in 1984, Breimanetal [Breimanetal '84] introduces CART algorithm, which is another popular decision tree algorithm that is still in use. Two years after that, ID3 [Quilan '86] has been introduced by Quilan and then Quilan proposes C4.5 [Quilan '94]. All of these four algorithms are still in use. After these basics trees, gradient boosting trees [Mason '99] have been 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 on decision trees again.

At the time of writing this book, deep learning algorithms have the highest accuracy along with gradient boosting algorithms. Nevertheless, since most decision trees are **explainable** algorithms, and operate on a small size dataset, unlike deep learning algorithms, they are very popular algorithms.

	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	?

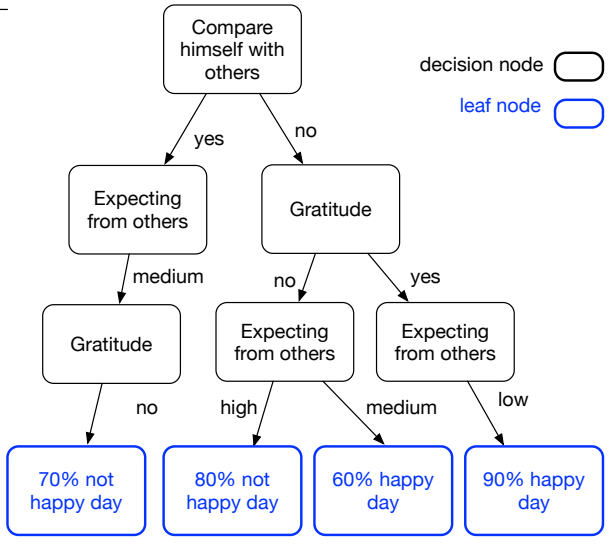


Table 9-4: A sample dataset, we like to predict the happiness status of the test dataset. On the right side, the algorithm constructs an example tree, and by tracing this tree the algorithm can make the decision about the test dataset (the ? marks of the blue record in the table).

A decision tree, similar to another supervised algorithm, uses a training dataset to construct a tree. Then, after the tree has been constructed, the algorithm uses the tree to assign labels to the test dataset.

For example Table 9-4 presents the information that Mr. Nerd collects daily about himself and what he is thinking about his behaviors and attitudes. He would like to know when he went to the bed and thinks about his day, whether he feels happy or not? He likes to create a prediction algorithm that predicts the label (happy is the label) before he ends his day based on the data (test dataset) he gives to the algorithm. Note this tree is just an example to show the decision tree and not a representation of the table on the left.

In a more technical sense, after the training phase, the decision tree receives a vector of features (the test record in Table 9-4, which is presented in blue color), and the decision tree starts to navigate its branches until it identifies the missing labels, i.e., '?', in Table 9-4.

Decision trees have three important characteristics, that we should keep in mind.
(i) Decision tree algorithms are operated based on the policy of *splitting 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 completely. Next, the test dataset will use this tree to label its records.
(ii) The **leaf nodes** in decision trees are presenting the **outcome** of the path from the root to that node. All nodes in between are called **decision nodes,** because the algorithm uses them to make a decision and reaches a leaf node at the end. The decision tree of Table 9-4 shows leaf nodes in blue, and decision nodes in black.
(iii) Decision tree is hyperparameter independent and this makes them very popular algorithms for real-world applications. Even, decision trees results are a set of rules, and this makes them explainable, which is not the case for neural network algorithms. If you encounter a fancy word like **explainable AI (xAI)**, it means using a machine learning algorithm that its decision process is explainable and not BlackBox.
Please, fasten your seat belt we are about to land on the big planet of decision trees.

Iterative Dichotomiser 3 (ID3)

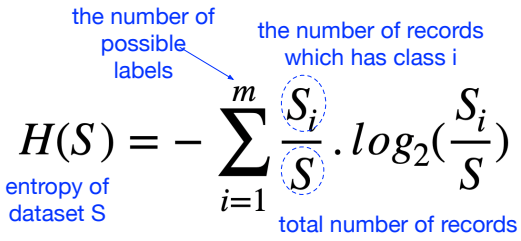
ID3 is one of the basic algorithms that iteratively dichotomizes (divides into two groups) the dataset at each step. Recall that we have explained the process of decision tree creation is to split the training data into smaller subsets.

Take a look at the tree in Table 9-4, there 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 that can better split the data. By better 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.

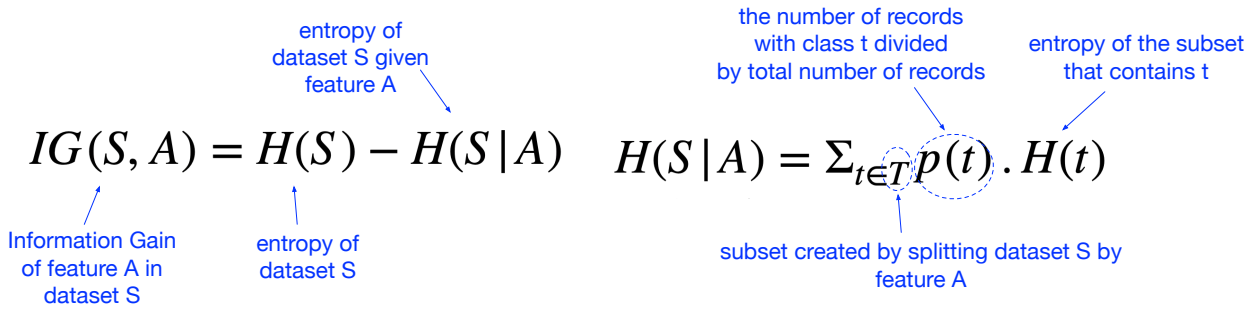
ID3 uses **information gain** which is based on **entropy** to select the best feature. Entropy is a number that quantifies the uncertainty or disorder (check Chapter 3 to recall it). Therefore,

entropy specifies (in number) how well a given feature, separates the dataset records based on the label.

Assuming m is the number of possible labels (classes) in the training set, S denotes the dataset, p_i denotes the probability of having class i (the number of records with class label i divided by total number of dataset records), the entropy $H(S)$ is written as follows:



Here $H(S)$ refers to the *average amount of information required to identify the class label of a new record.* Each column in Table 9-4 is presenting features of the decision tree, and we can write Information Gain (IG) for the feature A (a column of data) as follows:



In simple terms, we can assume IG as the differences between original information, i.e. $H(S)$ and the required information, i.e. $H(S|A)$. Now, we learn 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) that has the highest split against one of the possible values of the labels. To find this feature, it measures the IG for every single feature (column) of the dataset.

In the second step, it splits 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 remained 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 Table 9-4. 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 (✗) for this column. Therefore, its entropy will be 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 calculated IG for each column (feature).

We have the following IGs calculated for each feature (column in Table 9-4). The sign # is used to refer to the number of frequencies.

"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

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

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

$IG(S, Gratitude) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - (6/12)0.918 - (6/12)0.918 = 0.04$

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

$H(S) = -(4/4).log_2(4/4) - (0/4).log_2(0/4) = 0$

"Compare himself with others": no, #rows: 8, #happy:2, # not happy:6

$H(S) = -(2/8).log_2(2/8) - (6/8).log_2(6/8) = 0.811$

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

$IG(S, Compare) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - 0 - (8/12)0.811 = 0.4193$

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

$H(S) = -(0/4).log_2(0/4) - (4/4).log_2(4/4) = 0$

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

$H(S) = -(3/5).log_2(3/5) - (2/5).log_2(2/5) = 0.97$

"Expecting from others": low, #rows: 3, #happy:3, #not happy: 0

$H(S) = -(3/3).log_2(3/3) - (0/3).log_2(0/3) = 0$

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

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

$= 0.96 - 0 - (5/12)0.97 - 0 = 0.55$

We understand that the largest information gain (0.55) belongs to Exp. This means that the dataset can be split on this feature, because the Exp feature provides the best split, in comparison to other features (Comp, Gratitude). Therefore, we start to construct the table as it is shown in Figure 9-12.

Now we have three splits of data, and thus three other datasets to analyze, at it is shown in Table 9-5. However, two branches (low and high) are not splittable based on the "Happy" status and thus the algorithm marks them as a leaf node. They are shown in blue color in Figure 9-12. Still, however, we can split "Exp" when it is medium. In other words, when a new data point arrives in the system, the algorithm first checks its "Exp", if it is "low" or "high", it can easily decide about the happy status. Otherwise, if it is "medium" the algorithm should go deeper into the three. This means again it should identify which column has the highest amount of 'yes' or 'no' for the happy attribute, and split the dataset (the smaller one in the bottom of Table 9-5) based on them. We do not further explain the tree construction, and the rest will be done the same until no

decision node remained and we end up all branches in a leaf node.

Chi-square Automatic Interaction Detector (CHAID)

Back in Chapter 3, we have explained the Chi-square can be used to identify the relationship between two categorical variables, and here we benefit from that attribute to split the tree. Just as a reminder, the following equation is used to calculate Chi-Square value, in which O stayed for observed and E stayed for expected values.

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

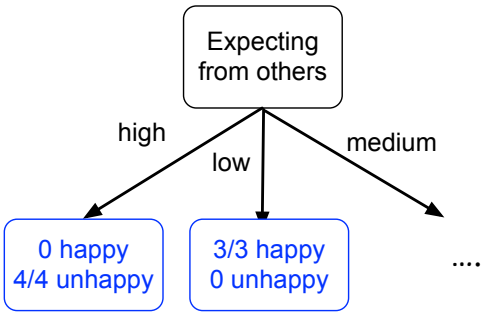


Figure 9-12: Expecting from others, show it has the highest IG and thus we split the dataset based on this. Since two children are not splittable anymore (the blue ones), they will be marked as leaf node and again the algorithm starts to split the dataset (which it's Expecting from others is medium).

CHAID uses the Chi-square test (for categorical variables) and F-test (for numerical variables but rarely CHAID is used for numerical values) to calculate the Chi-square value of each feature (each feature is a column) with the target column. Then, it chooses the split based on the highest Chi-square value. In other words, first, the algorithm calculates the Chi-square for each feature. Then, it chooses the feature with the highest Chi-square and split the table. This process continues until all remaining features can

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 Figure 9-11

Table 9-5: The result of splitting the original dataset in Table 9-4 based on the "Expecting from Others" column. Now we have three datasets, but the two on the top are leaf node, because we cannot further split them.

no longer be split, and thus they are all considered to be the leaf nodes (no decision node remained).

Back to our Table 9-4 example here we need to calculate the Chi-square for each feature, including (Gratitude, Comp. and Exp.). This is presented in Table 9-6.

For the "Gratitude" feature "Comp.", and "Exp." features we have Table 9-6, in which their Chi-square has been calculated. Based on these results the best feature to split the table is "Gratitude", because it has the largest Chi-square. It might sound a bit strange why we do not get "Exp." as the highest Chi-square like ID3. The reason is that each algorithm operates in a different way, and because of this phenomenon for solving a solution we should never rely on one single algorithm, we will experiment with many different versions, and do cross-fold validation until we are able to choose the algorithm with the highest accuracy.

The CHAID algorithm starts to create a tree with the "Gratitude" as the root node. However, since both "Gratitude" values are still splittable, both nodes are decision nodes (no leaf node) and the algorithm splits them further.

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy	
Gratitude	X	2	4	6	(6 x 7) / 12 = 3.5	(6 x 5) / 12 = 2.5	Chi-square happy: (6-3.5)^2/3.5 = 1.785
Gratitude	✓	5	1	6	(6 x 7) / 12 = 3.5	(6 x 5) / 12 = 2.5	Chi-square unhappy: (6-2.5)^2/2.5 = 4.9
Total		7	5	12			**Total: 6.685**

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy	
Compare himself with others	X	3	5	8	4.66	3.33	Chi-square happy: (8-4.66)^2/4.66 = 2.39
Compare himself with others	✓	4	0	4	2.33	1.66	Chi-square unhappy: (4-2.33)^2/2.33 = 1.19
Total		7	5	12			**Total: 3.58**

		Total Happy	Total Unhappy	Total Observed	Total Expected Happy	Total Expected Unhappy	Chi-square happy	Chi-square unhappy
Expectation from others:	high	0	4	4	(4 x 7) / 12 = 2.33	(4 x 5) / 12 = 1.66	0.69	1.35
Expectation from others:	medium	4	1	5	(5 x 7) / 12 = 2.91	(5 x 5) / 12 = 2.08	0.87	1.69
Expectation from others:	low	3	0	3	(3 x 7) / 12 = 1.75	(3 x 5) / 12 = 1.25	0.41	1.02
	Total	7	5	12				**Total: 6.03**

Table 9-6: Chi-square calculation for the each feature.

In the split the algorithm deals with two subsets of the dataset, one subset has only "Gratitude" equal to "yes" and one subset has only "gratitude" equal to "no". These steps of spiting the dataset continue until all nodes are leaf nodes. You can do the rest on your own and it does not

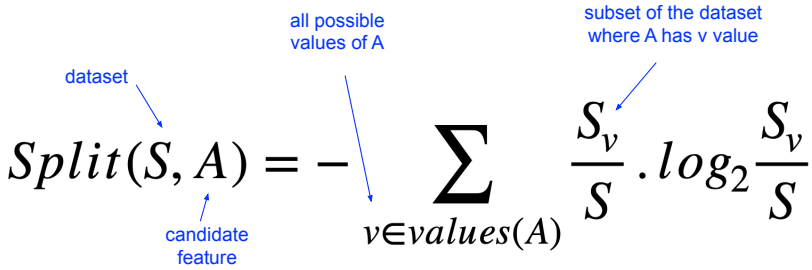
make sense to explain the rest of the algorithm in detail. Save your brain energy for more algorithms, still, two more decision tree algorithms remained, and then we switch to boosting and bagging algorithms, which are extremely important.

C4.5

It is an extension of ID3 algorithm and is developed by the author of ID3 algorithm. Since it is very popular 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 also **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 deciding by using information gain C4.5 calculates the GainRatio for all features and decides on the split based on the highest gain ratio.

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

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



Let's rewrite Table 9-4 with continuous data in Table 9-7 to see with an example how C4.5 creates the decision tree.

We show an example how to decide on the split for Expecting from other fields. First, we need to calculate the entropy of Happy for Table 9-7 and it will be 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 we have the entropy of "Happy" and we should find which column has the largest split on "Happy" so we calculated IG for each column (feature).

We have the following information gains calculated 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

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

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 9-7: (left) the original dataset (right) the dataset is ordered based on the Excepting from others which is continuous. Every time a single feature is going to be selected for split information calculation the dataset should be ordered based on that feature.

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

$IG(S, Gratitude) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - (6/12)0.918 - (6/12)0.918 = 0.04$

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

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

$$GainRatio(S, Gratitude) = \frac{IG(S, Gradtitude)}{Split(S, Gratitude)} = 0.04/1 = 0.04$$

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

$IG(S, Compare) = H(S) - (S_{yes}/S)H(S_{yes}) - (S_{no}/S)H(S_{no}) = 0.96 - 0 - (8/12)0.811 = 0.41$

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

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

$GainRatio(S, Compare) = 0.41/0.9 = 0.45$

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

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

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

"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

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

"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. But 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."). Assume "Expecting from others" for 0.8 receives the highest gain ratio. Therefore the decision tree starts with something like Figure 9-13, 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 is using a recursive approach similar to other decision trees to split the dataset into smaller units. CART uses **Gini split (Gini index or Gini Impurity)** to determine the best split.

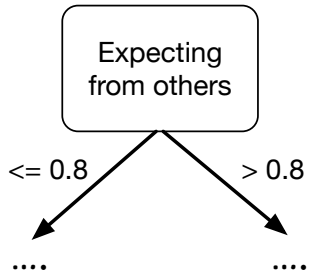
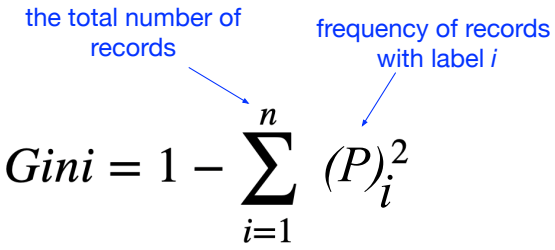


Figure 9-13: C4.5 decision tree result example "Expecting from others" for 0.8 has the highest gains ratio.

Gini split quantifies the impurity of each features. It 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. Therefore, the algorithm calculates the Gini index for each feature and chooses the lowest one for the split.



It can handle the continuous data in a similar fashion that C 4.5 can handle it, check the example we explained for "Expecting from others" in C 4.5 algorithm description.
Let's calculate the Gini index for the example presented in Table 9-4. Similar to other algorithms the algorithm calculates the Gini index for each feature (column) and then it selects the feature which has the lowest Gini index for the split.
We can calculate the Gini index of "Gratitude" as follows:

$$Gini(gratitude = yes) = 1 - (5/6)^2 - (1/6)^2 = -0.27$$
$$Gini(gratitude = no) = 1 - (4/6)^2 - (2/6)^2 = 0.44$$
$$Gini(gratitude) = 0.44(6/12) - 0.27(6/12) = 0.085$$

The Gini index of "Compare himself with others" will be done follows:

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

The Gini index of "Expecting from others" will be done follows:

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

Now it is time to decide about the lowest Gini index, which the smallest one is for $Gratiude = 0.085$. Therefore, the tree starts the first split by using gratitude, 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.

Decision trees are one of the most useful classification algorithms and state-of-the-art classification algorithms, built based on decision trees. The remained of this chapter explains them.

A question might arise while working with decision trees when to stop creating the tree? One answer is to define a minimum number of records under a leaf node, if the dataset remained 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.

Another approach is to specify a maximum depth for a tree. Note, that the larger tree depth leads to more complex inferences from the decision tree. Therefore, a shallow tree is preferred over a deep

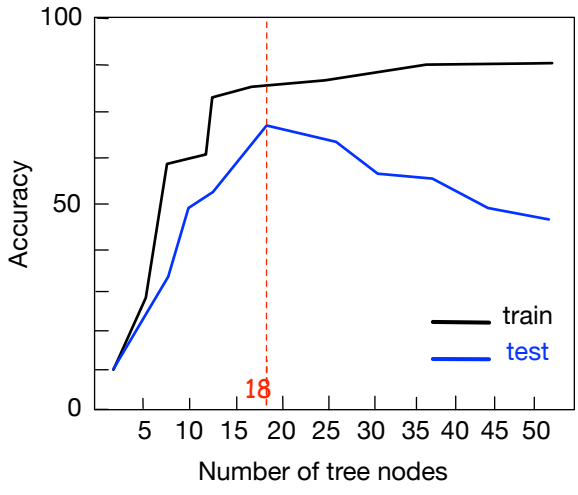


Figure 9-14: Overfitting problem in tree. You can see after the number node increases to more than 20 test data accuracy decreases significantly, but training is still performing good.

tree. On the other hand, shallow trees which have too many branches are prone to error and less accurate inferences.

One common problem while working with trees is the **overfitting** problem. We can continuously go deeper until each leaf node holds one single data point. In other words, for the train dataset, we can observe that the tree splits until all data points can be classified and some leaf nodes include only one single data point. Besides, there are lots of leaf nodes generated.
If we plot the accuracy of test and train datasets we might end up having Figure 9-14, which is a demonstration of overfitting in a decision tree. This figure shows that after the 18th node in the tree وthe test dataset accuracy starts to decrease. However, while working on the train dataset, we went to 50 nodes, but the accuracy didn't decrease. In other words, this problem state that we should cut the tree before it becomes too specific for the train dataset (overfitting), i.e. in this example we should stop at the 18th node.
How to know where to stop? Or speaking more accurately when should we decide to stop? It is clear that we do not have access to the test dataset if we need to use our decision tree in software to decide about the new coming data points in the system.

A popular approach to dealing with this problem is to separate the train dataset into two parts, one is a test and one is a **validation** set as it is shown in Figure 9-15. We construct the tree by using the test data, and not using the validation set.

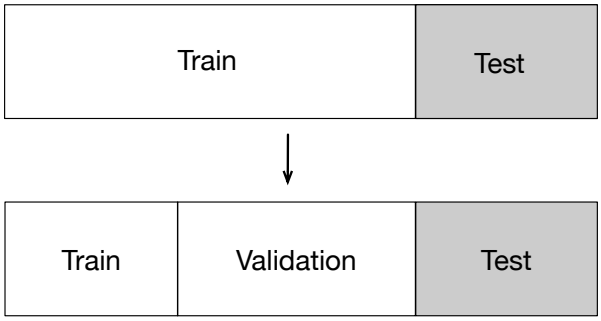


Figure 9-15: Splitting the train dataset into train and validation. Next, algorithm uses the train to construct the decision tree, then it evaluates how accurate is the decision tree on the validation set and if needed the developer can fine tune the decision tree.

Then, we test the tree by validation set. Next, we remove each node and again test the tree (now it is pruned tree) accuracy on the validation set. If the accuracy of the validation set improves, we do not incorporate the recently removed node anymore and 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. This process is something similar to what we have explained about Sequential Backward Selection (SBS) that we have explained back in Chapter 6.

Note 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 take a look at the toy example in Figure 9-16 1. By using the train dataset the algorithm constructs the decision tree, and we experiment with its accuracy on the validation set. The accuracy is 80%. Then, we remove branch B and the accuracy improves to 90%. That is good, so we keep the new tree (Figure 9-16 2) and from this tree, we remove another branch which is branch C, now the accuracy decrease to 70%, which is worse. Therefore, we return to the pruned tree in Figure 9-16 2, because it has the highest accuracy.

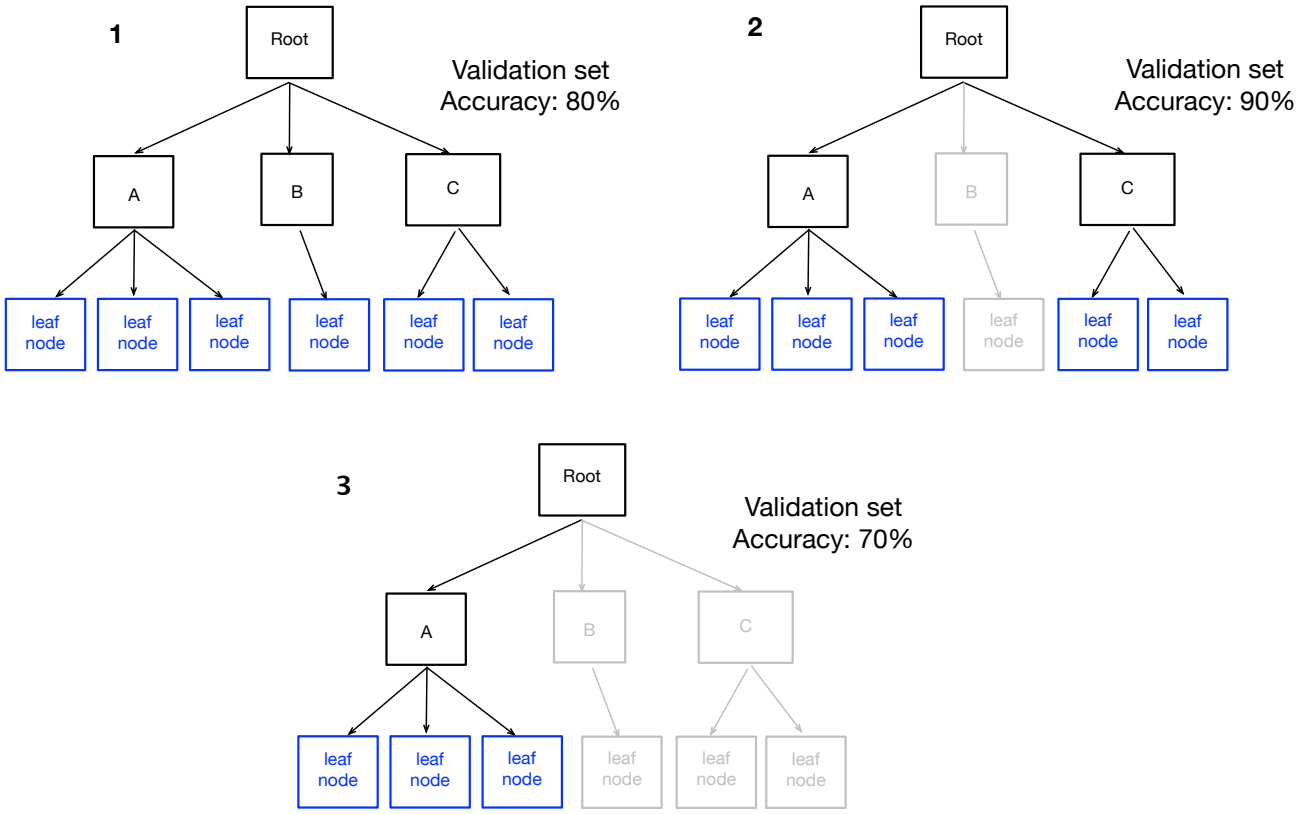


Figure 9-16: An example of pruning the original tree (1) and as a result removing branch B improves the accuracy (2), but removing another branch, branch C decrease the accuracy so the algorithm returns (2) as the final result of pruning.

Another method to prune a tree, which is fairly simpler is to remove a branch and substitute it with the most frequent leaf node. For example, take a look at Figure 9-17. There we have a branch $> X1$ that most of its leaf values are C, so the pruning process removes this branch (and its sub-branches) and substitutes one leaf with a leaf node with value C. In other words, it checks the value of the new record (test set), if it is $> X1$, then it assigns C label to that record and does not prune the tree.

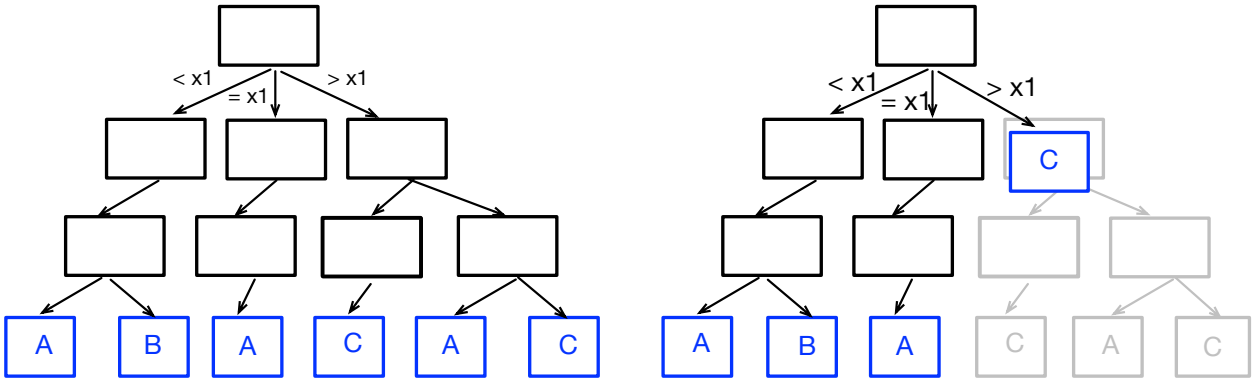


Figure 9-17: (Left) Original tree (Right) the pruned version of the tree, by substituting any branches under > x1 as *C* leaf node. Because most of the leaf nodes under this branch was *C* (two *C* and one *A*) before pruning.

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

Table 9-8: Decision tree algorithms, their splitting method and usage / characteristics.

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 decision trees that are balanced have the computational complexity of $O(log_2(m))$ for testing and $O(n \times log_2(m))$ for training.

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

Table 9-8 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 chose the one that has the highest accuracy and/or resource efficiency.

NOTE:

* 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 heavily invest in removing or reconstructing missing data. Also, note that decision trees can not handle duplicates and it is better to remove duplicates and then fed the dataset into a decision tree for classification.

* We can say that the decision tree is nothing but set of rules that assign the label to the training data point based on that rule. This is a very useful feature of the decision tree, and decision tree algorithms are explainable. At the time of writing this book, 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 which 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 usually tree is loaded into the memory this is very inefficient. This problem can also get mitigated by some condition checks, i.e. rule based classifiers or condition checks (If-Then command).

* ID3, C4.5, and CART algorithms are using greedy search methods. Greedy algorithms perform heuristic search and they might 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 2 is longer than Path 1, 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 get rid of the decision tree, please accept our sincere condolence, we still have a long way to go with decision trees. All classification methods we described until now are very useful. However, in a real-world setting usually, we do not stay with one method, and most of the time (not always) and we use ensemble learning or ensemble methods.

Ensemble learning refers to *the use of more than one classification algorithm to perform the classification task, i.e., decide on labels of the test dataset*. Classification algorithms that we explained until now are called *weak learners* or *vanilla learners* in the context of ensemble learning models. They are not weak, but combining them together to build stronger classifiers, makes them more accurate, and this is the objective of ensemble learning.

Some ensemble learning methods combine several different algorithms together. This can reduce the risks of using each classifier alone, and increase the chance of benefiting from the advantages of different algorithms altogether. It is very similar to human behavior. Assume you ask a person's opinion about a product. What that person said is probably correct, but if you ask the opinion of more people you get a better understanding of that particular product. In a more technical sense, the advantages of using an ensemble learner are lower error rate, and less prone to overfitting.

However, it is not possible to always use ensemble learning methods due to their resource utilization, for example, they are not good for low power or battery powered devices.

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. Since state-of-the-art classifiers are working based on gradient boosting, we postpone explaining these algorithms at the end of this chapter.

Bootstrap Aggregating (Bagging)

This ensemble learning method starts by choosing *more than one random subset* from the training set and builds 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 candidate labels, which are provided by each classifier, the final label will be assigned. Bagging reduces the variance because the combination of different models reduces the variance.

Figure 9-18 visualizes how bagging is working. Here we have a dataset and as the first step, the bagging algorithm creates three subsets from the original dataset. These subsets which hold data points are considered the *bag of data*, and usually, each subset holds **63% of the training dataset**. A data point is usually common between several subsets because the training dataset size is limited and the entire dataset it is not that large to have a distinct subset.

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 one classifier. Afterward, the test dataset uses the majority voting or averaging from models to label the test dataset.

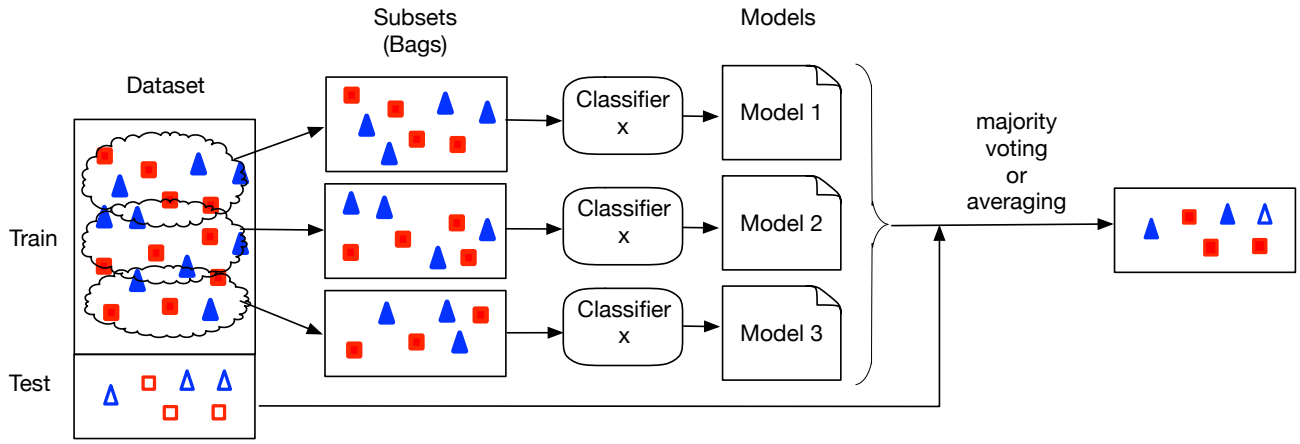


Figure 9-18: A toy example that presents how bagging algorithm works. In the context of ensemble learning a single classifier is called weak learner. Note that all classifiers should be the same algorithm.

Despite the usefulness of vanilla decisions trees, they suffer from high variance. This means that if we separate the train dataset into two parts and train a decision tree on them, the two parts could end up having two very different decision trees. This shows the vulnerability of the decision tree for variance.

We can use a Bagging method (e.g. Random Forest) to reduce the 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 English language bootstrapping[7] is an old expression for some one who is using the shoe laces to climb a fence (which means doing something impossible). In 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 assign an accuracy score to sampled data.

Boosting

Boosting is another ensemble learning method in that each weak learner 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 to buy a product we consult with several other individuals. 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 first starts by applying a classification algorithm to the entire train dataset and makes a model. Next, it collects the errors of the previous model and

[7] https://listserv.linguistlist.org/pipermail/ads-l/2005-August/052756.html

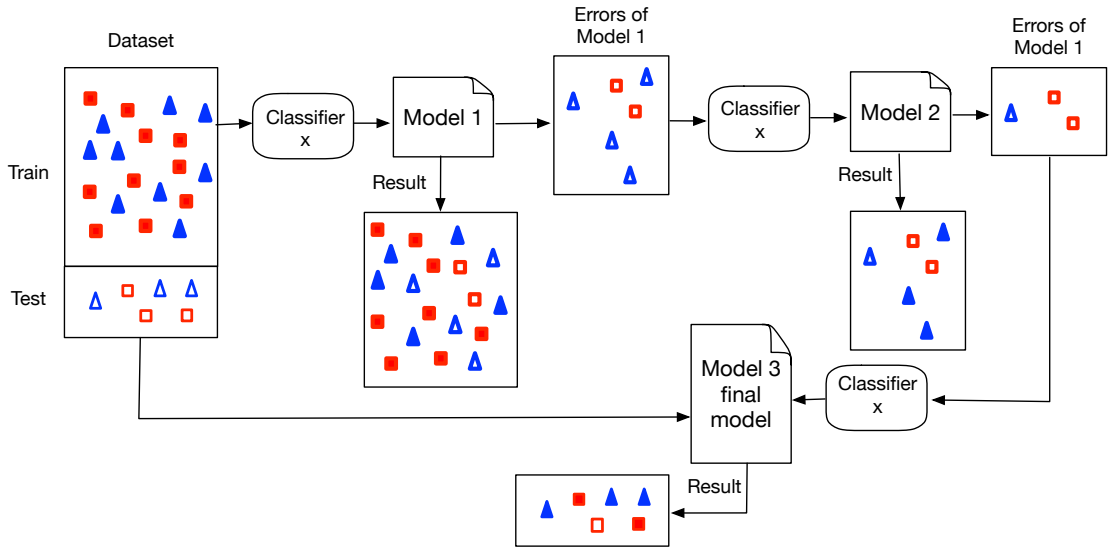


Figure 9-19: A toy example that presents how boosting algorithm works. Note that the test dataset uses the final model which is the strongest model.

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 model of Boosting depends on the previous model. In other words, each new model focuses on the errors of the previous round. The models which perform better will receive higher weights and the model which performs weaker receives lower weight.

Figure 9-19 shows an abstract process flow of 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 trees to construct, (ii) the shrinkage parameter (which 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, hyper-parameter tuning becomes important to avoid overfitting.

Stacking

Stacking applies *different* weak learners algorithms in *parallel* and combines their results (base models) by training a **meta-model**. Meta-model or meta-learner aims to minimize bias 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), their output 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 get extracted from the underlying dataset. Both Bagging and Boosting are usually

used for decision trees, but stacking uses a variety of weak learners and this 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 is called level 0 and it includes 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.

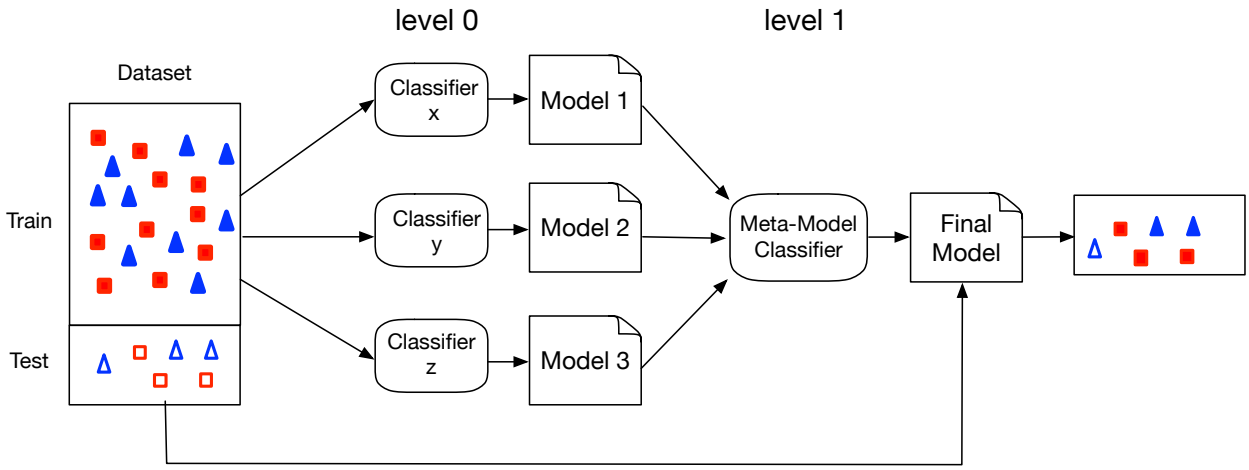
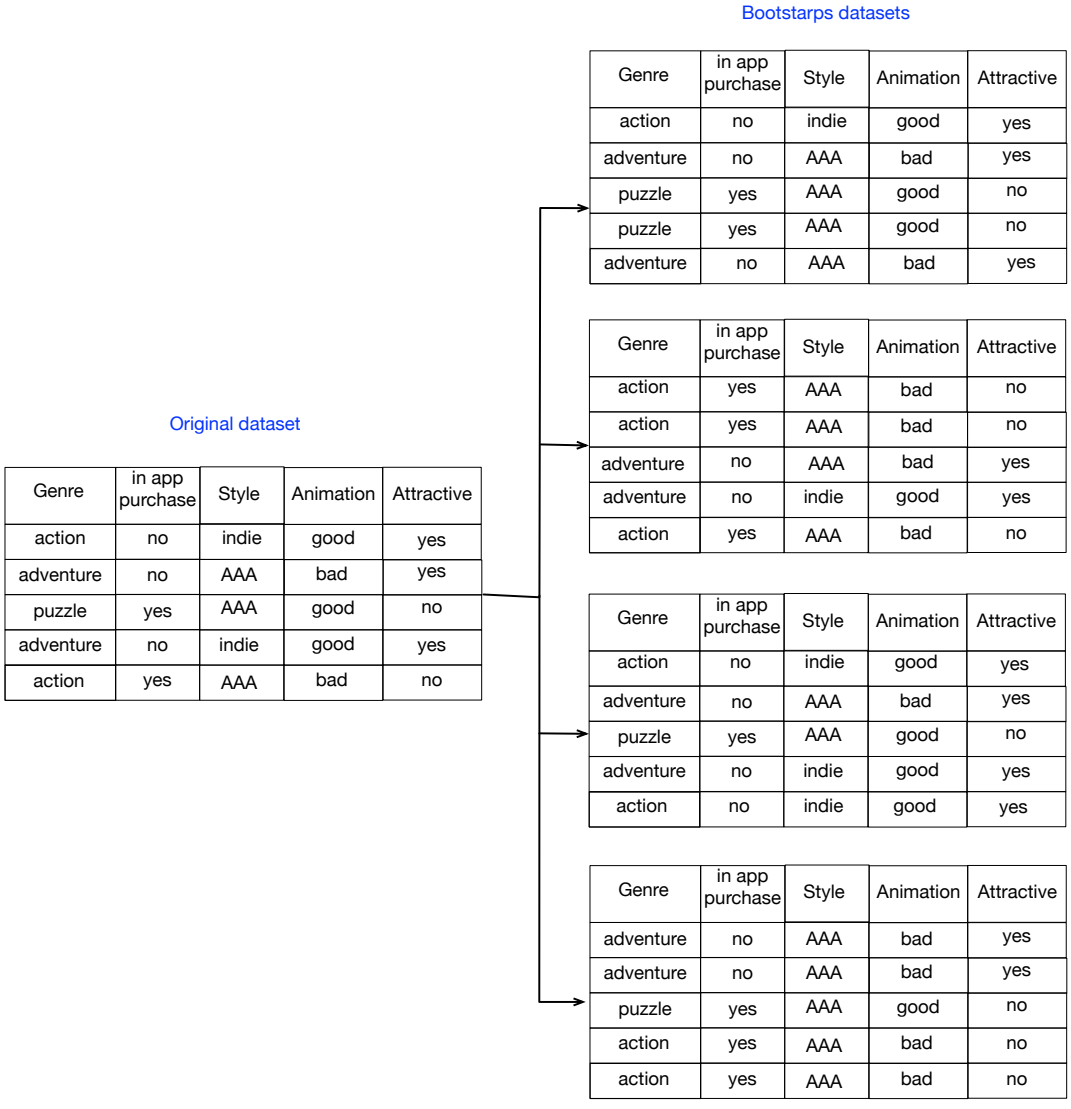


Figure 9-20: A toy example that presents how stacking algorithm works. The first classifiers are called level 0, the second classifier is called level 1.

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 are suffering from weak accuracy. The random forest algorithm resolves the accuracy problem of decision trees with a subtle solution, which we explain soon. Since the random forest algorithm uses decision trees, similar to decision trees, it has a low variance.

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 data science concepts has been resolved. Now, you have started to play some mobile games. A recommendation algorithm intends to suggest you more mobile games. 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.



Original dataset

Genre	in app purchase	Style	Animation	Attractive
action	no	indie	good	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
adventure	no	indie	good	yes
action	yes	AAA	bad	no

Bootstarps datasets

Genre	in app purchase	Style	Animation	Attractive
action	no	indie	good	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
puzzle	yes	AAA	good	no
adventure	no	AAA	bad	yes

Genre	in app purchase	Style	Animation	Attractive
action	yes	AAA	bad	no
action	yes	AAA	bad	no
adventure	no	AAA	bad	yes
adventure	no	indie	good	yes
action	yes	AAA	bad	no

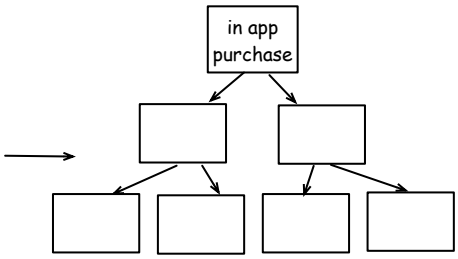
Genre	in app purchase	Style	Animation	Attractive
action	no	indie	good	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
adventure	no	indie	good	yes
action	no	indie	good	yes

Genre	in app purchase	Style	Animation	Attractive
adventure	no	AAA	bad	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
action	yes	AAA	bad	no
action	yes	AAA	bad	no

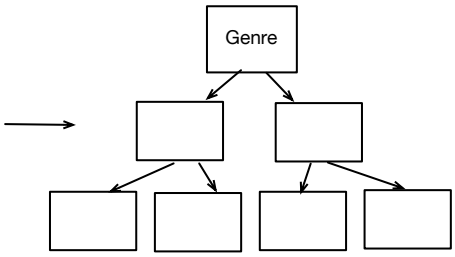
Figure 9-21: From original dataset a set of bootstrap datasets (subsets) are created. The size of the subsets are similar to original dataset, but each of them contain about 70% of the original data. The rest of their data points are duplicates.

(i) As the first step the random forest algorithm starts by creating subsets from the original dataset, which 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 also called bootstrap datasets and the algorithm selects some records randomly to construct them. Since, only 63% of the original dataset only existed in each bootstrap (subset) dataset, the part of the dataset that does not exist in the bootstrap dataset is

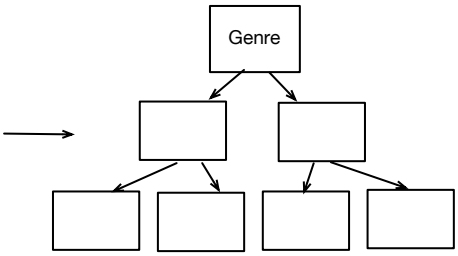
Genre	in app purchase	Style	Animation	Attractive
action	no	indie	good	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
puzzle	yes	AAA	good	no
adventure	no	AAA	bad	yes



Genre	in app purchase	Style	Animation	Attractive
action	yes	AAA	bad	no
action	yes	AAA	bad	no
adventure	no	AAA	bad	yes
adventure	no	indie	good	yes
action	yes	AAA	bad	no



Genre	in app purchase	Style	Animation	Attractive
action	no	indie	good	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
adventure	no	indie	good	yes
action	no	indie	good	yes



Genre	in app purchase	Style	Animation	Attractive
adventure	no	AAA	bad	yes
adventure	no	AAA	bad	yes
puzzle	yes	AAA	good	no
action	yes	AAA	bad	no
action	yes	AAA	bad	no

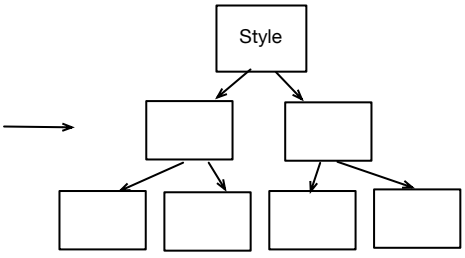


Figure 9-22: k number of features from each select is selected (marked in blue dots) and the split for each decision tree is selected based on analyzing these k features, unlike decision trees which use all features.

called **Out-Of-Bag (OOB) dataset**, i.e. $100 - 63 = 37\,\%$.

(ii) In the second step, assuming we have 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 considers $k = \sqrt{d}$. For example, in Figure 9-22 $d = 5$ so 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.

In other words, unlike vanilla decision trees, we only use a random number of features (columns) and not all of them. This is in contrast to the decision trees that we have explained because there we choose the best feature to split from *all features and not selected features*. Here, however, the split is chosen from a *smaller set of features and not all of them*.

Now a question arises, why did we select k features from the d number of features? If one single feature is very much contributed to a prediction, this feature will be used in the top split of all decision trees. By changing the split features of trees, Random forest creates trees that are highly **uncorrelated**.

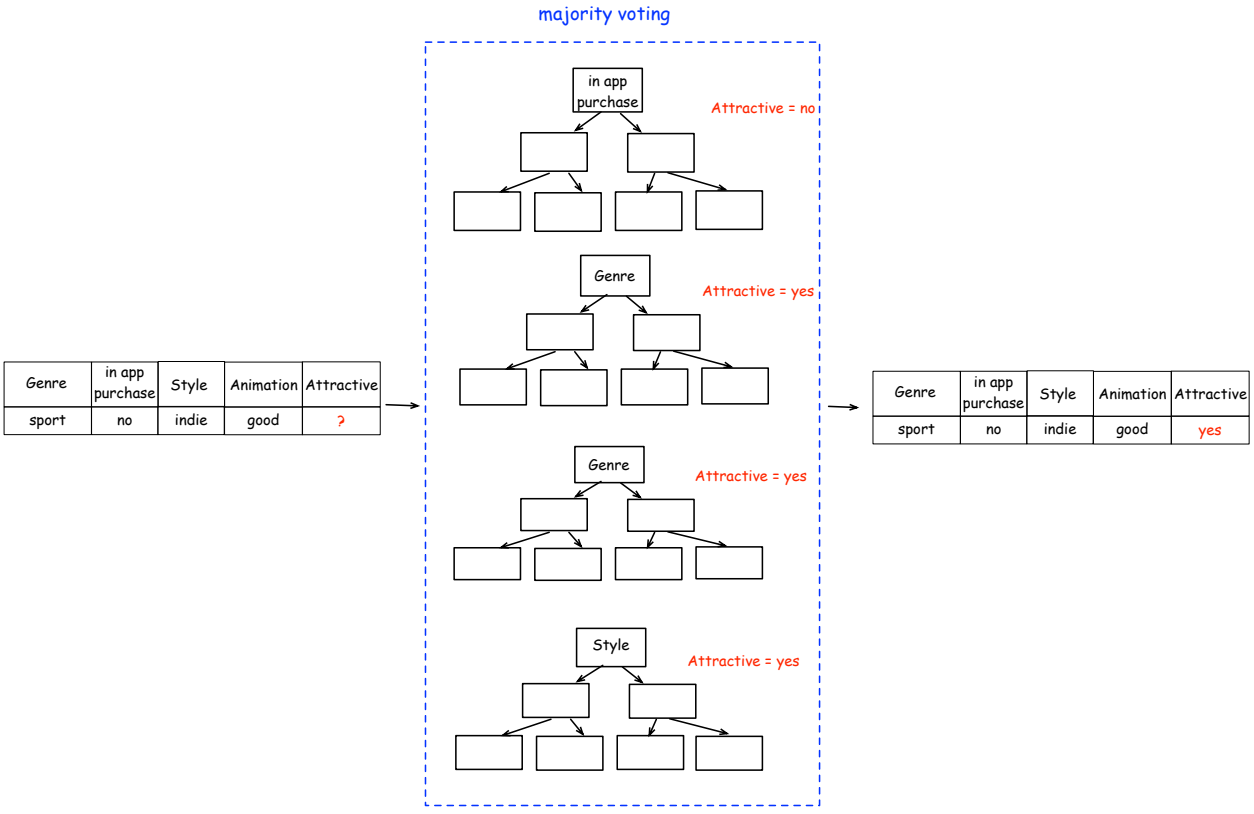


Figure 9-23: A test dataset which has only one record is fed into all decision trees. Each of them is assigned a label for the "Attractive" feature and at the end based on majority voting the final label will be assigned.

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

Now at the end of the training phase we have a set of uncorrelated decision trees. For the test dataset, we easily feed them into the set of decision trees and each of them assigns a label. The algorithms perform a majority voting and decide about the final label based on votes, as it has been 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 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 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 dataset is composed of $[a, b, d]$, we can say $[c]$ is 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 error could be also used for the validation of the dataset, in addition to the test dataset that will be used. This means that we are not relying on just of test dataset for validation, 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 the dataset we have has a small size.

Assuming m is the tree depth and n is the number of data points, we explained that the computational complexity of tree is $O(n \times log_2(m))$. Random forest creates m' number of decision trees, so the computational complexity of training will be $O(m' \times n \times log_2(m))$ 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 also uses decision trees as its weak learners. To learn Adaboost, first, we should learn some concepts, as follows.

- 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 (only one level deep) they are called **stumps**.
- Each data point (e.g. a record in table format) is associated with **weight** and this weight at the beginning is equal for each data point. The sum of weight at each iteration/level should be equal to 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 being calculated as follows: $\frac{\#errors - n}{n}$. For example, if from 5 data points 3 are miss-classified, then the error rate is calculated as: $error\ rate = \frac{3-5}{5}$.
- The result of an Adaboost algorithm is a set of classifiers (stumps) each associated with a weight.

Now, that we learned these preliminary concepts we could delve into the description of the algorithm. Dancing is a part of the culture in many places, especially in Africa, Asia (North, South, and West), Latin America, etc. People gather together in ceremonies, and parties and dance together. After you finish this book, you are in charge of an AI corporation that is building dancing robots for parties. Your robots will learn to dance by looking at people who are dancing at parties.

Standing and Clapping Hand (C1)	Move too many body parts (C2)	Shake the bum (C3)	Actual	Good Dancer (Prediction)
No	No	No	No	No
No	Yes	Yes	No	No
No	Yes	Yes	Yes	Yes
No	No	Yes	Yes	Yes
No	Yes	No	No	No
Yes	Yes	Yes	No	Yes

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

By looking at the people who are dancing at a party the robot constructs the dataset presented in Table 9-9 along with our labels (Good Dancer). The output or prediction variable that we are looking for its label is “Good Dancer”. For 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 $\frac{1}{n}$ of total *n* records (See Table of Figure 9-24). Then it uses each feature alone and creates a decision tree with a depth equal to one, for each feature. In our example, we have three columns as features and thus we have three stumps. Therefore, *the number of stumps is equal to the number of features*. For example, if we have *m* features then Adaboost creates *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 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, which is called **stump performance score**, **significance,** or **stage value** of the stump by using the following equation.

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

In this equation ϵ_t is the total error, which is the sum of weights for the misclassified records in the stump, $\epsilon_t = \sum (misclassied \quad records \quad in \quad that \quad stump)$. Based on Figure 9-24, in

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

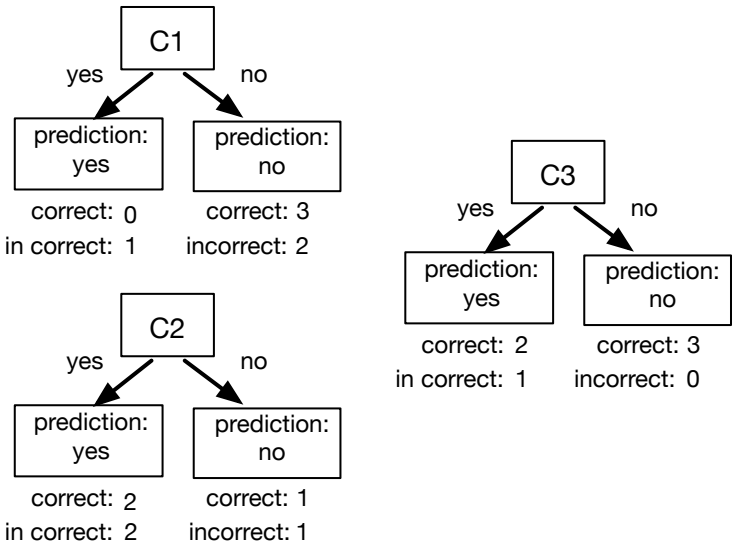


Figure 9-24: First each record receives an equal weight. For each feature a stump is created, the number of its correct vs incorrect instances were written at the bottom of each stump.

stump C3, we have one incorrect record, and thus $\epsilon_t = 1/6$. Now, we can calculate the α_t for stump C3 as follows:

$$\alpha_t = \frac{1}{2} ln(\frac{1-(1/6)}{(1/6)}) = 0.804.$$

(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 $H(x) = \sum_{t=1}^{T} \alpha_t h_t(x)$, Therefore, we have $H(x) = 0.804(Stump(C_3))$ as the result of the first level (*Stump (C3)* is a function).

(v) The next level (level 1) 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 we can see 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 this is the only record that is incorrectly classified by Stump C3 (the last record), which its new weight will be $1/6 \times e^{0.804} = 0.372$. Others were correctly classified, they will be $1/6 \times e^{-0.804} = 0.074$. 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 $(5 \times 0.074 + 1 \times 0.372 = 0.742)$ and dividing each weight by the sum. This helps us to populate the "Level 2 weight (normalized)" column in Table 9-10.

C1	C2	C3	Actual	(Prediction)	Level 1 weight	Level 2 weight	Level 2 weight (normalized)
No	No	No	No	No	1/6	0.074	0.09
No	Yes	Yes	No	No	1/6	0.074	0.09
No	Yes	No	Yes	Yes	1/6	0.074	0.09
No	No	Yes	Yes	Yes	1/6	0.074	0.09
No	Yes	No	No	No	1/6	0.074	0.09
Yes	Yes	Yes	No	Yes	1/6	0.372	0.501

Table 9-10: 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 one records have the largest weight, i.e., 0.501, about half of the records will be similar to these records and the other half will be based on the other weights. Therefore, we will have something like the table in Figure 9-25. The blue records in this table are the one that is sampled from the one that has a weight of 0.501, and others (black ones) are sampled from the 0.074 weights.
(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 same 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 and $\alpha_t = 0.622$, the final prediction model will be written as follows:

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

By looking at $H(x)$ we can see that *Adaboost is a combination of several stumps (weak learners), and each stump is built upon errors of the previous stump.*

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

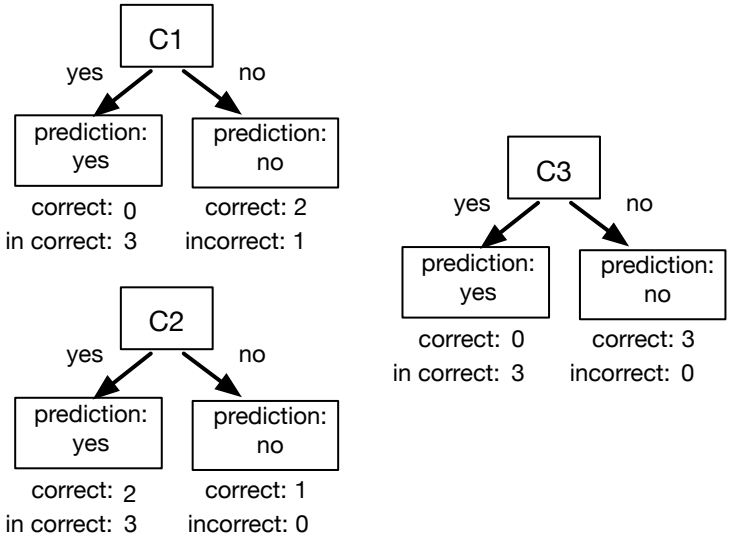


Figure 9-25: A new dataset created based on the normalized level 2 weights. Blue ones are because of weight 0.501, and because of that the algorithm creates multiple copies of that record. Then, based on the new dataset, again for each feature one stump will be created.

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 given number of iterations are 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", which is a predefined variable (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.

NOTE:

* When we are dealing with a small dataset Bagging is the best option. Also, Bagging is robust to noise and outlier, but Boosting is very flexible while having noise in the dataset.

* Decision trees are the most common algorithm used in Bagging. This is due to the fact that they have high variance and Bagging is useful to reduce 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 for estimating the population's statistical characteristics (e.g. distribution) by sampling a dataset with replacement.

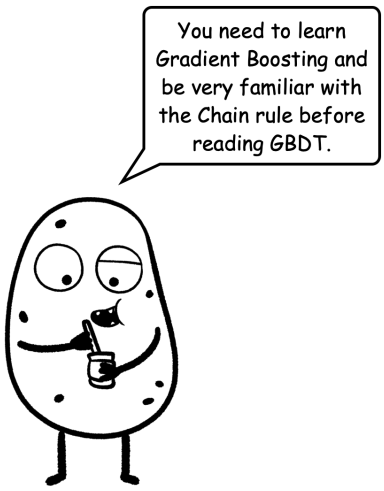
* Random forest enables us to measure the importance of each feature based on its contribution to tree split (e.g. impurity), which makes it an attractive algorithm to use for feature selection as well. In particular, per feature, we can calculate the impurity of the tree and then rank features based 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*.

* Usually, Random forest is better at handling overfitting than Adaboost, but Adaboost provides a higher accuracy

Gradient Boosting Decision Tree

At the time of writing this book, there is an online community, i.e., Kaggle[8], that launches many online machine-learning competitions. Most of the awards go to the participants who use gradient boosting algorithms. Gradient Boosting algorithms have higher accuracy than traditional classification algorithms and even sometimes perform better than Deep learning algorithms (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 seek an accurate algorithm, then experimenting with gradient boosting algorithms is a wise decision.



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

Gradient Boosting Decision Tree (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 (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 remained unchanged (frozen) in the model. This additive model is using 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 used to *minimize residual errors of 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 classification chapter and explain its regression here.

Regression: To implement the regression with gradient boosting, the algorithm performs the following steps.
(1) First, it starts with averaging the data that we intend to predict. Next, the **residual** will be calculated by the loss function. A simplified version of residual (pseudo residual) error can be written as $actual - predicted$

---

[8] http://kaggle.com

(2) Then a new decision tree (e.g. CART) is constructed based on differences between pseudo residuals and predicted value.
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 and 2 repeat, until adding more new trees does not reduce the pseudo residual values or a specific number of trees reaches (the maximum number of trees will be given as a hyperparameter). Then the additive model is constructed by summing up all constructed decision trees together.

It is better to understand its details with an example. Assume we have access to the historical data of "SocialLove", which is a corporation that is a pioneer in Internet social media. The socialLove[9] claims that they respect diversity and treat all their employees fairly.

A friend of ours who is a good data scientist is willing to join this company, and we got access to some internal data, and we could use it to predict the salary in that corporation, based on his/her race and gender. Our algorithm uses gender and race of historical data to recommend a salary of a new candidate. The historical data that is used for our prediction is presented in Table 9-12.

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

Table 9-12: Salary of the SocialLove company based on race and gender for employees with equal qualification.

In the first step the algorithm calculates the average of the target feature used for prediction (i.e. average salary) and then it calculates the residual error for this feature as well. 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, it 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 of its leaf nodes (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 ensure handling the overfitting, the algorithm adds to a learning rate ($\alpha = 0.1$) multiplied by residuals to the predicted value. It uses the following equation.

$$\hat{y}_{new} = \hat{y}_{old} + \alpha \,.\, Residual$$

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

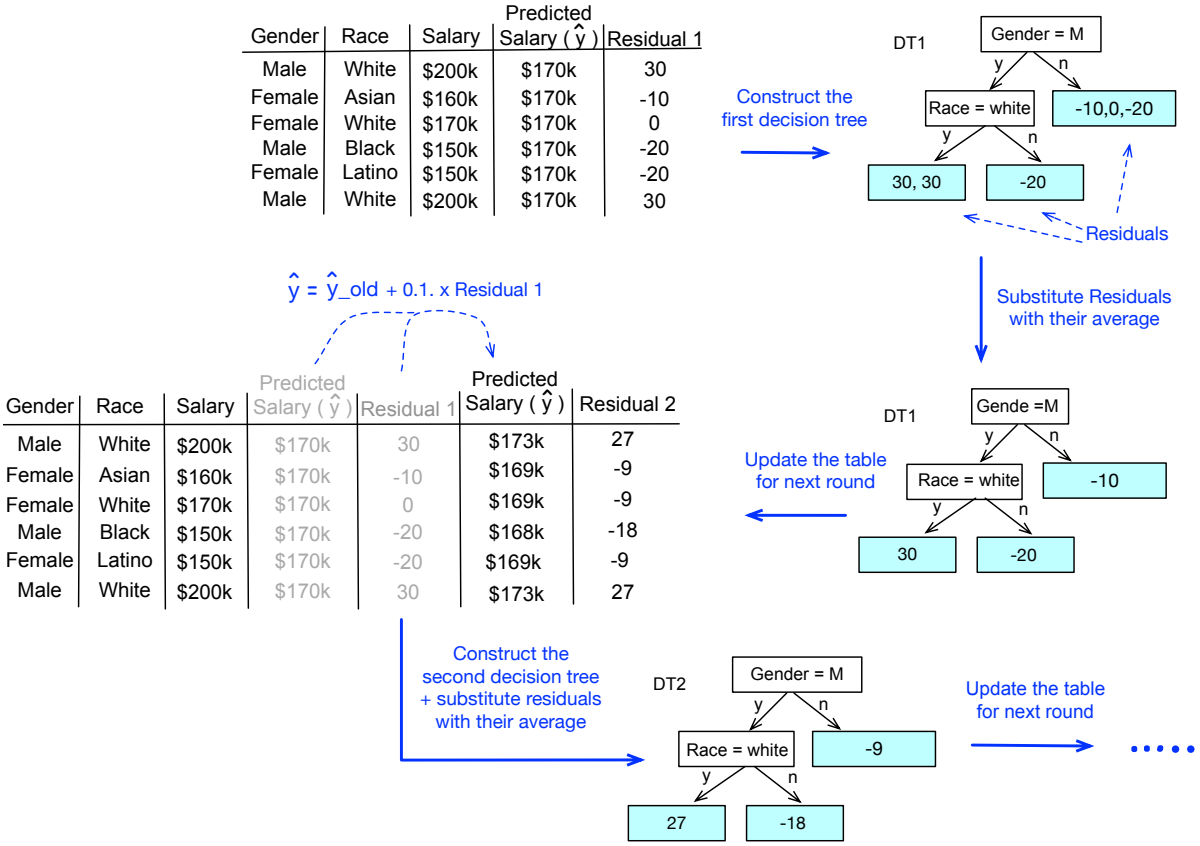


Figure 9-26: Flow of the Gradient Boosting algorithm. The same information from Table 9-12, along with the predicted salary has been added and residuals are calcualted. Since it is the first round the predicted salary is just an average of all salaries. After residuals have been calculated they will 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 maximum number of trees (i.e. hyperparameter)

Multiplication by a learning rate reduces the contribution of the decision tree in the final model and thus more decision trees will be required for 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 residuals are slightly decreased, 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. Therefore, the prediction model will be written as follows:

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

Speaking more technically, we can formalize the prediction model as follows.

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

Here, $H_i(x)$ presents the tree 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 reaches or the residual values do not change. Then, $H(x)$ is the final model and use to label new data.

When a new test data arrives to estimate its salary, the algorithm simply 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 to be too simple, no optimization algorithm has been used and there is nothing about gradient here. However, to describe gradient boosting these simple examples help us to understand, the intuition behind Gradient Boosting.

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

(1) First, the algorithm starts calculating the average of the target prediction variable and then computing its **log(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[10]. 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 calculated ($actual - predicted$) in terms of probability (it will be a number between 0 and 1).
(3) Afterwards, by using a tree construction method (e.g. ID3, CART, etc.), a decision tree (weak learner) will be constructed to predict residuals. The number of residuals is more than the number of leaves in the tree, and some leaves have more than one residual. To assign each leave one residual value, a transformation should be implemented to assign one single value as residual to each leave. The following equation is used for calculating the leaf value from its residuals.

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

P_{old} in this equation 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 its leaf values 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

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 9-13: Previous reaction of the chatbot to the question we asked it.

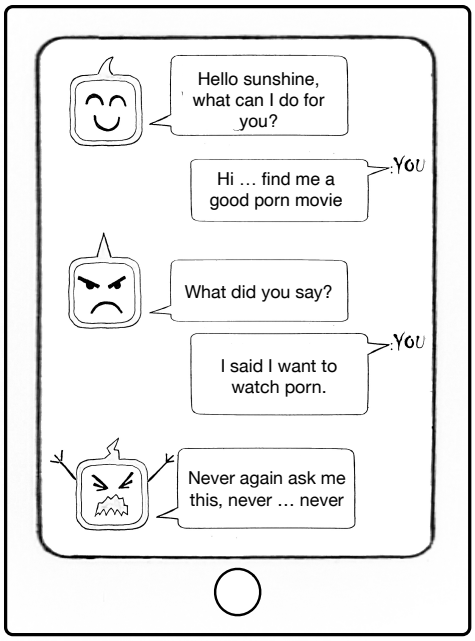
given as a hyperparameter). Then the final model is constructed by summing up all constructed decision trees together.

Table 9-13 presents an example, here we try to create an AI chatbot that unlike existing chatbots (at January 2021), this chatbot is getting angry when you are talking about something useless or harmful. You are in love with your chatbot, but you really hesitate to make your chatbot angry.

[10] If you recall from Chapter 8 we use the same approach for logistic regression and convert log (odds) into probability.

The mood of the chatbot changes based on your questions. Therefore, we try to develop a predictive model to avoid asking questions to the chatbot that results in anger.

We would use the gradient boosting algorithm to predict by the given data, whether the chatbot gets angry or not.



(1) The first step as we have explained before is to make an initial prediction, and then calculate its log(odds). In Table 9-13, column GA, there are four 'yes' and two 'no', in total, we will have six states. Its log(odds) will be calculated as follows:

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

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

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

Since the value of the initial prediction $0.66 > 0.5$, the initial prediction states that all sentences are predicted to make the chatbot angry (we assign 1 to get angry and 0 to not get angry). Definitely, this is not correct, but it is the initial prediction and the algorithm improves its prediction.

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

(2) Now, the probability value of the initial prediction will be 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 9-14.

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 9-14: Previous reaction of the chatbot to the question we asked it.

(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 builds 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 to each leave.
Afterward, we perform the transformation for every leaf (even the one that has only one value)

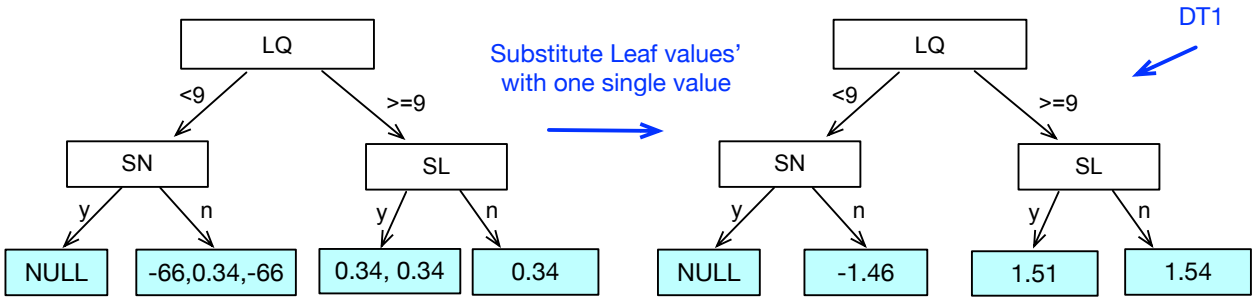


Figure 9-27: A sample decision tree is constructed (by hand to simplifying its readability) from the data in Table 9-14. Leaf nodes assigned residuals as value, but then the described equation of step (2) is used to substitue leaf nodes with more than one variable with new variable.

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.
(4) Now the first decision tree is ready, let's call it DT1 and it could be added to the additive model. In particular, the result mode (additive model) will be the summation of the initial prediction and learning rate (e.g. 0.1) times the decision tree.
$model = 0.69 + 0.1 \times DT1$
(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 9-14, we have LQ<9 and SN=NULL, by substituting them in the model we have $model = 0.69 + 0.1 \times 0 = 0.69$. For the fourth record, we have LQ>=9 and SL=No, by substituting them in the model we have
$model = 0.69 + 0.1 \times 1.59 = 0.11$. These log(odds) values will be used by the Sigmoid function to calculate the new prediction value and construct a table similar to Table 9-14. Obviously, 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 number of trees reach. The final model will be something like the following:
$model = 0.69 + 0.1 \times DT1 + 0.1 \times DT2 + 0.1 \times DT3 + \ldots$

When a new record is added to the table (test data) the algorithm identifies its label (get angry, not get angry) by substituting its values in the model.

Now we understand a simplified version of the algorithm, let's write it pseudo-code which could help us in understanding the algorithm in more technical detail.

In the described examples both regression and classification, we try not to describe its math. Nevertheless, it is useful to learn the Gradient boosting algorithm a bit more formally.
The algorithm gets the input dataset, i.e., $\{(x_i, y_i)\}_{i=1}^{n}$, and a differentiable[11] loss function, i.e., $L(y_i, \gamma)$, as an input. The loss function should be differentiable because it is using 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. L is calculated as $1/2(predicted - actual)^2$ for regression problem.
The gradient boosting algorithm can be written as follows. While reading the following 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 the predict y, given the differentiable loss 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 that we need to find the log(odds) that minimizes the sum. This minimization is implemented with the Newton method or negative gradient (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.

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

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

the algorithm uses the train dataset to construct a weak learner ($H_m(x)$). $H_m(x)$ fits the predicted value to the residual value. In other words, we build a tree to predict pseudo-residuals ($\{(x_i, r_{i,m})\}_{i=1}^{n}$) instead of prediction value ($\{(x_i, y_i)\}_{i=1}^{n}$).

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

each leaf of 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), which is called a "multiplier" (or log(odds) in classification case). γ is the minimum of the cost function. It is an optimization problem that is resolved with a derivative of cost function and chain rule. In simple words, this equation is doing the minimization (we did average in the first iteration) with the optimization function.

---

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

#Note: in the 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.

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

At the end the model will be updated by using the log-odds value and making a new prediction for each sample by adding it to the previous prediction. α is the learning rate.

$$H_m(x) = H_{m-1}(x) + \alpha \times \gamma_m$$

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 density of residual grows after each iteration and it will be a matrix of size $N \times L$.

NOTE:

* In the described example, we have explained 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 example, 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 use trees that have 8 to 32 leaves.

* The pseudo-code for the algorithm for both regression and classification is the same, except a few differences while calculating γ_m.

* In the classification case, to calculate γ_m instead of derivative the algorithm can take second-order Taylor polynomial and approximate the minimum value.

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.

Traditional Decision Tree Gradient Boosting Decision Trees

eXtreme Gradient Boost (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 demand of the users. XGBoost is a derivation of gradient boosting 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 "weighted quantile approach" to identify the threshold value of split faster. In addition, the algorithm is handling 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 details of optimization methods to preserve your energy for understanding the core of this fantastic, but heavy, algorithm.

Unlike the GBDT method, XGBoost uses a specific type of decision tree, i.e., XGBoost tree. This algorithm has many hyperparameters, we describe the important ones here, i.e. η, γ, "minimum child weight", "maximum depth", "subsample", λ and α. η (default = 0.3) is acting as 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 the algorithm randomly samples 50% of training data before growing a tree, at every iteration. λ is a L_2 regularization parameter and α is a L_1 regularizer on weights (leaf values). There are several other hyperparameters, which we do not explain and you can check them from XGBoost documentation.

XGBoost can be used for both classification and regression. The loss function for classification and regression is different, but both cases use second-order Taylor approximation (Check Chapter 8 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 GDBT with a specific regularizer and the regularizer in XGBoost for tree k 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 the algorithm calculates all initial residuals similar to the GBDT algorithm. Then, it constructs a single leaf tree that holds all residuals.

(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 probability, it calculates a *similarity weight*. If XGBoost is used for classification it uses the square summation of all residuals divided by the summation of previous probabilities (P_{old-i}):

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

If XGBoost is used 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)

Tree with the best Gain will remain and other trees will be removed.

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

(5) The algorithm calculates the "output prediction 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 initialy 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 similar to the gradient boosting algorithm. In particular, for new test data, the algorithm calculates the log(odds) of the previous prediction probability 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, let's learn it with an example.

We have the data of a few readers who have read this chapter and experimented with described algorithms by writing some codes from scratch. We would like to predict whether a new reader who read this chapter couple of times, with some implementations of these algorithms, is already learned described algorithms or not? Table 9-15 shows the dataset we have and we will this dataset for training. For testing, we give a new record, with specified RC and EC and the algorithm predicts its 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
1	No	No
2	Yes	Yes
1	No	No

Table 9-15: Sample dataset used by XGBoost.

In the example we describe here, for sake of simplicity 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 sake of simplicity we use binary classification). Similar to the GDBT, the initial prediction will be the average and it is $(0+1)/2 = 0.5$. Initial pseudo-residuals are calculated as the differences between observed and predicted values (*Residual = Observed - Predicted*). The predicted value, in the beginning, is equal to the initial prediction, which is also called *the base model* as well, because it participates in constructing the final model. By substituting initial residuals in Table 9-15 we get the table presented on the left side of Figure 9-28 (just look at the table there and not the right side).

After residuals have been calculated the XGBoost algorithm fits a binary tree for each feature (column) to the residuals. However, it is not yet 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 it is 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 branches, but for 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 will be 1.5, and rounded to 1.

RC	EC	LA	Residuals
2	Yes	1	1
1	No	0	0
2	Yes	0	0
1	No	0	0
2	Yes	1	1
1	No	0	0

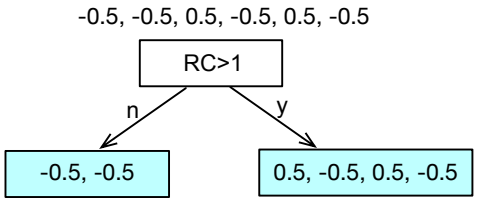


Figure 9-28: Table 9-15 is used to calculate the residuals, we intend to do a classification and since it is binary we substitute yes and no with 1 and 0 in LA column. On the right side a binary tree is constructed and we assume the RC>1 as a split.

(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:

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

And respectively the similarity weight for the leaf on the right side is calculated as:

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

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.

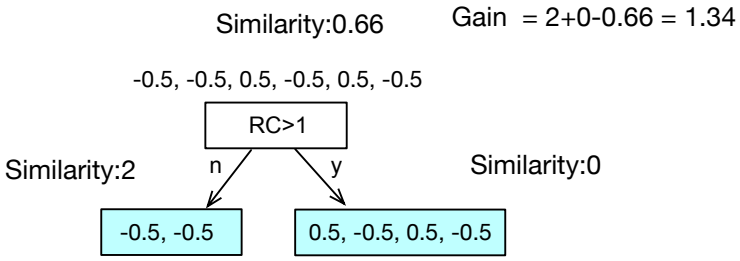


Figure 9-29: Similarity score of each node will be used to calculate the Gain score for RC.

(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 3.33 and it is higher than RC gain, which is 2. 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 experiment with different possible combinations for binary tree construction (XGBoost split trees are always binary trees) and at it selects the tree

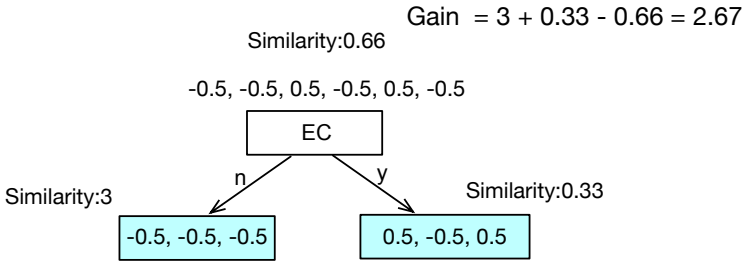


Figure 9-30: Similarity score of each node will be used to calculate the Gain score for EC.

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, next it calculates their gain. Then another tree is constructed that has G as one branch, the other branch includes R and

B, and calculates their gain. Afterward, it chooses the best tree with the highest gain. However, experimenting with too many trees is infeasible and it uses approximation, which we do not explain in more detail.

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

To see how does gain of a branch is computed, check the example presented in Figure 9-31.

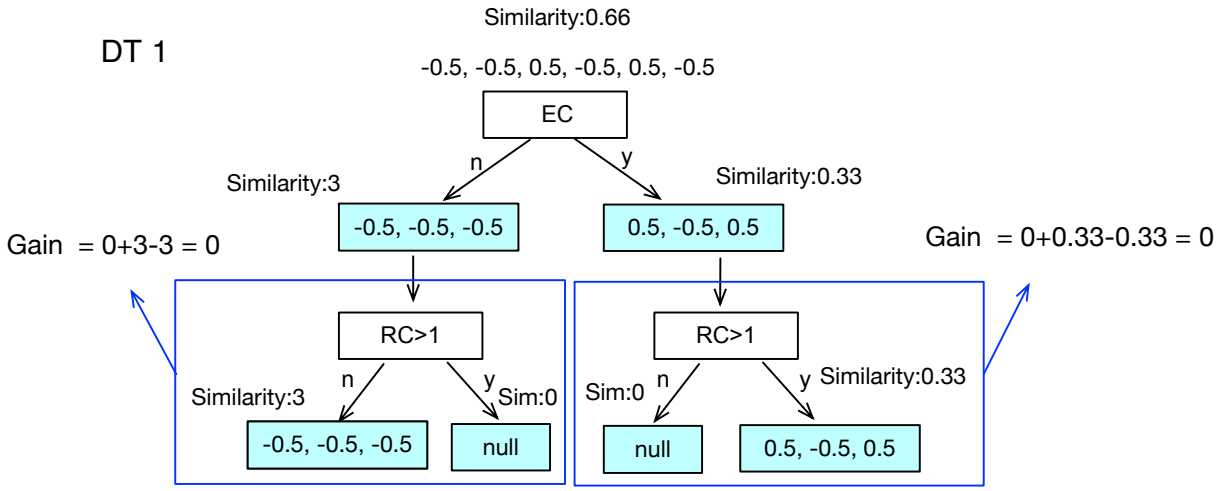


Figure 9-31: A split, and gain scores have been calcualted for other branches.

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) and recall we set λ to zero.

The followings present how we have calculated the output for each branch:

$EC = n, RC = n$:

$$output : \frac{-0.5 + -0.5 + -0.5}{3} + 0 = -0.5$$

$EC = n, RC = y$:

$output : 0$

$EC = y, RC = n$:

$output : 0$

$EC = y, RC = n$:

$$output : \frac{0.5 + -0.5 + 0.5}{3} + 0 = 0.16$$

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

(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. The new log(odds) of predicted probability will be $0 + 0.3 \times log(odds)\ of\ DT1_{output}$ (0.3 is the

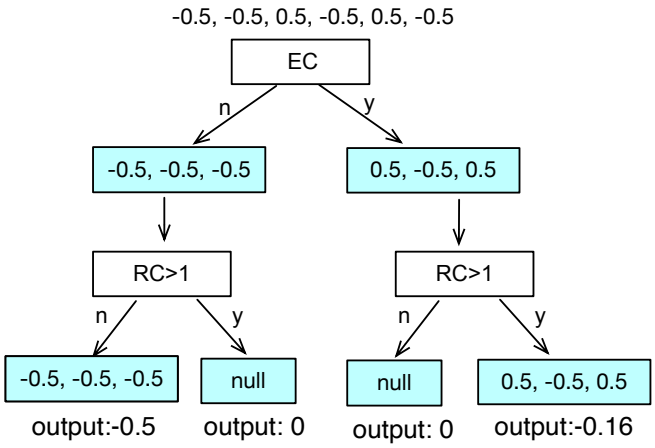


Figure 9-32: Output for each leaf has been calculated and added to the buttom of leaves.

recommended learning rate η). For each record, we can substitute the values into the above 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 9-16). 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	0.48	0.52
1	No	0	-0.5	0.3 x -0.5	0.18	-0.18
2	Yes	0	-0.5	0.3 x 0.16	0.48	-0.52
1	No	0	-0.5	0	0.5	-0.5
2	Yes	1	0.5	0.3 x 0.16	0.48	0.52
1	No	0	-0.5	0	0.5	-0.5

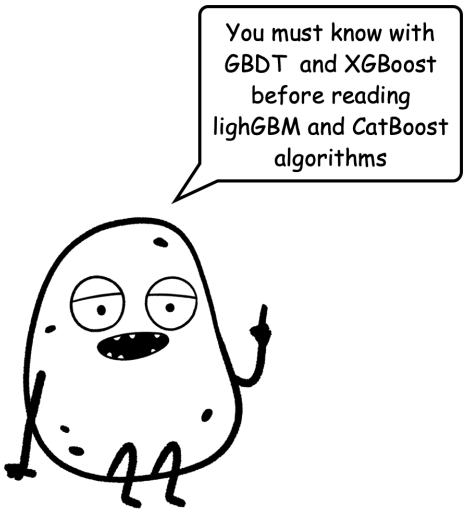
Table 9-16: New prediction probabilities and their residuals which are constructed based on old residuals. In each iterations residuals will get smaller.

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 keep in mind 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 that have 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.

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 that existed in XGBoost is the need to scan many possible combinations to decide on the best split. LightGBM, which operates as another gradient boosting tree, 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 removes those data points that have small gradients. This process decreases their role 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.

The EFB is using the sparsity that existed in most real-world datasets. In other words, it is very common that 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.

EFB merges exclusive features into a single feature (similar to feature reduction techniques we described back in Chapter 6).

Exclusive features are features that never take non-zero values together. EFB reduces the problem of feature reduction to a graph coloring problem and uses a greedy algorithm[12] to remove features. In particular, all features will be a vertex in the graph, and if they are not mutually exclusive[13] the algorithm adds one edge between them. The weight of edges corresponds to conflicts between features.

To understand the characteristics of EFB take a look at Table 9-17 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

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 2
0	2+0
1	0+1
3	2+0
0	0+3
4	0+0
0	2+0

Feature 1	Feature 2, Feature 3
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 9-17: (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.

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 conflict (e.g. few records have non-zero values but still both features merged) in each bundle, but we did not use it in our example.

While constructing bundles, one problem might arise; the bundled feature should not be in the same range as the original feature and 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.

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

---

[12] Greedy algorithm refers to heuristic algorithm (Chapter 6) in which each step choose a local optimum. At the end it might not end up in the best optimum solution, but each step alone choose the best optimum that is available for the algorithm.

[13] Mutual exclusion means that two process, can not exists at the same state. To remember that assume a rest-room which usually two person can not enter one rest-room.

CatBoost

CatBoost [Prokhorenkova '18] is a third popular GBDT based algorithm that has been 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 happens when *the distribution of the training set changes and moves away from the actual distribution of data*. In simple words, the variable we want to predict moves closer to the data in the current sub-sampled dataset, and further away from the true values in the actual dataset.

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 is using 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 to the current data point. p stays for 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 that is 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 9-18, here we assume p as a constant and equal to 0.5 and α is 1. on Table 'a' we have features and their values. It is a result of binary classification, thus each row has a value of either 0 or 1. In Table 'b' the average of each feature is calculated and in table 'c', these features have been substituted on the original table. This is the plain target encoding that is prone to data leakage.

The authors proposed to use the described equation and resolve the data leakage by ordering. Take a look at Table 9-18 'd', which is the ordered target encoding value. Note that *current_count* and *total_feaute_count* both are calculated by looking only at previous records, not the current one and overall table. For example, we describe the justification of the sixth row in Table 9-18 'd'. As another example, let's take a look at the value of the third row, where Apple=1. The *current_count* of this row is equal to 1, because there is one row before this row,

i.e. first row that has Apple=1, and the *total_feature_count* of the Apple feature is also 1 because

a

Fruit	Value (target)
Apple	1
Orange	0
Apple	1
Orange	1
Orange	1
Banana	1

b

Fruit	Avg. Target Value
Apple	2/2 = 1
Orange	0+1+1/3 = 0.67
Banana	1/1 = 1

c

Fruit	Avg Target Value
Apple	1
Orange	0.67
Apple	1
Orange	0.67
Orange	0.67
Banana	1

d

Fruit	Ordered Target Statistics
Apple	(0+0.5) / (0+1) = 0.05
Orange	(0+0.5) / (0+1) = 0.05
Apple	(1+0.5) / (1+1) = 0.75
Orange	(0+0.5) / (1+1) = 0.25
Orange	(1+0.5) / (2+1) = 0.5
Banana	(0+0.5) / (0+1) = 0.5

What is the current count of having Orange =1?
There is only one record before this record, therefrore: *current_count =1*

There are two rows before this record, second and fourth record.
Despite they have different values the *current_feature_count = 2*

Table 9-18: An example how CatBoost calculate the Ordered Target Encoding.

there is only one row that includes Apple before this row, i.e. first row.

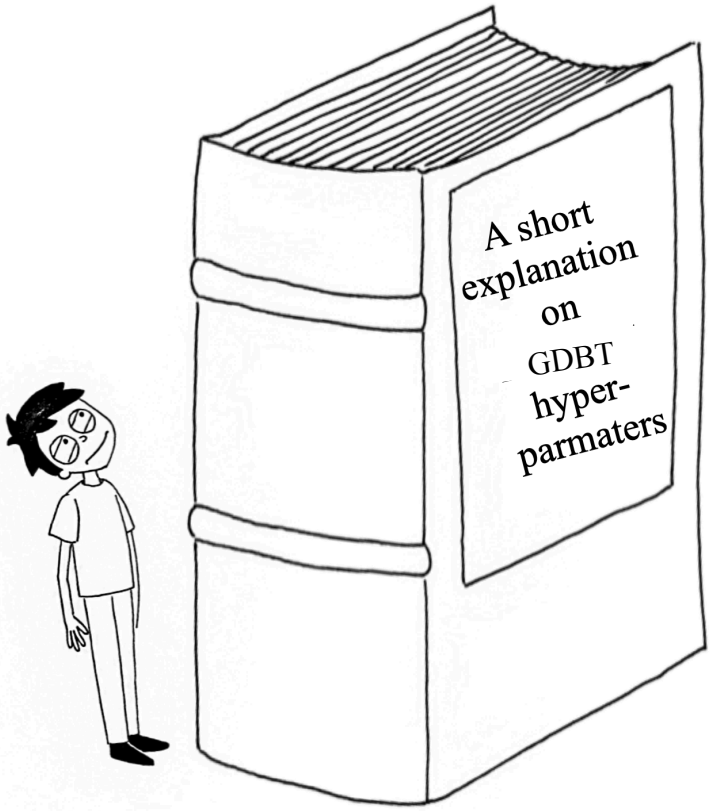
The second novelty of CatBoost is its Order Boosting. Order Boosting does not use the whole dataset to calculate the residuals at every iteration of GBDT.

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, by using a tree that is constructed by using a subset of the dataset, and it calculates the residuals by using the data points that it did not see before (from another subset of the dataset). To understand this issue, let's say we have a 1D dataset of 10 data points, i.e. $\{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\}$. 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*), 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. This means that the model which is trained on $\{x_4, x_6, x_3, x_1, x_8\}$ never see 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 to construct decision trees for data points that are located at 2^j, where $j = 1,2,4,8,...log_2(n)$. It means the first model is trained on the first data point, the second model is trained on the first and second data points,

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 the 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, however, summing them together makes CatBoost a very computationally expensive algorithm. In an experiment, Keshavarz et al. [Keshavarz '20] compared the computational complexity of these Gradient Boosting algorithms and CatBoost is the most resource intensive one.

NOTE:

* Except lightGBM other gradient boosting 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 grow 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 not GPU available and you are using a local machine for training usually LightGBM performs better than XGBoost and CatBoost, which operate well with GPU.

* These three boosting algorithms are the state-of-the-art algorithms that can work accurately on tabular data and when we have small data.

* Despite all Gradient Boostin 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. There are approaches such as SEFR [Keshavarz '21] algorithm that try to mitigate this limitation, but still they are not widely adopted among the community.

How to select the best 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 is relying solely on the accuracy of the model. In particular, comparing the cost (false-negative, false-positive) with benefit (true-positive, true-negative), or using the ROC curve of both models (check Chapter 8 to recall what is ROC curve).
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 results 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.

Review and Remark

This chapter starts by explaining different classification methods including Bayesian network, *k*NN, SVM, and four different decision trees.

We started to explain the Rule-based classifier. The rule base classifier is not really a machine learning algorithm, because we define rules by hand and somehow manually we are building the model. There are some basic classification algorithms that are still widely in use, although they sound simple. These include the legendary kNN algorithm with different implementations (LSH Voronoi tessellation and KD-Tree), SVM with different Kernel functions, and Naive Bayes.

The backbone of the most accurate algorithms is the decision tree, and we explained three decision trees, 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.

Next, we explain the ensemble learning algorithm which uses listed algorithms as a weak learners and combines them to have a better learning result. Table 9-19 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. Random Forest algorithm is the most popular bagging algorithm. First, it creates subsets from the original dataset, then, it selects a number of 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 the decided by averaging the

Ensemble Learning Method	Description	Algorithm Examples
Stacking	It is a combination of weak learners and combine them to build the final model is	Combination of Weak Learners
Bagging	Choosing a random subset from the training set and building 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 which is provided by each classifier.	Random Forest
Boosting	Boosting is a sequential process that first starts by applying a classification algorithm first 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 9-19: Summary of ensemble learning algorithms.

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.

Gradient Boosting Decision Tree (GBDT) algorithm is an ensemble algorithm, that operates by using a gradient cost function. The boosting process of this algorithm is a numerical optimization problem. 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 GDBT 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. There are three algorithms built of top of GBDT, including XGBoost, LightGBM, and CatBoost. At the time of writing this Chapter, these algorithms are state-of-the-art algorithms to be used on tabular data whose number of data points are limited.

Further Reading

* There is a good book for basic algorithms, Mastering Machine Learning Algorithm written by Jason Brownlee [Brownlee ‘16], if you are interested to see more example and learn basic

supervised machine learning algorithm we recommend to check this book. We use this book to construct the Naive Bayesian section of this chapter as well.

* Victor Lavrenko (https://www.youtube.com/user/victorlavrenko/videos) has a very good video series on teaching LSH, we have used his explanation to construct our explanation as well, and also good videos to explain decision tree.

* There are many explanations on Kernel function and even book written about kernels [Scholkopf '01] available. Nevertheless, slides for Matt Gormley from CMU is one of the few ones that we find useful and help us to learn the motivation behind using kernel functions. https://www.cs.cmu.edu/~mgormley/courses/10601-s17/slides/lecture12-svm.pdf. Also Josh Starmer online videos on teaching the mathematics behind kernel function is very clear and helpful, like most of his videos.

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

* Motaz Saad has a very good visualize 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. Their online videos helped us a lot in understanding the details of the Adaboost Algorithm and they are available on YouTube.

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

* If you need more explanation about gradient boosting algorithm with good examples and details, Josh Stramer has a good example for that. You can check his StatQuest videos here: https://statquest.org. We have used his explanations to construct the gradient boosting and XGBoost algorithm of this chapter as well. Most of Josh's videos are very easy to understand and descriptive.

* To learn LightGBM and CATBoost from other resources as well, 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 for the description of 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 following link: https://explained.ai/gradient-boosting/ written by Terence Parr and Jeremy Howard.