Notes on Machine Learning

Sivaram N
Department of IT
Bapatla Engineering College

Contents

Notation

This section provides a concise reference describing notation used throughout this document. If you are unfamiliar with any of the corresponding mathematical concepts, (Goodfellow et al., 2016) describe most of these ideas in chapters 2–4.

x :- A scalar (integer or real) denoting a feature of training example (e.g., the height of a person)

$\boldsymbol{x}$:- A vector denoting a training example with m features (e.g., with $m = 3$ we could represent the height, weight, and age of a person), represented as a column vector (i.e., a matrix with 1 column, $\boldsymbol{x} \in \mathbb{R}^m$),

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{1}$$

also shown as $[x_1 \quad x_2 \quad \cdots \quad x_m]^\top$

$\boldsymbol{X}$:- Design matrix, $\boldsymbol{X} \in R^{n \times m}$, which stores n training examples, where m is the number of features,

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^\top \\ \boldsymbol{x}_2^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{bmatrix}. \tag{2}$$

Note that in order to distinguish the feature index and the training example index, we will use a square-bracket superscript notation to refer to the ith training example and a regular subscript notation to refer to the jth feature:

$$\boldsymbol{X} = \begin{bmatrix} x_1^{[1]} & x_2^{[1]} & \cdots & x_m^{[1]} \\ x_1^{[2]} & x_2^{[2]} & \cdots & x_m^{[2]} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{[n]} & x_2^{[n]} & \cdots & x_m^{[n]} \end{bmatrix}. \tag{3}$$

The corresponding targets are represented in a column vector $\boldsymbol{y}; \boldsymbol{y} \in \mathbb{R}^n$:

$$\boldsymbol{y} = \begin{bmatrix} y^{[1]} \\ y^{[2]} \\ \vdots \\ y^{[n]} \end{bmatrix}. \tag{4}$$

Chapter 1

Introduction

1.1 Key Machine Learning Terminology

1.1.1 What is Machine Learning?

A machine learning algorithm is an algorithm that is able to learn from data. Mitchell (1997) provides a succinct definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

1.1.2 Task T

Machine learning tasks are usually described in terms of how the machine learning system should process an example. An **example** is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $\boldsymbol{x} \in \mathbb{R}^n$ where each entry x_i of the vector is another feature.

In the spam detector example, the features could include the following:

- words in the email text
- sender's address
- time of day the email was sent
- email contains the phrase "donate money."

Examples are of two categories:

- **labeled examples**: A labeled example includes both feature(s) and the label. That is:

$$labeled\,examples : \{features, label\} : (\boldsymbol{x}, y).$$

  In our spam detector example, the labeled examples would be individual emails that users have explicitly marked as "spam" or "not spam."

- unlabeled examples

Some of the most common machine learning tasks include the following:

- **Classification**: In this type of task, the computer program is asked to specify which of k categories some input belongs to. To solve this task, the learning algorithm is usually asked to produce a function $f : \mathbb{R}^n \rightarrow \{1, \ldots, k\}$. When $y = f(x)$, the model assigns an input described by vector $\boldsymbol{x}$ to a category identified by numeric code y. There are other variants of the classification task, for example where f outputs a probability distribution over classes.

Figure 1.1: Hand written digits

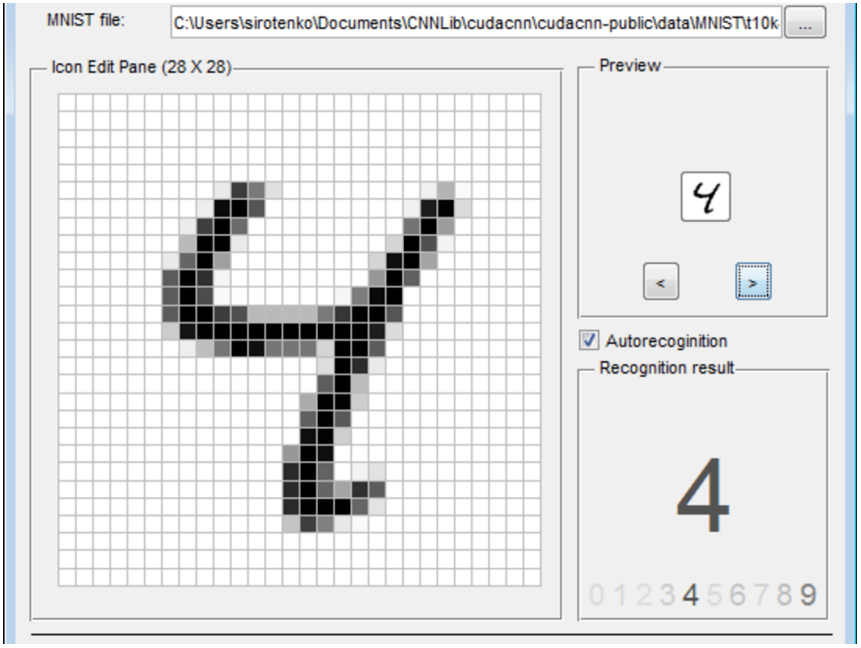


Figure 1.2: Classification of hand written digits

- **Regression**: In this type of task, the computer program is asked to predict a numerical value given some input. To solve this task, the learning algorithm is asked to output a function $f : \mathbb{R}^n \to \mathbb{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums), or the prediction of future prices of securities.

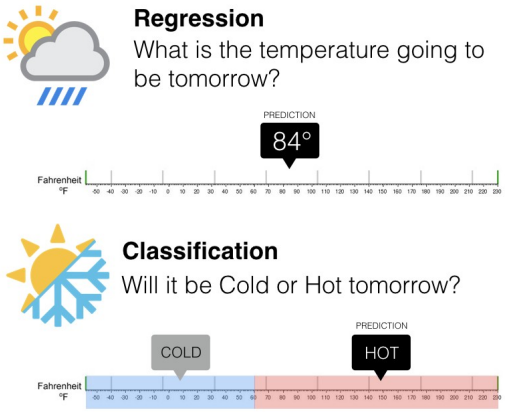


Figure 1.3: Classification vs Regression

- **Transcription**: In this type of task, the machine learning system is asked to observe a relatively unstructured representation of some kind of data and transcribe the information into discrete textual form. For example, in optical character recognition, the computer program is shown a photograph containing an image of text and is asked to return this text in the form of a sequence of characters (e.g., in ASCII or Unicode format).

- **Machine translation**: In a machine translation task, the input already consists of a sequence of symbols in some language, and the computer program must convert this into a sequence of symbols in another language. This is commonly applied to natural languages, such as translating from English to Telugu.

- **Structured output**: Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships between the different elements. This is a broad category and subsumes the transcription and translation tasks described above, as well as many other tasks. One example is parsing—mapping a natural language sentence into a tree that describes its grammatical structure by tagging nodes of the trees as being verbs, nouns, adverbs, and so on.

the chef cooks the soup

↓

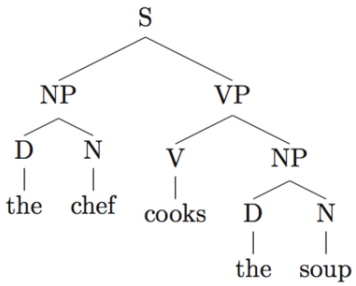


Figure 1.4: structured output

- **Anomaly detection**: In this type of task, the computer program sifts through a set of events or objects and flags some of them as being unusual or atypical. An example of an anomaly detection task is credit card fraud detection. By modeling your purchasing habits, a credit card company can detect misuse of your cards. If a thief steals your credit card or credit card information, the thief's purchases will often come from a different probability distribution over purchase types than your own. The credit card company can prevent fraud by placing a hold on an account as soon as that card has been used for an uncharacteristic purchase.

- **Density estimation or probability mass function estimation**: In the density estimation problem, the machine learning algorithm is asked to learn a function $p_{model} : \mathbb{R}^n \rightarrow \mathbb{R}$, where $p_{model}(\boldsymbol{x})$ can be interpreted as a probability density function (if $\boldsymbol{x}$ is continuous) or a probability mass function (if $\boldsymbol{x}$ is discrete) on the space that the examples were drawn from. To do such a task well (we will specify exactly what that means when we discuss performance measures P), the algorithm needs to learn the structure of the data it has seen. It must know where examples cluster tightly and where they are unlikely to occur.

1.1.3 The Performance Measure, P

To evaluate the abilities of a machine learning algorithm, we must design a quantitative measure of its performance. Usually this performance measure P is specific to the task T being carried out by the system. For tasks such as classification and transcription, we often measure the **accuracy** of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the **error rate**, the proportion of examples for which the model produces an incorrect output. We often refer to

the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if it is not.

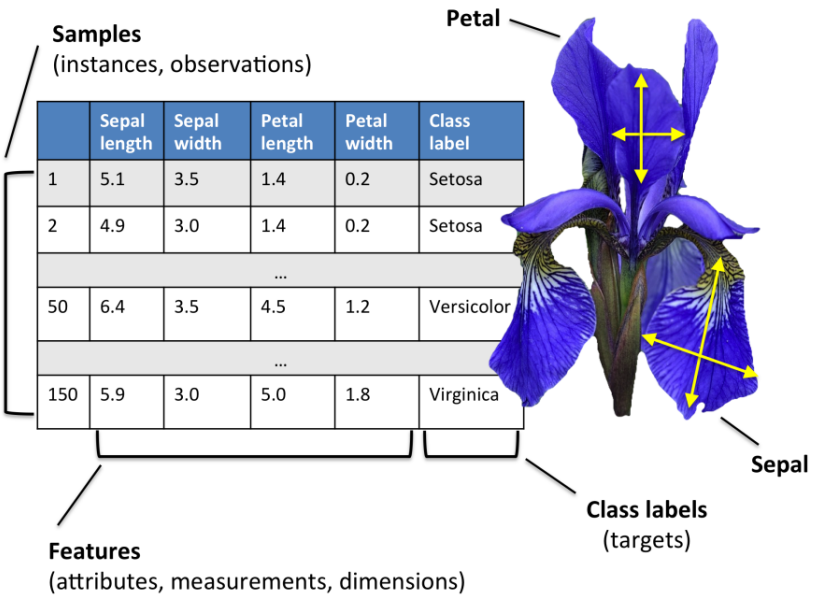
Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a **test set** of data that is separate from the data used for training the machine learning system.

1.1.4 The Experience, E

Machine learning algorithms can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process. Most of the learning algorithms are allowed to experience an entire **dataset**.

A dataset is a collection of examples, which are in turn collections of features. One common way of describing a dataset is with a **design matrix**. A design matrix is a matrix containing a different example in each row. Each column of the matrix corresponds to a different feature. For instance, the Iris dataset contains 150 examples with four features for each example. This means we can represent the dataset with a design matrix $\boldsymbol{X} \in \mathbb{R}^{150\times 4}$, where $X_{i,1}$ is the sepal length of plant i, $X_{i,2}$ is the sepal width of plant i, etc. Often when working with a dataset containing a design matrix of feature observations $\boldsymbol{X}$, we also provide a vector of labels $\boldsymbol{y}$, with $\boldsymbol{y}_i$ providing the label for example i. Of course, sometimes the label may be more than just a single number. For example, if we want to train a speech recognition system to transcribe entire sentences, then the label for each example sentence is a sequence of words.

Figure 1.5: 3 Classes / Categories of Iris flower



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
			...		
50	6.4	3.5	4.5	1.2	Versicolor
			...		
150	5.9	3.0	5.0	1.8	Virginica

Figure 1.6: Design matrix of Iris dataset

1.2 Components of Learning

Let us take credit approval application to understand the essence of machine learning. A bank receives thousands of applications for credit every day, and it has to automate the process of evaluating them.

Informal definition of the problem:

- Bank collects information like salary, debt, years in residence, etc., from the applicant and decides whether to give approval for credit or not.

feature	value
age	32 years
gender	male
salary	40,000
debt	26,000
years in job	1 year
years at home	3 years
$\cdots$	$\cdots$

- No magic credit approval formula.
- Banks have lots of data.
  - customer information: salary, debt, etc.
  - whether or not they defaulted on their credit

A pattern exists. We don't know it. We have data to learn it.

Formal definition of the problem:

Item	Mathematical notation
Salary, debt, years in residence,$\cdots$	input $\boldsymbol{x} \in \mathcal{R}^n = \mathcal{X}$. Where $\mathcal{X}$ is the input space (set of all possible inputs)
Approve credit or not	output $\boldsymbol{y} \in \{-1\,(no), +1\,(yes)\} = \mathcal{Y}$. Where $\mathcal{Y}$ is the output space (set of all possible outputs)
True relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$	target function $f : \mathcal{X} \to \mathcal{Y}$. (The target f is an ideal formula / function for credit approval but unknown.)
Data on customers	data set $\mathcal{D} = \{(\boldsymbol{x}_1, \boldsymbol{y}_1), \cdots, (\boldsymbol{x}_m, \boldsymbol{y}_m)\}$. $(\boldsymbol{y}_i = f(\boldsymbol{x}_i).)$ for $i = 1, 2, \cdots, m$ (inputs corresponding to previous customers and the correct credit decision for them in hindsight)

Learning process

- Start with a set of candidate hypotheses $\mathcal{H}$ which you think are likely to represent f. $\mathcal{H} = \{h_1, h_2, \cdots, \}$ is called the hypothesis set or model.

- Select a hypothesis g from $\mathcal{H}$. The way we do this is called a learning algorithm.
- Use g for new customers. We hope $g \approx f$.
- $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{D}$ are given by the learning problem; The target f is fixed but unknown.
- We choose $\mathcal{H}$ and the learning algorithm.

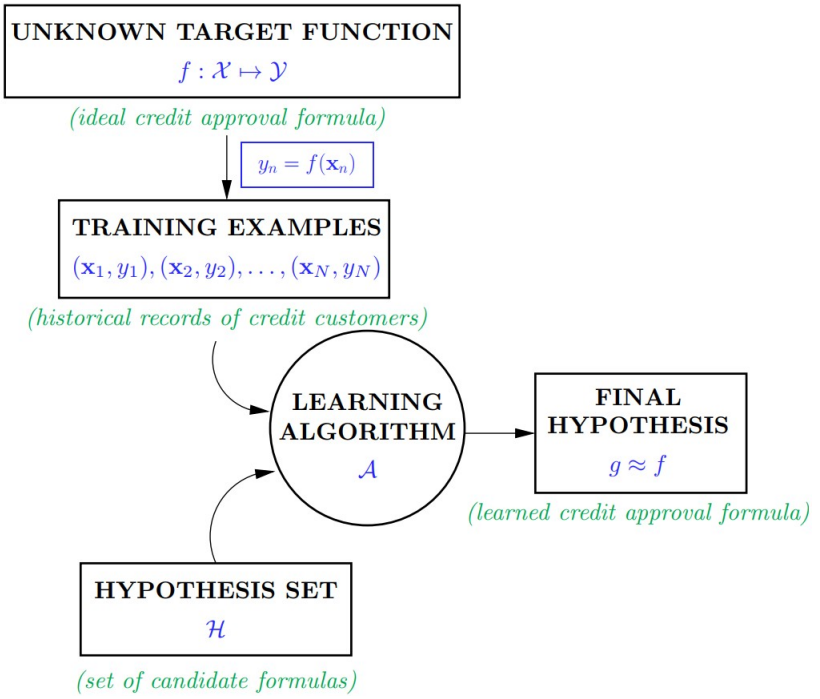


Figure 1.7: Elements of Machine Learning

1.3 Perceptron Learning Algorithm

- Input vector $\boldsymbol{x} = [x_1, \cdots, x_n]^\top$.
- Give importance weights to the different inputs and compute a "Credit Score"

$$\text{“}CreditScore\text{”} = \sum_{i=1}^{n} w_i x_i.$$

- Approve credit if the "Credit Score" is acceptable.
- Approve credit if $\sum_{i=1}^{n} w_i x_i > threshold$, ("Credit Score" is good)
- Deny credit if $\sum_{i=1}^{n} w_i x_i < threshold$. ("Credit Score" is bad)
  can be written formally as

$$
\begin{aligned}
h(x) &= sign\left(\left(\sum_{i=1}^{n} w_i x_i\right) - threshold\right) \\
&= sign\left(\left(\sum_{i=1}^{n} w_i x_i\right) + \underbrace{(-threshold)}_{w_0}\underbrace{(+1)}_{x_0}\right) \\
&= sign\left(\sum_{i=0}^{n} w_i x_i\right) \\
&= sign\left(\boldsymbol{w}^\top \boldsymbol{x}\right)
\end{aligned}
$$

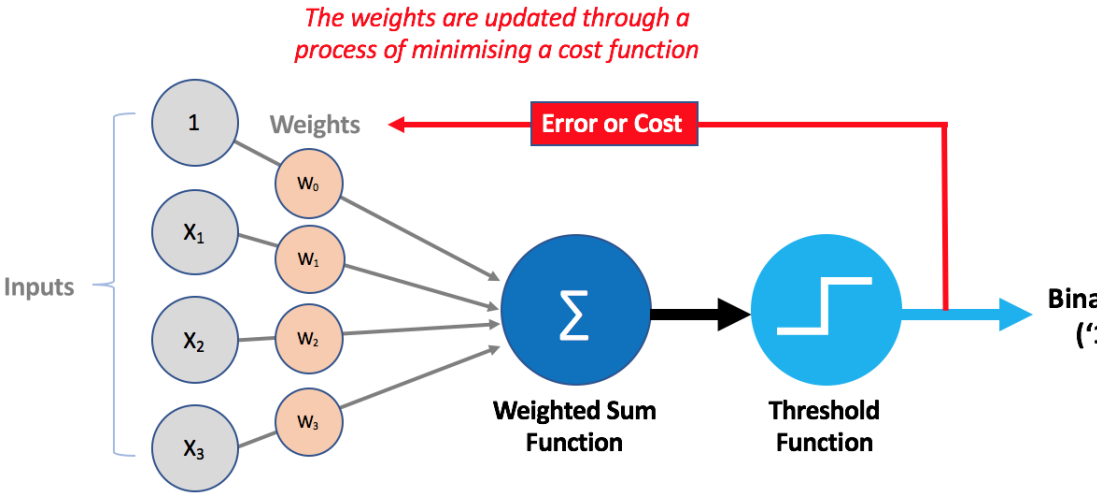


Figure 1.8: Perceptron

Id. No.	Petal Length	Petal Width	Sepal length	Sepal Width	Species
1	5.1	3.5	1.4	0.2	1
2	7.0	3.2	4.7	1.4	0
3	5.2	3.4	1.6	0.3	1

Figure 1.9: Training data

- The set of hypothesis $h(x)$ is called Hypothesis set $\mathcal{H} = \{h(x)\}$. This hypothesis set is called **"Perceptron"** or **"Linear Separator"**

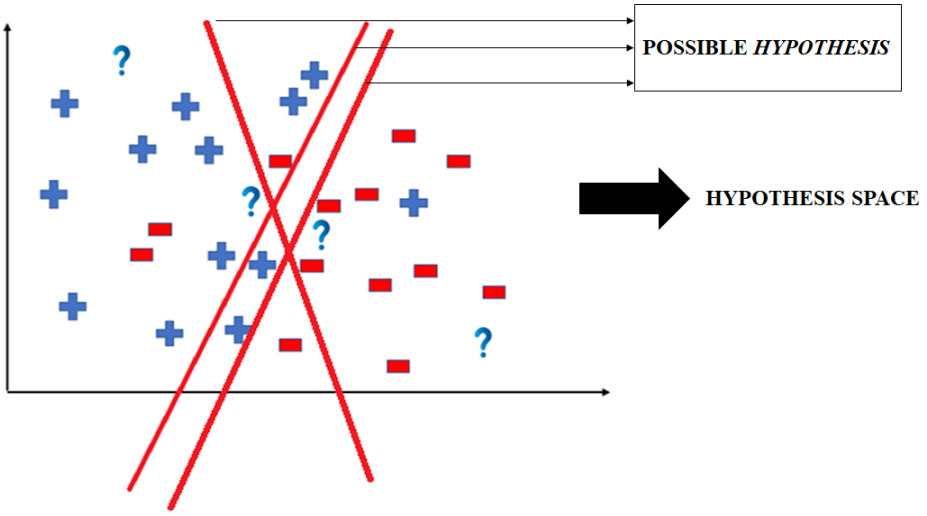


Figure 1.10: Hypothesis set

- Selection of final hypothesis g from $\mathcal{H}$

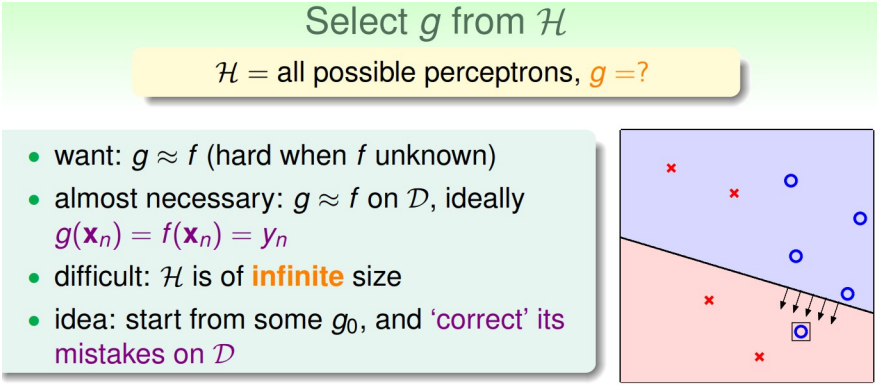


Figure 1.11: Selection of hypothesis

- How to choose the importance weights

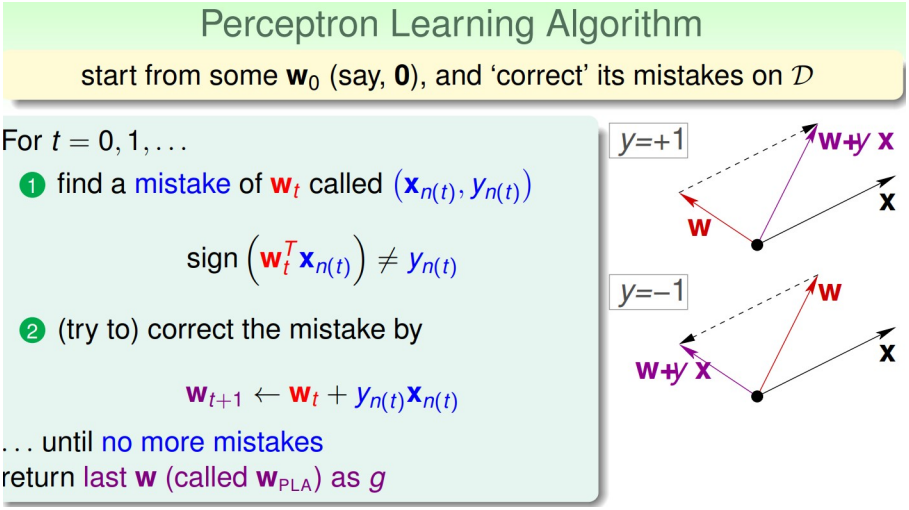


Figure 1.12: Perceptron Learning Algorithm

1.3.1 Linear Separability

- If the positive and negative points are linearly separable, then with a finite number of steps the perceptron algorithm will converge.
- If PLA stops (i.e. no more mistakes), (necessary condition) $\mathcal{D}$ allows some $\boldsymbol{w}$ to make no mistake.
- call such $\mathcal{D}$ linear separable.

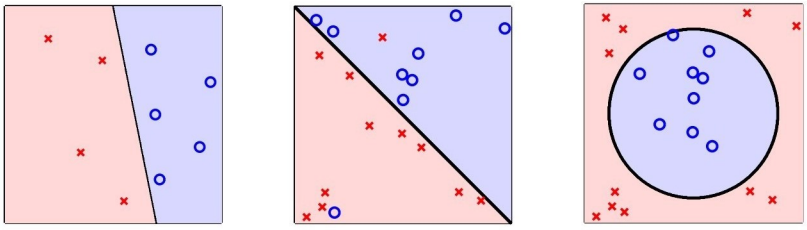


Figure 1.13: Linearly separable problem

1.4 Some Binary classification problems

- credit approve/disapprove
- email spam/non-spam
- patient sick/not sick
- ad profitable/not profitable
- answer correct/incorrect (KDDCup 2010)

Figure 1.14: Binary classification problems

1.5 Types of Machine Learning

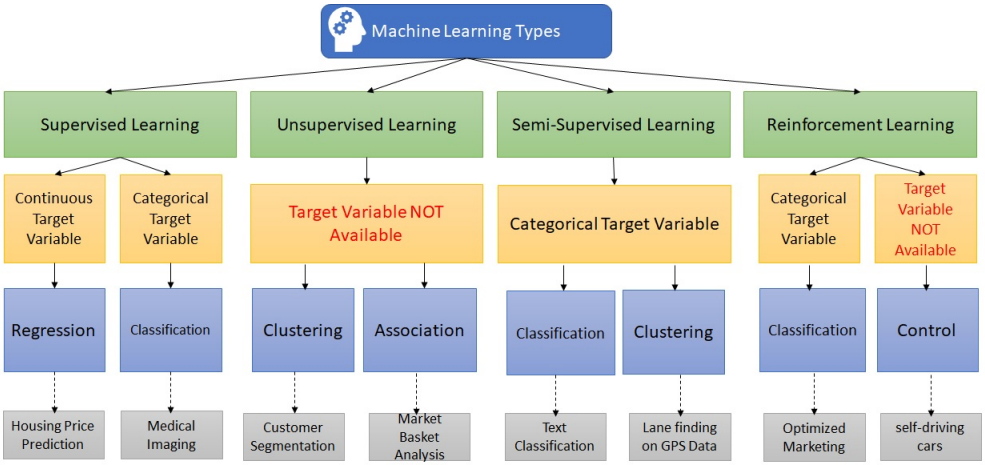


Figure 1.15: Types of Machine Learning

1.5.1 Supervised learning algorithms

experience a dataset containing features, but each example is also associated with a **label** or **target**. For example, the Iris dataset (fig. 1.6) is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements.

Supervised learning involves observing several examples of a random vector $\boldsymbol{x}$ and an associated value or vector $\boldsymbol{y}$, then learning to predict $\boldsymbol{y}$ from $\boldsymbol{x}$, usually by estimating $p(y|x)$

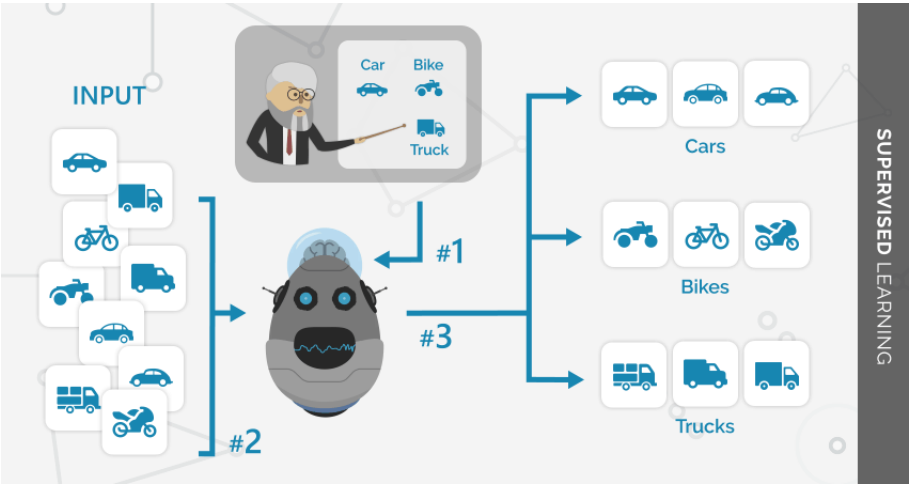


Figure 1.16: Supervised Learning

1.5.2 Unsupervised learning algorithms

experience a dataset containing many features, then learn useful properties of the structure of this dataset. Unsupervised learning algorithms perform roles, like clustering, which consists of dividing the dataset into clusters of similar examples.

Roughly speaking, unsupervised learning involves observing several examples of a random vector $\boldsymbol{x}$ and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution.

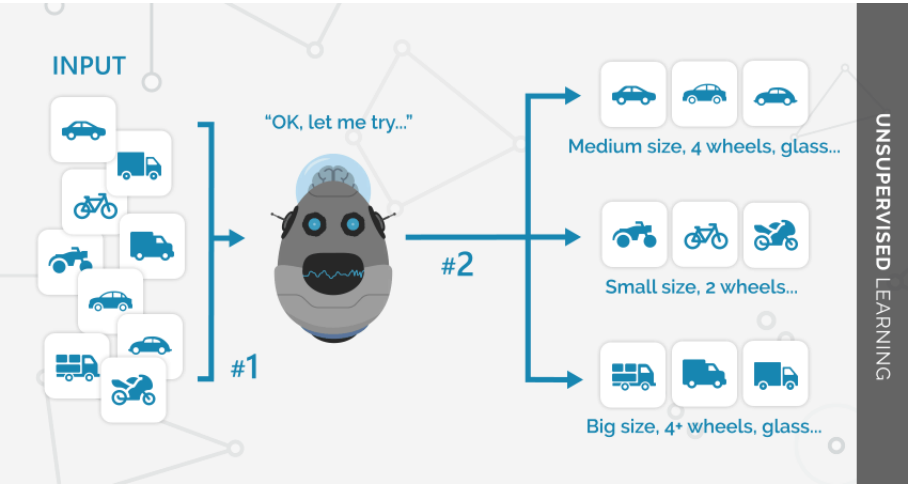


Figure 1.17: Unsupervised Learning

Traditionally, people refer to regression, classification (fig. 1.3) and structured output problems (fig. 1.4) as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning.

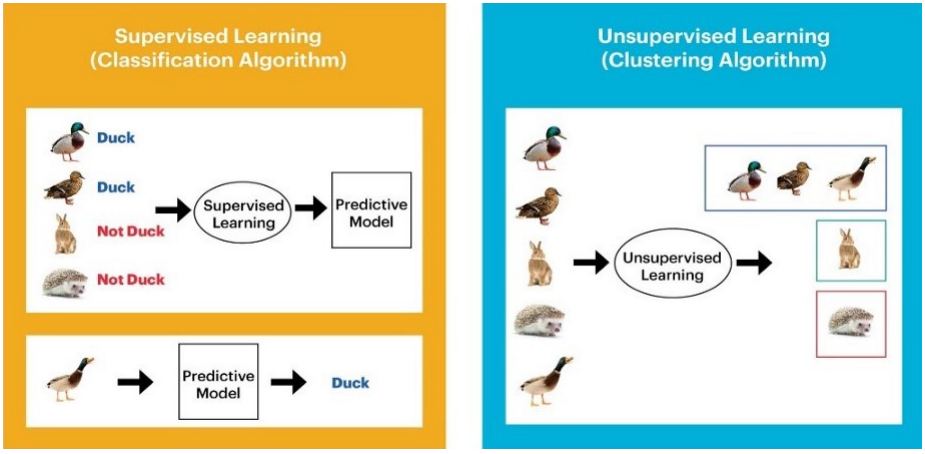


Figure 1.18: Supervised Vs Unsupervised Learning

1.5.3 Semi-supervised learning algorithms

Semi-supervised learning is supervised learning where the training data contains very few labelled examples and a large number of unlabelled examples. The goal of a semi-supervised learning model is to make effective use of all of the available data, not just the labelled data like in supervised learning. This can be achieved

by learning an initial model from the labelled training data available.

Making effective use of unlabelled data may require the use of or inspiration from unsupervised methods such as clustering and density estimation. Once groups or patterns are discovered, the model learnt with few labelled examples in the previous step, may be used to label the unlabelled examples. The newly labelled data are then incorporated into the already available training set and used to retrain the model, hence generally improving it.

It is common for many real-world supervised learning problems to be examples of semi-supervised learning problems given the expense or computational cost for labelling examples. For example, classifying photographs requires a dataset of photographs that have already been labelled by human operators. Many problems from the fields of computer vision (image data), natural language processing (text data), and automatic speech recognition (audio data) fall into this category and cannot be easily addressed using standard supervised learning methods.

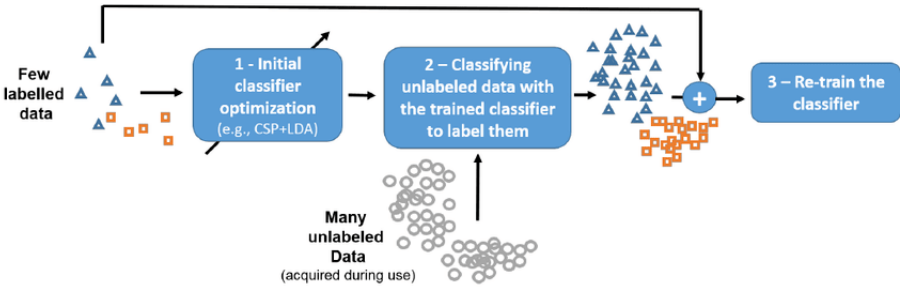


Figure 1.19: Semi-supervised learning

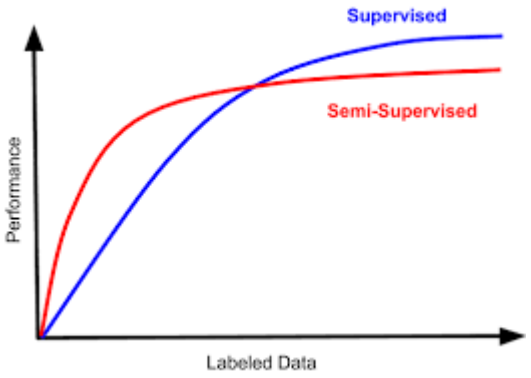


Figure 1.20: Supervised Vs Semi-supervised learning

1.5.4 Reinforcement learning

Reinforcement Learning has four essential elements:

- **Agent** : The program you train, with the aim of doing a job you specify.
- **Environment** : The world, real or virtual, in which the agent performs actions.
- **Action** : A move made by the agent, which causes a status change in the environment.
- **Rewards** : The evaluation of an action, which can be positive or negative.

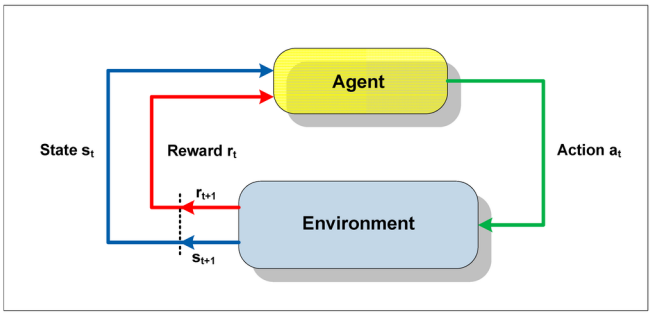


Figure 1.21: Reinforcement Learning

Example: Creating A Personalized Learning System

- **Agent** : The program that decides what to show next in an online learning catalog.
- **Environment** : The learning system.
- **Action** : Playing a new class video and an advertisement.
- **Reward** : Positive if the user chooses to click the class video presented; greater positive reward if the user chooses to click the advertisement; negative if the user goes away.

This program can make a personalized class system more valuable. The user can benefit from more effective learning and the system can benefit through more effective advertising.

1.6 Applications of Machine Learning

Popular applications of machine learning include the following:

- Email spam detection
- Face detection and matching (e.g., iPhone X, Windows laptops, etc.)
- Web search (e.g., DuckDuckGo, Bing, Baidu, Google)
- Sports predictions
- Post office (e.g., sorting letters by zip codes)
- ATMs (e.g., reading checks)
- Credit card fraud
- Stock predictions
- Smart assistants (Apple Siri, Amazon Alexa, . . .)
- Product recommendations (e.g., Walmart, Netflix, Amazon)
- Self-driving cars (e.g., Uber, Tesla)
- Language translation (Google translate)
- Sentiment analysis
- Drug design
- Medical diagnoses

1.7 Common loss functions in Machine Learning

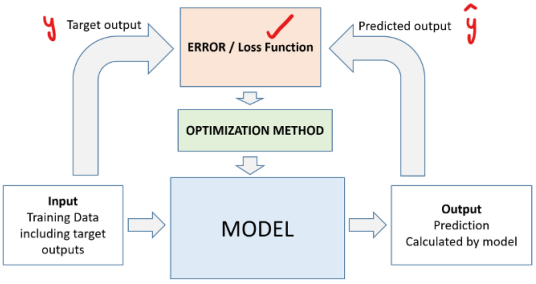


Figure 1.22: Loss calculation as a component of Supervised Learning work flow

Loss function: Often used synonymously with **cost function**; sometimes also called **error function**. In some contexts *the loss for a single data point*, where as *the cost function refers to the overall (average or summed) loss over the entire dataset*. Sometimes also called **empirical risk**.

There are various **factors involved in choosing a loss function** for specific problem such as type of machine learning algorithm chosen, ease of calculating the derivatives and to some degree the percentage of outliers in the data set.

1.7.1 Regression losses

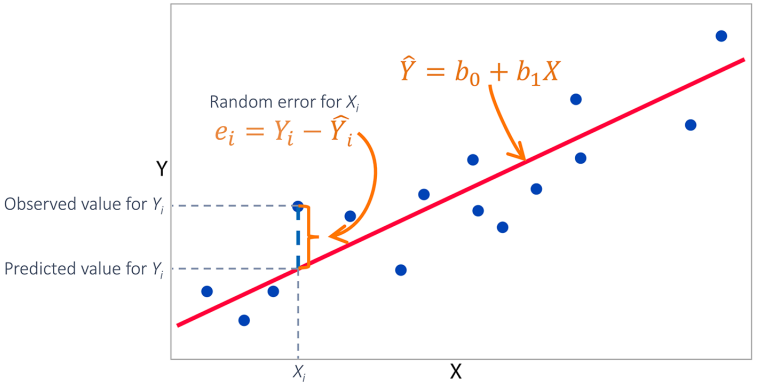


Figure 1.23: Regression error

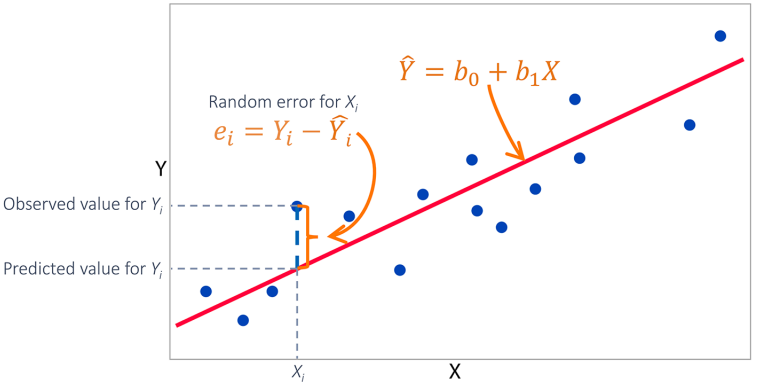


Figure 1.24: Regression error

Observed value y	Predicted value $\hat{y}$ by I model	Error for I model	Predicted value $\hat{y}$ by II model	Error for II model
100	101	-1	110	-10
200	199	$+1$	190	$+10$
300	301	-1	310	-10
400	399	$+1$	390	$+10$

NOTE
n - Number of training examples.
i - ith training example in a data set.
y_i - Observed value for ith training example.
$\hat{y}_i$ - Predicted value for ith training example.
Accuracy in % + Error in % = 100%.

- **Mean Bias Error** This is not a popular loss function. This is same as MAE with the only difference that we don't take absolute values. Clearly there's a need for caution as positive and negative errors could cancel each other out. Although less accurate in practice, it could determine if the model has positive bias or negative bias.

$$MBE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i) \tag{1.1}$$

- **Mean Absolute Error / $L1$ Loss** Mean absolute error, on the other hand, is measured as the average of absolute differences between predictions and actual observations. Like MSE, this measures the magnitude of error without considering their direction. Unlike MSE, MAE needs more complicated tools such as linear programming to compute the gradients. MAE is more robust to outliers since it does not make use of square.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|(y_i - \hat{y}_i)| \tag{1.2}$$

- **Mean Square Error / Quadratic Loss / $L2$ Loss:** As the name suggests, Mean square error is measured as the average of squared difference between predictions and actual observations. It's only concerned with the average magnitude of error irrespective of their direction. However, due to squaring,

predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. MSE has nice mathematical properties which makes it easier to calculate gradients.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{1.3}$$

- **Root Mean Square Error** This is calculated by applying square root function on MSE

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}} \tag{1.4}$$

- **Mean Percentage Error** This is calculated as follows

$$MPE = \frac{1}{n}\sum_{i=1}^{n}\frac{(y_i - \hat{y}_i)}{y_i} \tag{1.5}$$

- **Mean Absolute Percentage Error** This is calculated as follows

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\frac{|(y_i - \hat{y}_i)|}{y_i} \tag{1.6}$$

- **Adjusted R-square** It represents the proportion of variation (i.e., information), in your data, explained by the model. The r-squared value ranges from 0 to 1, where 0 means no relationship, and 1 means 100% related. This corresponds to the overall quality of the model. The higher the adjusted R^2, the better the model

1.7.2 Classification losses

- Hinge Loss / Multi class SVM Loss In simple terms, the score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one). And hence hinge loss is used for maximum-margin classification, most notably for support vector machines. Although not differentiable, it's a convex function which makes it easy to work with usual convex optimizers used in machine learning domain.

$$HL = \sum_{j \neq y_i} max(0, S_j - S_{y_i} + 1) \tag{1.7}$$

Consider an example where we have three training examples and three classes to predict — Dog, cat and horse.



	Image #1	Image #2	Image #3
Dog	-0.39	-4.61	1.03
Cat	1.49	3.28	-2.37
Horse	4.21	1.46	-2.27

Figure 1.25: Hinge loss/ Multi class SVM loss

```
## 1st training example
max(0, (1.49) - (-0.39) + 1) + max(0, (4.21) - (-0.39) + 1)
max(0, 2.88) + max(0, 5.6)
2.88 + 5.6
8.48 (High loss as very wrong prediction)

## 2nd training example
max(0, (-4.61) - (3.28)+ 1) + max(0, (1.46) - (3.28)+ 1)
max(0, -6.89) + max(0, -0.82)
0 + 0
0 (Zero loss as correct prediction)

## 3rd training example
max(0, (1.03) - (-2.27)+ 1) + max(0, (-2.37) - (-2.27)+ 1)
max(0, 4.3) + max(0, 0.9)
4.3 + 0.9
5.2 (High loss as very wrong prediction)
```

Figure 1.26: Computing hinge losses for all 3 training examples

- **Cross Entropy Loss/Negative Log Likelihood** This is the most common setting for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label. In short, it is a product of log of the predicted probability for the ground truth class. An important aspect of this is that cross entropy loss penalizes heavily the predictions that are confident but wrong.

$$L(X_i, Y_i) = -\sum_{j=1}^{c} y_{ij} * \log(p_{ij})$$

where Y_i is one − hot encoded target vector $(y_{i1}, y_{i2}, \ldots, y_{ic})$,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$p_{ij} = f(X_i) = Probability\ that\ i_{th}\ element\ is\ in\ class\ j$

Figure 1.27: Cross Entropy Loss

Chapter 2

Optimization of Model parameters

2.1 Optimization of Model parameters: An Iterative Approach

A Machine Learning model is trained by starting with an initial guess for the weights and bias (model parameters) and iteratively adjusting those guesses until learning the weights and bias with the lowest possible loss.

The following figure suggests the iterative trial-and-error process that machine learning algorithms use to train a model:

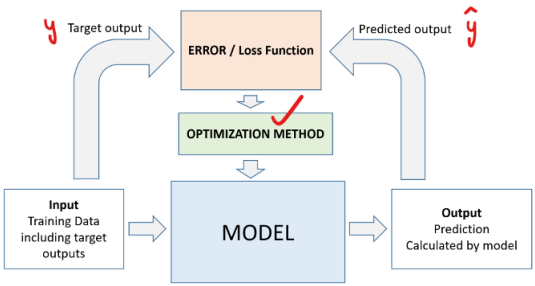


Figure 2.1: Parameter tuning as a component of Supervised Learning work flow

Iterative strategies are prevalent in machine learning, primarily because they scale so well to large data sets.

The "model" takes one or more features as input and returns one prediction ($\hat{y}$) as output. To simplify, consider a model that takes one feature and returns

one prediction:

$$\hat{y} = b + w_1 x_1 = w_0 + w_1 x_1$$

What initial values should we set for b and w_1? For linear regression problems, it turns out that the starting values aren't important. We could pick random values, but we'll just take the following trivial values instead: $b = 0$ and $w_1 = 0$.

The "Compute Loss" part of the diagram is the loss function that the model will use. Suppose we use the root mean squared loss function. At last, we've reached the "Compute parameter updates" part of the diagram.

It is here that the machine learning system examines the value of the loss function and generates new values for b and w_1. For now, just assume that this mysterious box devises new values and then the machine learning system re-evaluates all those features against all those labels, yielding a new value for the loss function, which yields new parameter values. And the learning continues iterating until the algorithm discovers the model parameters with the lowest possible loss. Usually, you iterate until overall loss stops changing or at least changes extremely slowly. When that happens, we say that the model has **converged**.

Suppose we had the time and the computing resources to calculate the loss for all possible values of w_1. For the kind of regression problems we've been examining, the resulting plot of loss vs. weight(w_1) will always be **convex**. In other words, the plot will always be bowl-shaped, kind of like this:

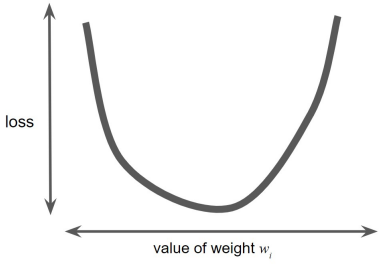


Figure 2.2: Regression problems yield convex loss vs. weight plots.

Convex problems have only one minimum; that is, only one place where the slope is exactly 0. That minimum is where the loss function converges.

Calculating the loss function for every conceivable value of over the entire data set would be an inefficient way of finding the convergence point.

2.2 Gradient Descent

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient. In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Linear Regression and weights in neural networks.

2.2.1 Batch Gradient Descent

The first stage in gradient descent is to pick a starting value (a starting point) for w_1. The starting point doesn't matter much; therefore, many algorithms simply set w_1 to 0 or pick a random value. The following figure 2.5 shows that we've picked a starting point slightly greater than 0. The gradient descent algorithm then calculates the gradient of the loss curve at the starting point. The gradient vector has both a direction and a magnitude.

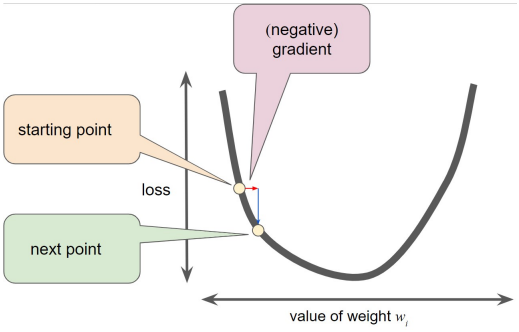


Figure 2.3: A gradient step moves us to the next point on the loss curve.

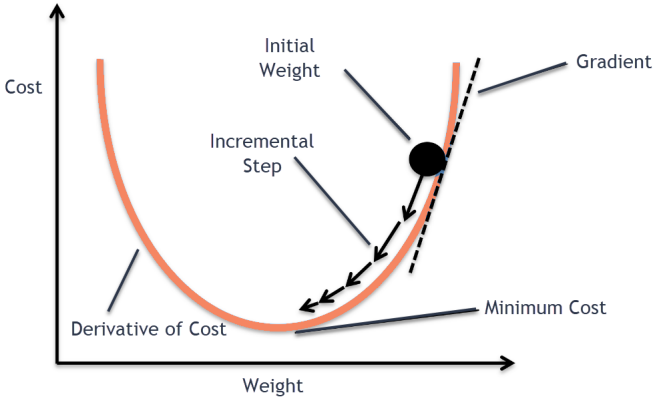


Figure 2.4: Cost function with one parameter

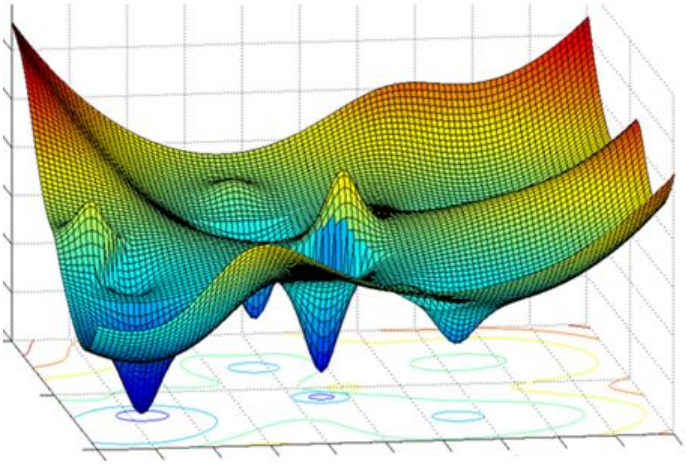


Figure 2.5: Cost function with two parameter

Here in figure 2.5, the gradient of the loss is equal to the derivative (slope) of the curve, and tells us which way is "increasing" or "decreasing". When there are multiple weights, the gradient is a vector of partial derivatives with respect to the weights. The gradient always points in the direction of steepest increase in the loss function.

The gradient descent algorithm takes a step in the direction of the negative gradient in order to reduce loss as quickly as possible. To determine the next point along the loss function curve, the gradient descent algorithm adds some fraction of the gradient's magnitude to the starting point as shown in the figure

2.5.

Gradient descent algorithms multiply the gradient by a scalar known as the **learning rate** (also sometimes called step size) to determine the next point. For example, if the gradient magnitude is 2.5 and the learning rate is 0.01, then the gradient descent algorithm will pick the next point 0.025 away from the previous point.

Hyperparameters are the knobs that programmers tweak in machine learning algorithms. Most machine learning programmers spend a fair amount of time tuning the learning rate. If you pick a learning rate that is too small, learning will take too long:

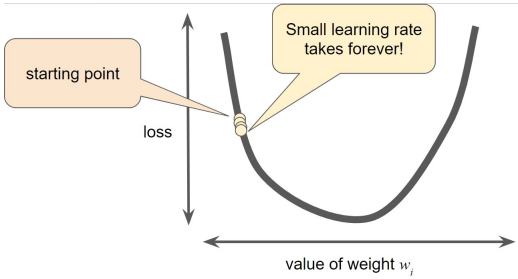


Figure 2.6: Learning rate is too small.

Conversely, if you specify a learning rate that is too large, the next point will perpetually bounce haphazardly across the bottom of the well (global minimum).

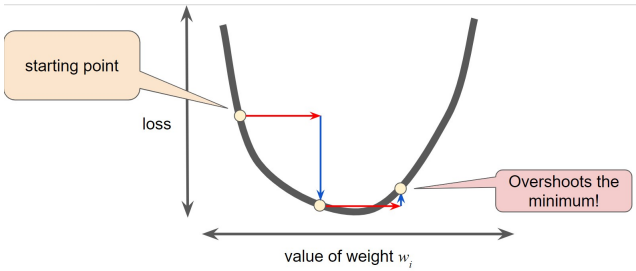


Figure 2.7: Learning rate is too large.

There's an optimal learning rate for every regression problem. The optimal value is related to how flat the loss function is. If you know the gradient of the loss function is small then you can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size.

In gradient descent, a batch is the total number of examples you use to calculate the gradient in a single iteration. So far, we've assumed that the batch has been the entire data set. When working at Google scale, data sets often contain

billions or even hundreds of billions of examples. Furthermore, Google data sets often contain huge numbers of features. Consequently, a batch can be enormous. A very large batch may cause even a single iteration to take a very long time to compute.

A large data set with randomly sampled examples probably contains redundant data. In fact, redundancy becomes more likely as the batch size grows. Some redundancy can be useful to smooth out noisy gradients, but enormous batches tend not to carry much more predictive value than large batches.

What if we could get the right gradient on average for much less computation? By choosing examples at random from our data set, we could estimate (albeit, noisily) a big average from a much smaller one.

Advantages of Batch Gradient Descent

- Less oscillations and noisy steps taken towards the global minima of the loss function due to updating the parameters by computing the average of all the training samples rather than the value of a single sample
- It can benefit from the vectorization which increases the speed of processing all training samples together
- It produces a more stable gradient descent convergence and stable error gradient than stochastic gradient descent
- It is computationally efficient as all computer resources are not being used to process a single sample rather are being used for all training samples

Disadvantages of Batch Gradient Descent

- Sometimes a stable error gradient can lead to a local minima and unlike stochastic gradient descent no noisy steps are there to help get out of the local minima
- The entire training set can be too large to process in the memory due to which additional memory might be needed
- Depending on computer resources it can take too long for processing all the training samples as a batch

2.2.2 Stochastic gradient descent (SGD)

uses only a single example (a batch size of 1) per iteration. Given enough iterations, SGD works but is very noisy. The term “stochastic” indicates that the one

example comprising each batch is chosen at random.

Advantages of Stochastic Gradient Descent

- It is easier to fit into memory due to a single training sample being processed by the network
- It is computationally fast as only one sample is processed at a time
- For larger datasets it can converge faster as it causes updates to the parameters more frequently
- Due to frequent updates the steps taken towards the minima of the loss function have oscillations which can help getting out of local minimums of the loss function (in case the computed position turns out to be the local minimum)

Disadvantages of Stochastic Gradient Descent

- Due to frequent updates the steps taken towards the minima are very noisy. This can often lead the gradient descent into other directions.
- Also, due to noisy steps it may take longer to achieve convergence to the minima of the loss function
- Frequent updates are computationally expensive due to using all resources for processing one training sample at a time
- It loses the advantage of vectorized operations as it deals with only a single example at a time

2.2.3 Mini-batch stochastic gradient descent

(mini-batch SGD) is a compromise between full-batch iteration and SGD. A mini-batch is typically between 10 and 1,000 examples, chosen at random. Mini-batch SGD reduces the amount of noise in SGD but is still more efficient than full-batch.

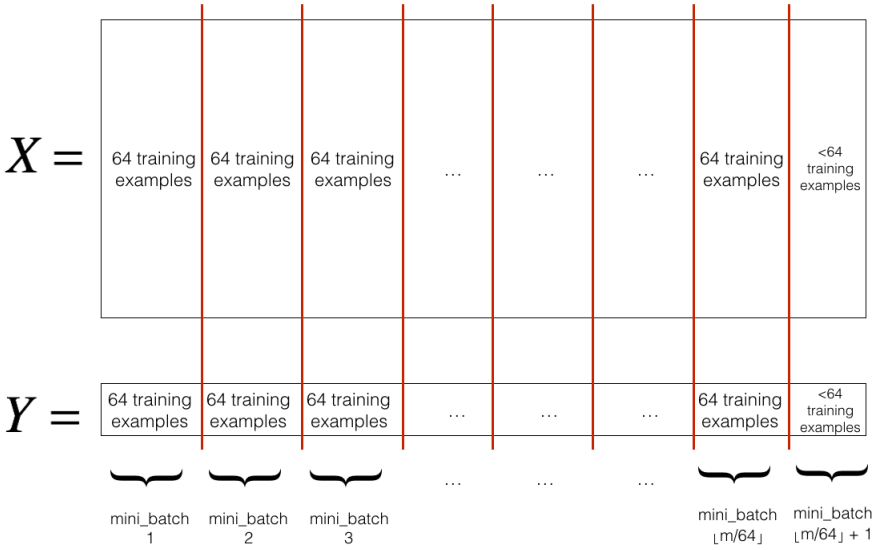


Figure 2.8: Mini batch definition

Advantages of Stochastic Gradient Descent

- Easily fits in the memory
- It is computationally efficient
- Benefit from vectorization
- If stuck in local minimums, some noisy steps can lead the way out of them
- Average of the training samples produces stable error gradients and convergence

We can divide the dataset of 2000 examples into batches of 500 then it will take 4 iterations to complete 1 epoch.

To simplify the explanation, we focused on gradient descent for a single feature. Rest assured that gradient descent also works on feature sets that contain multiple features.

Chapter 3

Model Validation

3.1 Train and Test set approach

We usually partition a data set into a training set and a test set. This partitioning enables us to train on one set of examples and then to test the model against a different set of examples. With two partitions, the workflow could look as follows:

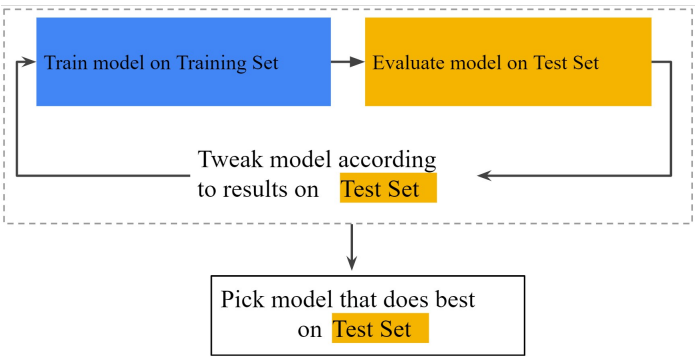


Figure 3.1: Model validation using Test set only.

The test set should meet the following two conditions:

- Is large enough to yield statistically meaningful results.
- Is representative of the data set as a whole. In other words, don't pick a test set with different characteristics than the training set.

Assuming that your test set meets the preceding two conditions, your goal is to create a model that generalizes well to new data. The test set should serve as

a proxy for new data. In the figure, "Tweak model" means adjusting anything about the model you can dream up—from changing the learning rate, to adding or removing features, to designing a completely new model from scratch. At the end of this workflow, you pick the model that does best on the test set.

Never train on test data.

If you are seeing surprisingly good results on your evaluation metrics, it might be a sign that you are accidentally training on the test set. For example, high accuracy might indicate that test data has leaked into the training set.

For example, consider a model that predicts whether an email is spam, using the subject line, email body, and sender's email address as features. We apportion the data into training and test sets, with an 80-20 split. After training, the model achieves 99% precision on both the training set and the test set. We'd expect a lower precision on the test set, so we take another look at the data and discover that many of the examples in the test set are duplicates of examples in the training set (we neglected to scrub duplicate entries for the same spam email from our input database before splitting the data). We've inadvertently trained on some of our test data, and as a result, we're no longer accurately measuring how well our model generalizes to new data.

Dividing the data set into two sets is a good idea, but not a panacea. You can greatly reduce your chances of overfitting by partitioning the data set into the three subsets shown in the following figure:

Use the validation set to evaluate results from the training set. Then, use the test set to double-check your evaluation after the model has "passed" the validation set. The following figure shows this new workflow:

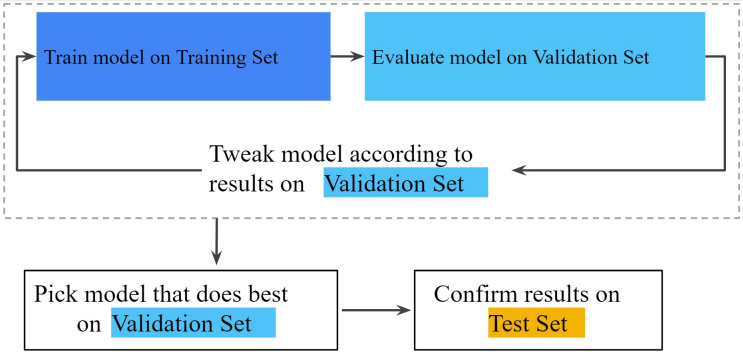


Figure 3.2: Model validation using Validation and Test.

In this improved workflow:

- Pick the model that does best on the validation set.

- Double-check that model against the test set.

This is a better workflow because it creates fewer exposures to the test set.

Training Dataset: The sample of data used to fit the model.

Validation Dataset: The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.

Test Dataset: The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.

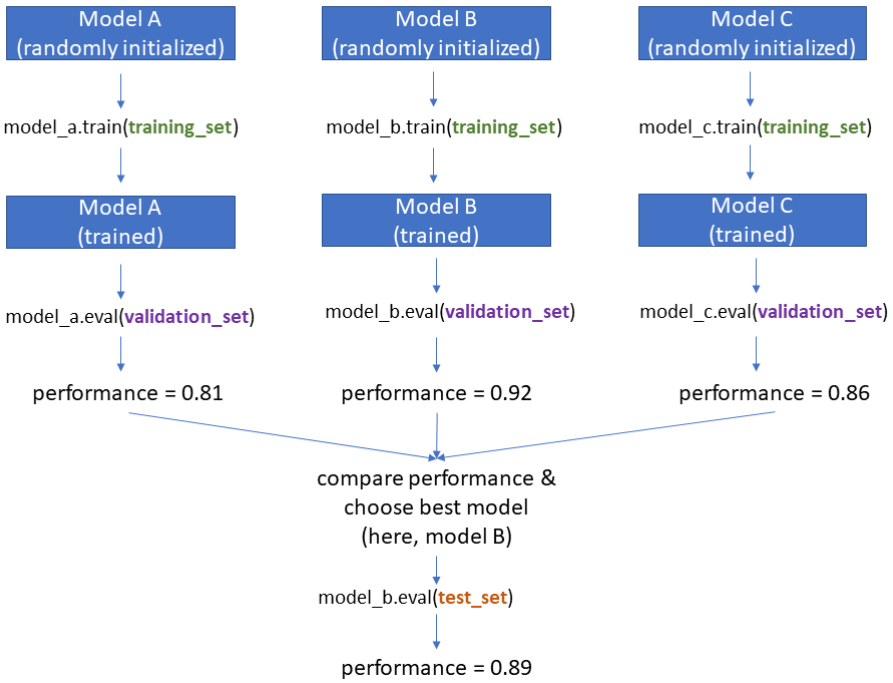


Figure 3.3: Model validation using Validation and Test sets.

Why can't we just use one data set? Let's imagine what would happen if we used ALL our data as a "training set." When we wanted to evaluate performance, we'd just look at the training set performance. Now, if we get lucky, the training set performance might be reflective of how the model would perform on data it

has never seen before. But if we get unlucky, the model has simply memorized the training data examples and when we feed it an example it's never seen before, it completely fails ("overfitting"). We have no way of figuring out whether we're lucky or unlucky – which is exactly why we need a validation set. The validation set consists of examples the model has never seen in training, so if we get good validation set performance, we can be encouraged that our model has learned useful generalizable principles.

But, if we have a training set and a validation set, why do we also need a test set? The test set is important because the step of "choosing the best model" (based on validation performance) can cause a form of overfitting. Think about it this way: let's say you tried a THOUSAND different models or model variations on your data, and you have validation set performance for all of them. The act of choosing the model with the best validation set performance inherently means that you, the human, have "tuned" the model details for the validation set. The performance value you see for the validation set on "the best model on the validation set" is inherently inflated. To get a non-inflated and more reliable estimate of how well this "best model" will do on data it's never seen before, we need to use more data it's never seen before! This is the test set. The test set performance will typically be slightly lower than the validation set performance.

3.1.1 Dataset split ratio

Now that you know what these datasets do, you might be looking for recommendations on how to split your dataset into Train, Validation and Test sets. This mainly depends on 2 things. First, the total number of samples in your data and second, on the actual model you are training.

Some models need substantial data to train upon, so in this case you would optimize for the larger training sets. Models with very few hyperparameters will be easy to validate and tune, so you can probably reduce the size of your validation set, but if your model has many hyperparameters, you would want to have a large validation set as well(although you should also consider cross validation). Also, if you happen to have a model with no hyperparameters or ones that cannot be easily tuned, you probably don't need a validation set too! All in all, like many other things in machine learning, the train-test-validation split ratio is also quite specific to your use case and it gets easier to make judgement as you train and build more and more models.

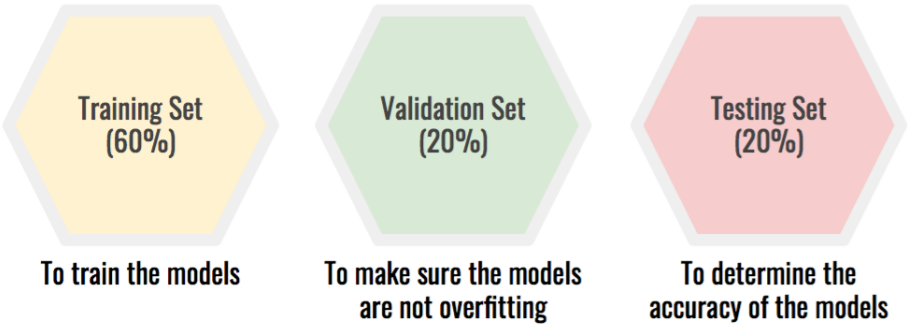


Figure 3.4: Split ratio for Train, Validation and Test sets.

However, let's first take a look at the concept of generating train/test splits in the first place. Why do you need them? Why can't you simply train the model with all your data and then compare the results with other models? We'll answer these questions first.

3.2 Cross Validation

Training a supervised machine learning model involves changing model weights using a training set. Later, once training has finished, the trained model is tested with new data – the testing set – in order to find out how well it performs in real life.

When you are satisfied with the performance of the model, you train it again with the entire dataset, in order to finalize it and use it in production.

However, when checking how well the model perform, the question how to split the dataset is one that emerges pretty rapidly. K-fold Cross Validation, is the solution.

Why we should split them in the first place when evaluating model performance? To find out the model that performs best in production, i.e. "when we really use it"

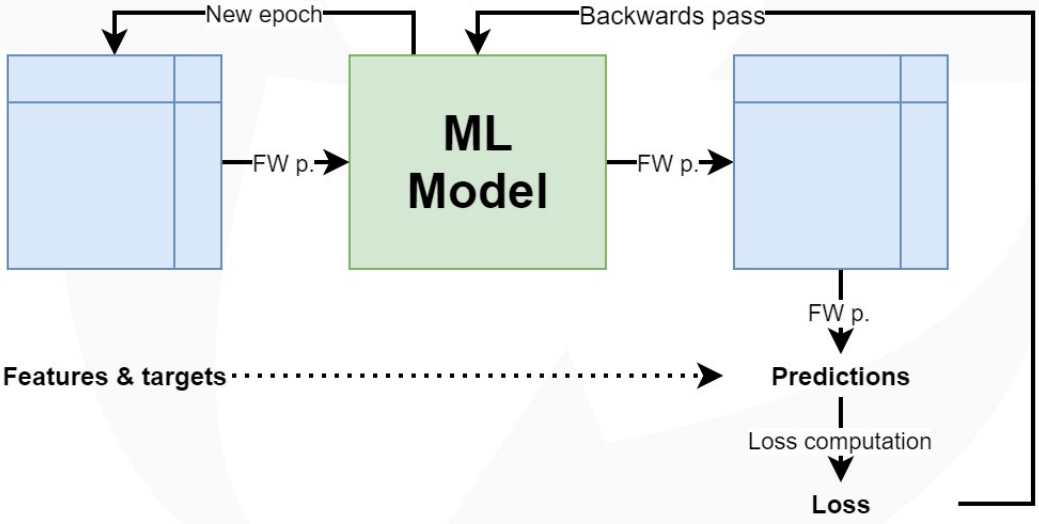


Figure 3.5: Training phase

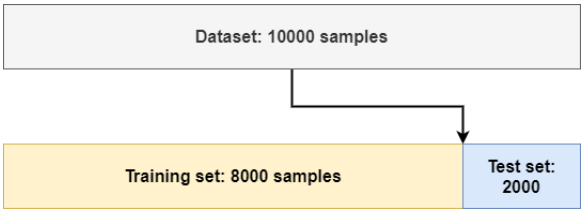


Figure 3.6: A simple hold-out split

Problems with simple hold-out approach:

- Data representativeness: all datasets, which are essentially samples, must represent the patterns in the population as much as possible. This becomes especially important when you generate samples from a sample (i.e., from your full dataset). For example, if the first part of your dataset has examples of one class, while the latter one has examples of another class, trouble is guaranteed when you generate the split as displayed above. Random shuffling may help you solve these issues.
- Data redundancy: if some samples appear more than once, a simple hold-out split with random shuffling may introduce redundancy between training and testing datasets. That is, identical samples belong to both datasets. This is problematic too, as data used for training thus leaks into the dataset for testing implicitly.

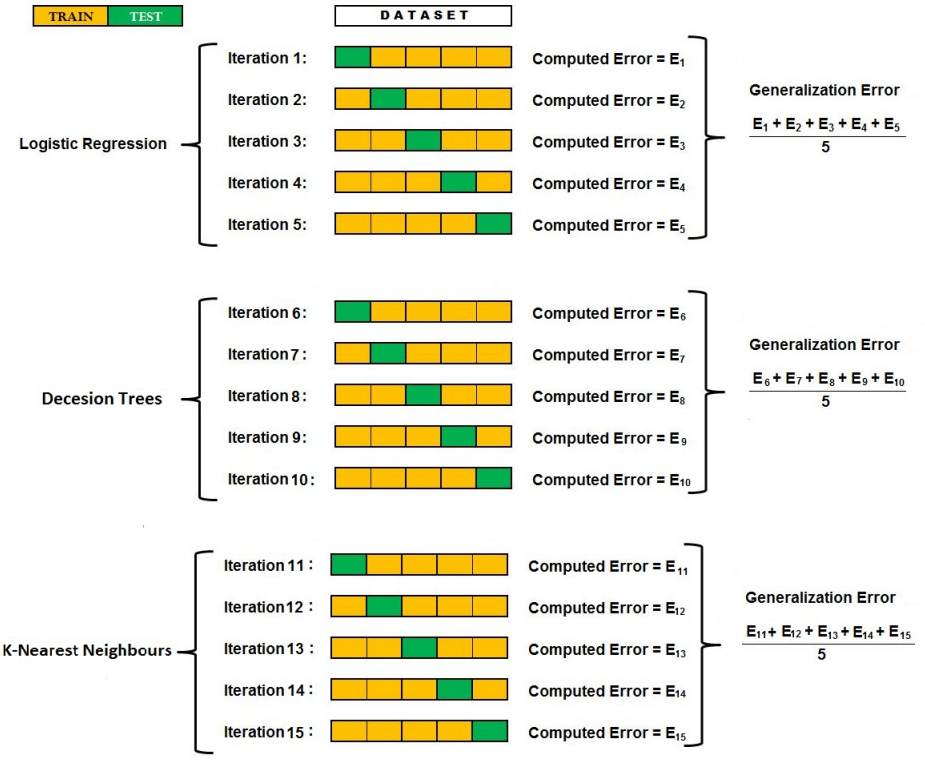


Figure 3.7: K-Fold Cross Validation

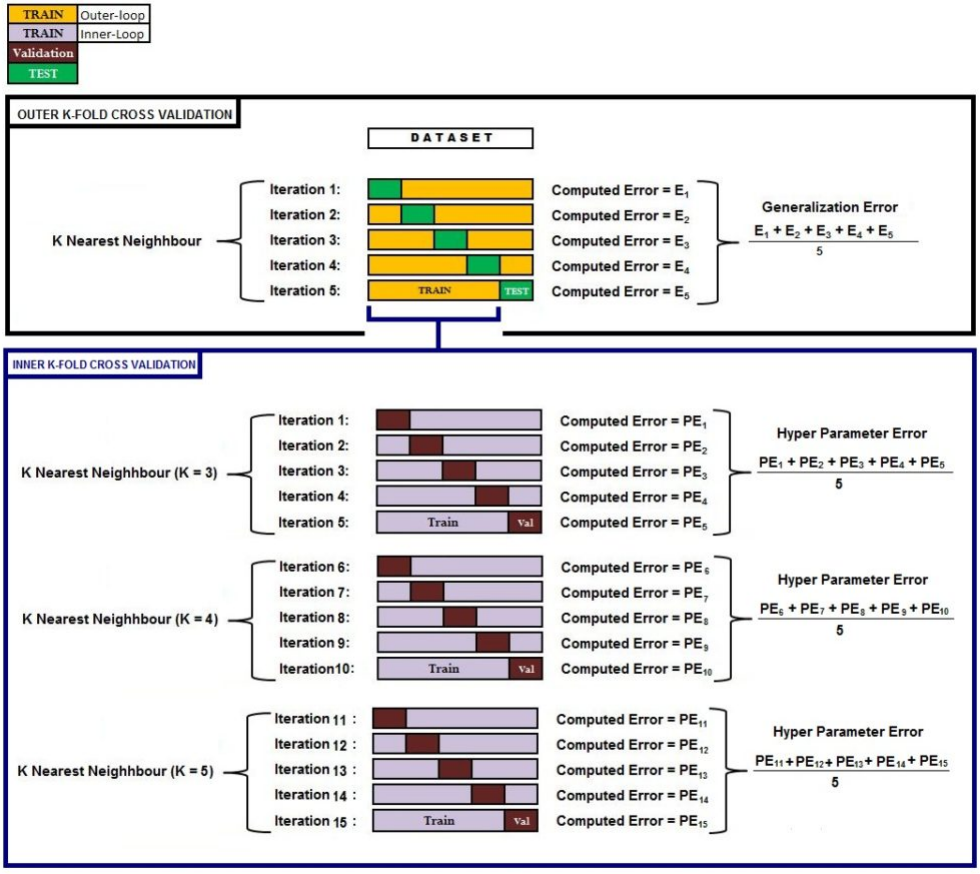


Figure 3.8: Nested K-Fold Cross Validation

3.2.1 Repeated K-Fold Cross Validation

Another variant of k-fold cross validation is Repeated k-Fold cross-validation where we perform k-fold cross-validation, then reshuffle the data and then perform k-fold again and so on. For example if we are performing 5-fold cross validation then the data is shuffled 5 different times thus providing us with 25 evaluations. This method is used to make sure that no data point is left behind even after randomly selecting the data points for train and test.

3.2.2 Stratified K-Fold Cross Validation

For example, we have a dataset with 120 observations and we are to predict the three classes 0, 1 and 2 using various classification techniques. Now if we perform k-fold cross validation then in the first fold, it picks the first 30 records for test and remaining for the training set. In the next iteration, it picks the second 30 records for test set and so on.

This methodology can cause a potential problem of a certain class not showing up on the training set and this will cause the trained model to fail when that class appears on the testing set. This problem can become more severe especially if the data is sorted by class. On top of all this, it is possible that the dataset has imbalanced class. For example, out of the 120 records, 60 belong to the class 0, 40 records belong to class 1 while only 20 may belong to class 2. Thus, we need a proper representation of these classes in our training and test dataset, for which we use Stratified k-Fold Cross-Validation.

For example, if we perform Stratified k-Fold Cross-Validation with the above mentioned imbalanced classes then we first consider all the observations separately as per their class label. Thus, we will be dealing with three sets of data with one having observation belonging to class 0, another having observations of class 1 and a third set having observations of the class label 2. Each of these sets is then divided into 5 folds and the first fold of all these three classes are joined together to form the fold 1. Similarly, the second fold of all these three sets of observations are joined together to form the second fold and so on. This way the class labels gets balanced across the training and testing set and we are able to counter the 'unlucky splits' from happening.

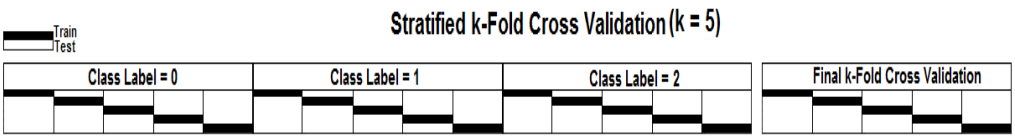


Figure 3.9: Stratified K-Fold Cross Validation

3.2.3 Leave-one-out cross-validation and Leave-p-out cross validation

In the k-Fold cross-validation, we discussed various examples where the value of k was 5. This way at every iteration, we created a model using 80% of the data. Among the most commonly used value of k is 10 where by doing so we are able to provide 90% of the data to the training set. If we further go with

this idea of maximising the number of observations for the training set, we get Leave-one-out cross-validation which is k fold cross validation only but with the k being almost equal to the sample size. This way if we have 100 observations then 99 observations are used to form the training set while the left, one observation, is used for evaluation. Thus, if we have n number of observations, we will end up having $n - 1$ iterations wherein each iteration, a single data point will be considered for testing. This method is the best way for learning a function and is more stable than the k-fold and saves us from deciding the number of folds as the value of k in k-fold can alter results which makes choosing the right value of k very crucial. However, leave-one-out is extremely computationally expensive and in comparison, K-fold is much quicker.

A variant of Leave-one-out cross-validation is Leave-p-out cross validation where rather than using only one observation for testing, we use p observations as the validation set while the remaining observations are used as the training set. This way the computational cost can be cut down. A Leave-p-out cross validation with p being 1 is Leave-one-out cross-validation only.

Chapter 4

Capacity, Overfitting and Underfitting

4.1 Introduction

The central challenge in machine learning is that our algorithm must perform well on new, previously unseen inputs - not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called **generalization** .

Typically, when training a machine learning model, we have access to a training set; we can compute some error measure on the training set, called the **training error** 4.1; and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the **generalization error**, also called the **test error** 4.2, to be low as well. The generalization error is defined as the expected value of the error on a new input. Here the expectation is taken across different possible inputs, drawn from the distribution of inputs we expect the system to encounter in practice.

We typically estimate the generalization error of a machine learning model by measuring its performance on a **test set** of examples that were collected separately from the training set. In our linear regression example, we trained the model by minimizing the training error,

$$\frac{1}{m^{train}} \parallel \boldsymbol{X}^{(\text{train})} \boldsymbol{w} - \boldsymbol{y}^{(\text{train})} \parallel_2^2 \tag{4.1}$$

but we actually care about the test error,

$$\frac{1}{m^{test}} \parallel \boldsymbol{X}^{(\text{test})} \boldsymbol{w} - \boldsymbol{y}^{(\text{test})} \parallel_2^2 \tag{4.2}$$

How can we affect performance on the test set when we can observe only the training set? The field of statistical learning theory provides some answers. The training and test data are generated by a probability distribution over datasets called the **data-generating process**. We typically make a set of assumptions known collectively as the **i.i.d. assumptions**. These assumptions are that the examples in each dataset are **independent** from each other, and that the training set and test set are **identically distributed**, drawn from the same probability distribution as each other. This assumption enables us to describe the data-generating process with a probability distribution over a single example. The same distribution is then used to generate every train example and every test example. We call that shared underlying distribution the **data-generating distribution**, denoted p_{data}. This probabilistic framework and the i.i.d. assumptions enables us to mathematically study the relationship between training error and test error.

The factors determining how well a machine learning algorithm will perform are its ability to

1. Make the training error small.

2. Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: **underfitting** and **overfitting** . Underfitting occurs when the model is not able to obtain a sufficiently low error value on the training set. Overfitting occurs when the gap between the training error and test error is too large. We can control whether a model is more likely to overfit or underfit by altering its **capacity**. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set. One way to control the capacity of a learning algorithm is by choosing its **hypothesis space** , the set of functions that the learning algorithm is allowed to select as being the solution. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials, rather than just linear functions, in its hypothesis space. Doing so increases the model's capacity.

By introducing x^2 as another feature provided to the linear regression model, we can learn a model that is quadratic as a function of x:

$$\hat{y} = b + w_1 x + w_2 x^2. \tag{4.3}$$

Though this model implements a quadratic function of its input, the output is still a linear function of the parameters, so we can still use the normal equations to train the model in closed form.

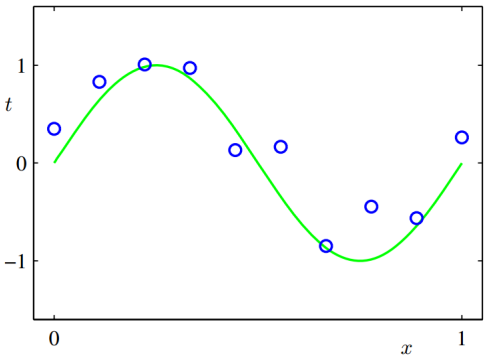


Figure 4.1: Plot of a training data set of $N = 10$ points, shown as blue circles, each comprising an observation of the input variable x along with the corresponding target variable t. The green curve shows the function $sin(2\pi x)$ used to generate the data. Our goal is to predict the value of t for some new value of x, without knowledge of the green curve.

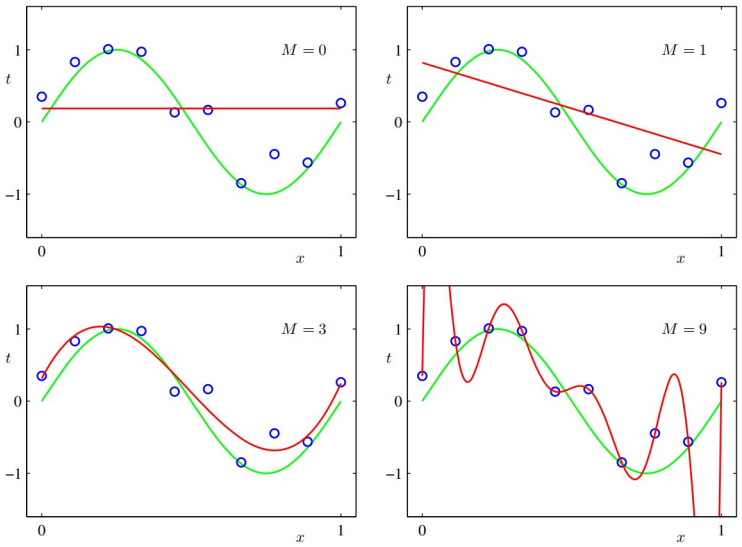


Figure 4.2: Relationship between capacity and order of the polynomial (Bishop, 2006)

Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit.

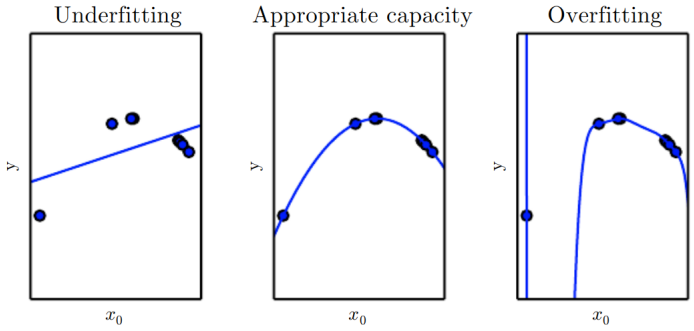


Figure 4.3: Underfitting and Overfitting

Figure 4.3 shows this principle in action. We compare a linear, quadratic and degree-9 predictor attempting to fit a problem where the true underlying function is quadratic. The linear function is unable to capture the curvature in the true underlying problem, so it underfits. The degree-9 predictor is capable of representing the correct function, but it is also capable of representing infinitely

many other functions that pass exactly through the training points, because we have more parameters than training examples. We have little chance of choosing a solution that generalizes well when so many wildly different solutions exist. In this example, the quadratic model is perfectly matched to the true structure of the task, so it generalizes well to new data.

Typically, generalization error has a U-shaped curve as a function of model capacity. This is illustrated in figure 4.4.

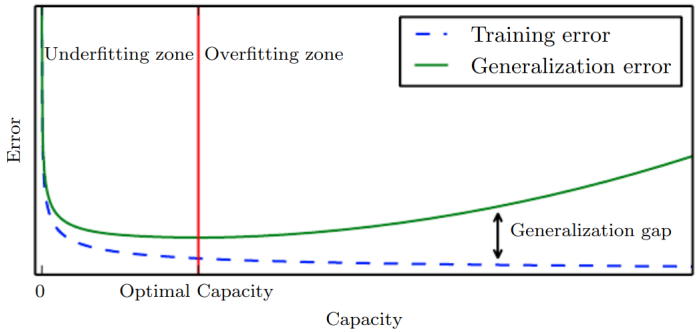


Figure 4.4: Typical relationship between capacity and error (Goodfellow et al., 2016, chap. 5)

4.2 Bias Variance tradeoff

Inability of a model to fit training data is called Bias

A model with low training error and high test error is said to have high Variance.

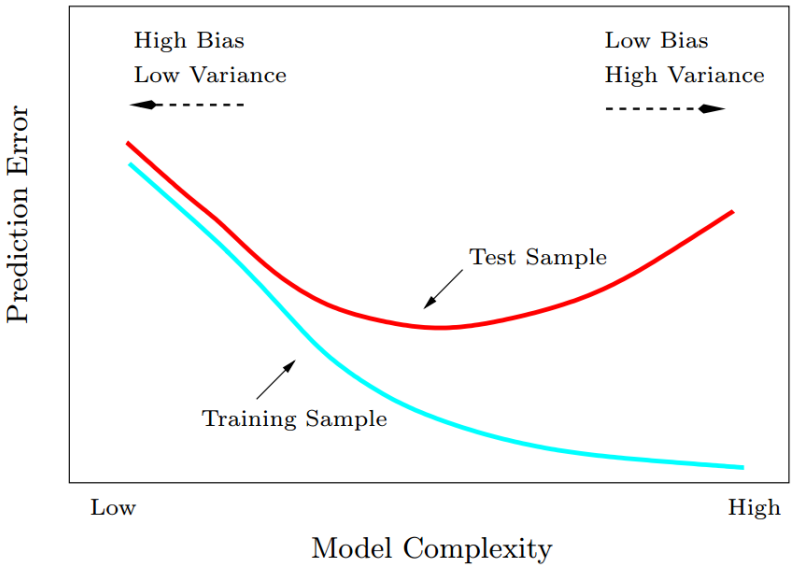


Figure 4.5: Test and training error as a function of model complexity (Hastie et al., 2017)

It is also interesting to examine the behaviour of a given model as the size of the data set is varied, as shown in Figure 4.6. We see that, for a given model complexity, the over-fitting problem become less severe as the size of the data set increases. Another way to say this is that the larger the data set, the more complex (in other words more flexible) the model that we can afford to fit to the data.

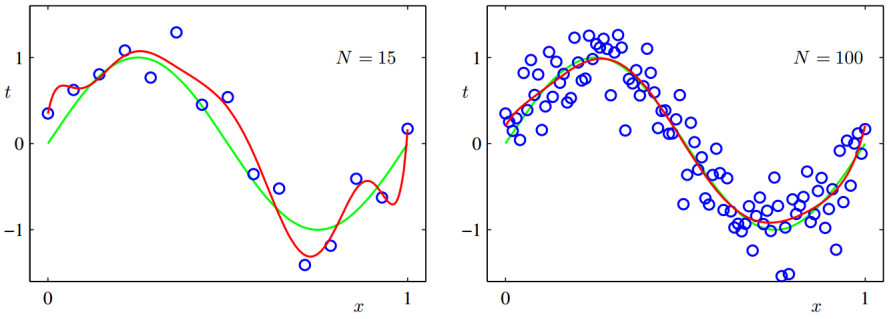


Figure 4.6: Plots of the solutions obtained by minimizing the sum-of-squares error function using the $M = 9$ polynomial for $N = 15$ data points (left plot) and $N = 100$ data points (right plot). We see that increasing the size of the data set reduces the over-fitting problem. (Bishop, 2006, chap. 1.1)

4.3 Regularization

Regularization allows complex models to be trained on data sets of limited size without severe over-fitting, essentially by limiting the effective model complexity.

We see that, as M increases, the magnitude of the coefficients typically gets larger. In particular for the $M = 9$ polynomial, the coefficients have become finely tuned to the data by developing large positive and negative values so that the corresponding polynomial function matches each of the data points exactly, but between data points (particularly near the ends of the range) the function exhibits the large oscillations observed in bottom right sub-figure of Figure 4.2. Intuitively, what is happening is that the more flexible polynomials with larger values of M are becoming increasingly tuned to the random noise on the target values.

	$M = 0$	$M = 1$	$M = 6$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

Figure 4.7: Table of the coefficients w^* for polynomials of various order. Observe how the typical magnitude of the coefficients increases dramatically as the order of the polynomial increases. (Bishop, 2006, chap. 1.1)

One technique that is often used to control the over-fitting phenomenon in such cases is that of regularization, which involves adding a penalty term to the error function in order to discourage the coefficients from reaching large values. The simplest such penalty term takes the form of summation of coefficients raised to a power q, leading to a modified error function of the form:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}\{y_i - \hat{y}_i\}^2 + \lambda\sum_{j=1}^{M}(w_j)^q\,. \tag{4.4}$$

Where N is the number of examples in Training set, M is the number of weights / coefficients of the model and the coefficient λ governs the relative importance of the regularization term compared with the (sum-of-squares) error term. Note that often the coefficient $w0$ is omitted from the regularizer because its inclusion causes the results to depend on the choice of origin for the target variable , or it may be included but with its own regularization coefficient.

The case of $q = 1$ is know as the **lasso** in the statistics literature. It has the property that if λ is sufficiently large, some of the coefficients w_j are driven to zero, leading to a sparse model.

The particular case of a quadratic regularizer is called **ridge regression**. In the context of neural networks, this approach is known as **weight decay**.

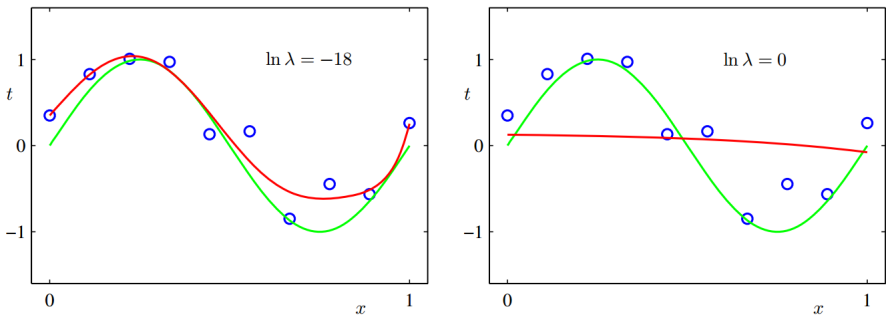


Figure 4.8: Plots of $M = 9$ polynomials fitted to the data set shown in 4.1 using the regularized error function 4.4 for two values of the regularization parameter λ corresponding to $\log \lambda = -18$ and $\log \lambda = 0$. The case of no regularizer, i.e., $\lambda = 0$, corresponding to $\log \lambda = -\infty$, is shown at the bottom right of Figure 4.2. (Bishop, 2006, chap. 1.1)

The impact of the regularization term on the generalization error can be seen by plotting the value of the RMS error (4.4) for both training and test sets against $\log \lambda$, as shown in Figure 4.9. We see that in effect λ now controls the effective complexity of the model and hence determines the degree of over-fitting.

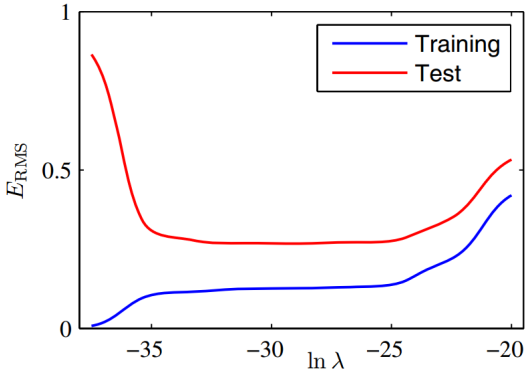


Figure 4.9: Graph of the root-mean-square error (4.4) versus $\log \lambda$ for the $M = 9$ polynomial. (Bishop, 2006, chap. 1.1)

Chapter 5

Regression

5.1 Regression Analysis

It is a Statistical technique that actually explains the change in dependent variable due to movement in other independent variables. To predict both the factors of 1) Amount of change 2) Direction of change as well as the significance of the relationship between the variables, we use Regression Analysis.

Other names of Dependent and independent variable.

Dependent variable = Regressand = response variable

Independent variable = Regressor = predictor variable = Explanatory variable

5.2 Regression Types

Type of regression	Typical use
Simple linear	Predicting a quantitative response variable from a quantitative explanatory variable
Polynomial	Predicting a quantitative response variable from a quantitative explanatory variable, where the relationship is modeled as an nth order polynomial
Multiple linear	Predicting a quantitative response variable from two or more explanatory variables
Multivariate	Predicting more than one response variable from one or more explanatory variables
Logistic	Predicting a categorical response variable from one or more explanatory variables

Figure 5.1: Regression Types

5.3 Linear Regression

To have better understanding of the Machine Learning Algorithm, we present an example of a simple machine learning algorithm: linear regression. We will return to this example repeatedly as we introduce more machine learning concepts that help to understand the algorithm's behavior. As the name implies, linear regression solves a regression problem. In otherwords, the goal is to build a system that can take a vector $\boldsymbol{x} \in \mathbb{R}^n$ as input and predict the value of a scalar $y \in \mathbb{R}$ as its output. The output of linear regression is a linear function of the input. Let $\hat{y}$ be the value that our model predicts y should take on. We define the output to be

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} \tag{5.1}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of **parameters**. Parameters are values that control the behavior of the system. In this case, w_i is the coefficient that we multiply by feature x_i before summing up the contributions from all the features. We can think of $\boldsymbol{w}$ as a set of weights that determine how each feature affects the prediction. If a feature x_i receives a positive weight w_i, then increasing the value of that feature increases the value of our prediction $\hat{y}$. If a feature receives a negative weight, then increasing the value of that feature decreases the value of our prediction. If a feature's weight is large in magnitude, then it has a large effect on the prediction. If a feature's weight is zero, it has no effect on the prediction. We thus have a definition of our task T: to predict y from $\boldsymbol{x}$ by outputting $\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$. Next we need a definition of our performance measure, P.

Suppose that we have a design matrix of m example inputs that we will not use for training, only for evaluating how well the model performs. We also have a vector of regression targets providing the correct value of y for each of these examples. Because this dataset will only be used for evaluation, we call it the test set. We refer to the design matrix of inputs as $\boldsymbol{X}^{(\text{test})}$ and the vector of regression targets as $\boldsymbol{y}^{(test)}$. One way of measuring the performance of the model is to compute the mean squared error of the model on the test set. If $\hat{y}^{(test)}$ gives the predictions of the model on the test set, then the mean squared error is given by

It is worth noting that the term linear regression is often used to refer to a slightly more sophisticated model with one additional parameter — an intercept term b. In this model

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x} + b. \tag{5.2}$$

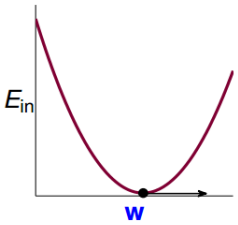
This extension to affine functions means that the plot of the model's predictions still looks like a line, but it need not pass through the origin. Instead of adding the bias parameter b, one can continue to use the model with only weights but augment $\boldsymbol{x}$ with an extra entry that is always set to 1. The weight corresponding to the extra 1 entry plays the role of the bias parameter.

The intercept term b is often called the bias parameter of the affine transformation. This terminology derives from the point of view that the output of the transformation is biased toward being b in the absence of any input.

$$
\begin{aligned}
E_{\text{in}}(\mathbf{w}) &= \frac{1}{N}\sum_{n=1}^{N}(\mathbf{w}^T\mathbf{x}_n - y_n)^2 = \frac{1}{N}\sum_{n=1}^{N}(\mathbf{x}_n^T\mathbf{w} - y_n)^2 \\
&= \frac{1}{N}\left\| \begin{bmatrix} \mathbf{x}_1^T\mathbf{w} - y_1 \\ \mathbf{x}_2^T\mathbf{w} - y_2 \\ \cdots \\ \mathbf{x}_N^T\mathbf{w} - y_N \end{bmatrix} \right\|^2 \\
&= \frac{1}{N}\left\| \begin{bmatrix} -- \mathbf{x}_1^T -- \\ -- \mathbf{x}_2^T -- \\ \cdots \\ -- \mathbf{x}_N^T -- \end{bmatrix} \mathbf{w} - \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ y_N \end{bmatrix} \right\|^2 \\
&= \frac{1}{N}\| \underbrace{\mathrm{X}}_{N\times d+1} \underbrace{\mathbf{w}}_{d+1\times 1} - \underbrace{\mathbf{y}}_{N\times 1} \|^2
\end{aligned}
$$

Figure 5.2: Mean Square Error / Quadratic Loss / L2 Loss function (Abu-Mostafa et al., 2012, chap. 3.2)

$$\min_{\mathbf{w}} E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\|\mathrm{X}\mathbf{w} - \mathbf{y}\|^2$$



- $E_{\text{in}}(\mathbf{w})$: continuous, differentiable, **convex**
- necessary condition of 'best' $\mathbf{w}$

$$\nabla E_{\text{in}}(\mathbf{w}) \equiv \begin{bmatrix} \frac{\partial E_{\text{in}}}{\partial w_0}(\mathbf{w}) \\ \frac{\partial E_{\text{in}}}{\partial w_1}(\mathbf{w}) \\ \cdots \\ \frac{\partial E_{\text{in}}}{\partial w_d}(\mathbf{w}) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \end{bmatrix}$$

—not possible to 'roll down'

task: find $\mathbf{w}_{\text{LIN}}$ such that $\nabla E_{\text{in}}(\mathbf{w}_{\text{LIN}}) = \mathbf{0}$

Figure 5.3: Global minimum value of Mean Square Error (Abu-Mostafa et al., 2012, chap. 3.2)

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\|\mathrm{X}\mathbf{w} - \mathbf{y}\|^2 = \frac{1}{N}\left(\mathbf{w}^T \underbrace{\mathrm{X}^T\mathrm{X}}_{\mathrm{A}}\mathbf{w} - 2\mathbf{w}^T \underbrace{\mathrm{X}^T\mathbf{y}}_{\mathbf{b}} + \underbrace{\mathbf{y}^T\mathbf{y}}_{c}\right)$$

one w only

$$E_{\text{in}}(w) = \frac{1}{N}\left(aw^2 - 2bw + c\right)$$
$$\nabla E_{\text{in}}(w) = \frac{1}{N}\left(2aw - 2b\right)$$

simple! :-)

vector $\mathbf{w}$

$$E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\left(\mathbf{w}^T\mathrm{A}\mathbf{w} - 2\mathbf{w}^T\mathbf{b} + c\right)$$
$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{1}{N}\left(2\mathrm{A}\mathbf{w} - 2\mathbf{b}\right)$$

similar (**derived by definition**)

$$\nabla E_{\text{in}}(\mathbf{w}) = \frac{2}{N}\left(\mathrm{X}^T\mathrm{X}\mathbf{w} - \mathrm{X}^T\mathbf{y}\right)$$

Figure 5.4: Gradient of Mean Square Error function (Abu-Mostafa et al., 2012, chap. 3.2)

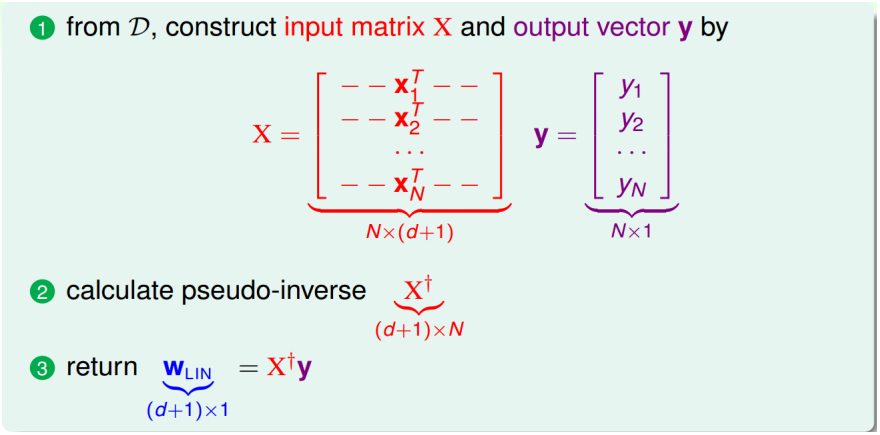


Figure 5.5: Linear Regression work flow (Abu-Mostafa et al., 2012, chap. 3.2)

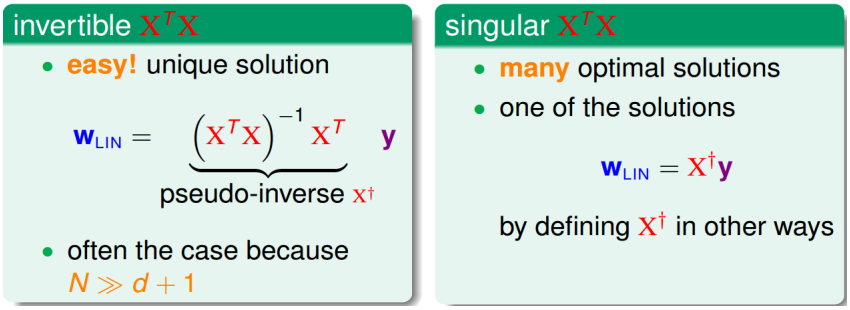


Figure 5.6: Pseudo Inverse Calculation (Abu-Mostafa et al., 2012, chap. 3.2)

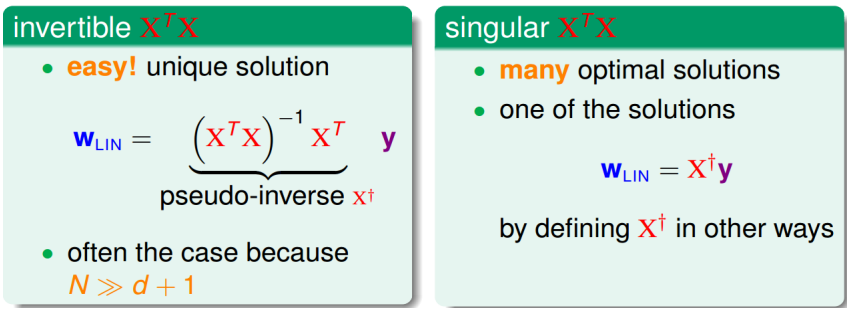


Figure 5.7: Pseudo Inverse Calculation (Abu-Mostafa et al., 2012, chap. 3.2)

5.4 Logistic Regression

Unlike actual regression, logistic regression does not try to predict the value of a numeric variable given a set of inputs. Instead, the output is a probability that the given input point belongs to a certain class. For simplicity, lets assume that we have only two classes(for multiclass problems, you can look at Multinomial Logistic Regression), and the probability in question is $P_+ \Rightarrow$ the probability that a certain data point belongs to the + class. Of course, $P_- = 1 - P_+$. Thus, the output of Logistic Regression always lies in $[0, 1]$.

The central premise of Logistic Regression is the assumption that your input space can be separated into two nice "regions", one for each class, by a linear(read: straight) boundary. If your data points do satisfy this constraint, they are said to be linear-separable. (Note: If your points aren't linearly separable in the original concept space, you could consider converting the feature vectors into a higher dimensional space by adding dimensions of interaction terms, higher degree terms, etc. Such usage of a linear algorithm in a higher dimensional space gives you some benefits of non-linear function learning, since the boundary would be non-linear if plotted back in the original input space.)

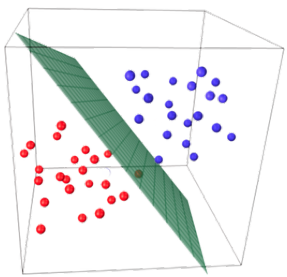


Figure 5.8: Linear boundary separating positive and negative class

But how does Logistic Regression use this linear boundary to quantify the probability of a data point belonging to a certain class?

Assuming two input variables for simplicity(unlike the 3-dimensional figure shown above)- x_1 and x_2, the function corresponding to the boundary will be something like

$$w_0 + w_1 x_1 + w_2 x_2$$

Consider a point (a, b). Plugging the values of x_1 and x_2 into the boundary function, we will get its output $w_0 + w_1 a + w_2 b$. Now depending on the location of (a, b), there are three possibilities to consider-

1. (a, b) lies in the region defined by points of the + class. As a result, $w_0 + w_1 a + w_2 b$ will be positive, lying somewhere in $(0, \infty)$. Mathematically, the higher the magnitude of this value, the greater is the distance between the point and the boundary. Intuitively speaking, the greater is the probability that (a, b) belongs to the + class. Therefore, P_+ will lie in $(0.5, 1]$.

2. (a, b) lies in the region defined by points of the − class. Now, $w_0 + w_1 a + w_2 b$ will be negative, lying in $(-\infty, 0)$. But like in the positive case, higher the absolute value of the function output, greater the probability that (a, b) belongs to the − class. P_+ will now lie in $[0, 0.5)$.

3. (a, b) lies ON the linear boundary. In this case, $w_0 + w_1 a + w_2 b = 0$. This means that the model cannot really say whether (a, b) belongs to the + or − class. As a result, P_+ will be exactly 0.5.

So now we have a function that outputs a value in $(-\infty, \infty)$ given an input data point. But how do we map this to the probability P_+, that goes from $[0, 1]$? The answer, is in the **odds function**.

Let $P(X)$ denote the probability of an event X occurring. In that case, the odds ratio $(OR(X))$ is defined by $\frac{P(X)}{1-P(X)}$, which is essentially the ratio of the probability of the event happening, vs. it not happening. It is clear that probability and odds convey the exact same information. But as $P(X)$ goes from $0 to 1$, $OR(X)$ goes from 0 to ∞.

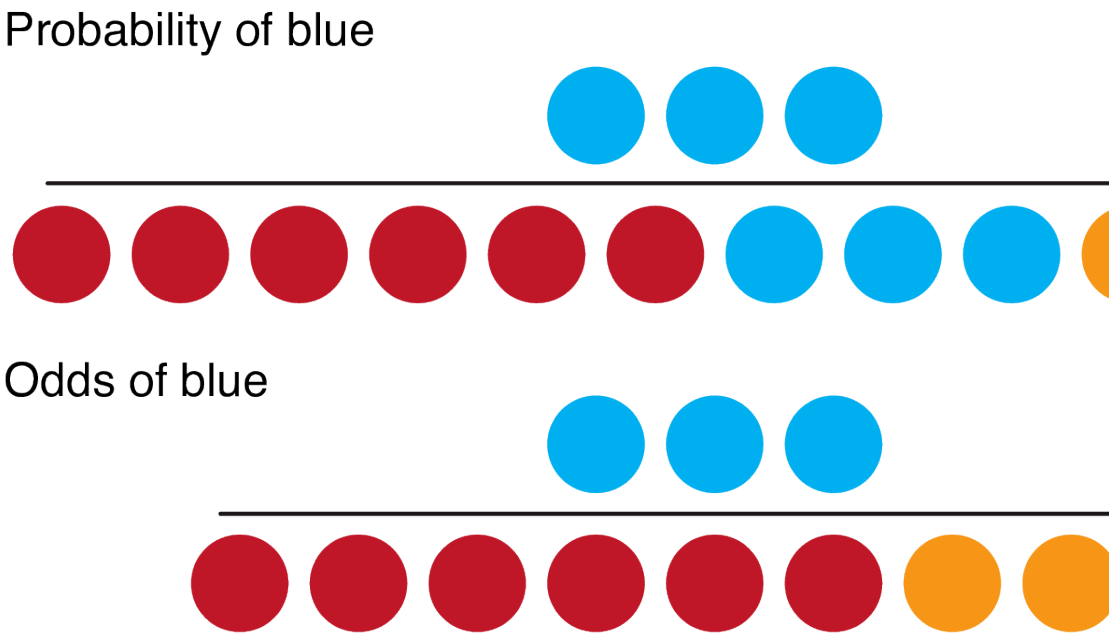


Figure 5.9: Logistic Regression

However, we are still not quite there yet, since our boundary function gives a value from ∞ to ∞. So what we do, is take the logarithm of $OR(X)$, called the **log-odds** function. Mathematically, as $OR(X)$ goes from 0 to ∞, $log(OR(X))$ goes from ∞ to ∞

So we finally have a way to interpret the result of plugging in the attributes of an input into the boundary function. The boundary function actually defines the log-odds of the + class, in our model. So essentially, in our two-dimensional example, given a point (a, b), this is what Logistic regression would do-

1. Compute the boundary function (alternatively, the log-odds function) value, $w_0 + w_1x_1 + w_2x_2$. Lets call this value t for short.
2. Compute the Odds Ratio, by doing $OR_+ = e^t$. (Since t is the logarithm of OR_+).
3. Knowing OR_+, it would compute P_+ using the simple mathematical relation

$$P_+ = \frac{OR_+}{1 + OR_+}$$

.

In fact, once you know t from step 1, you can combine steps 2 and 3 to give you

$$P_+ = \frac{e^t}{1+e^t}$$

The RHS of the above equation is called the **logistic function**. Hence the name given to this model of learning

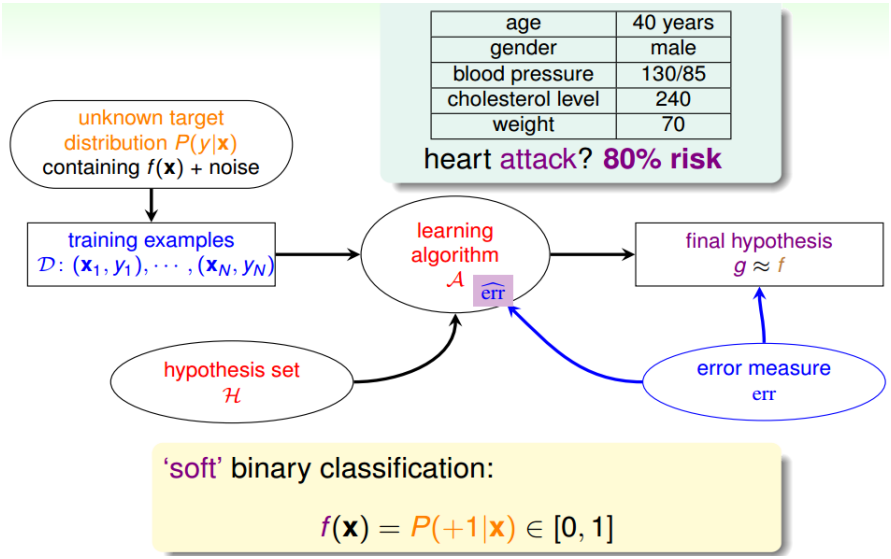


Figure 5.10: Logistic Regression problem (Abu-Mostafa et al., 2012, chap. 3.3)

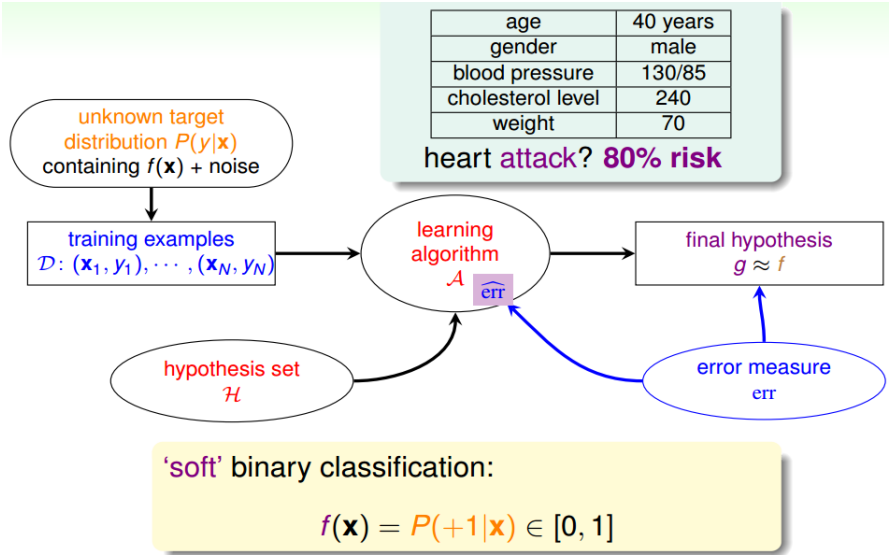


Figure 5.11: Logistic Regression problem (Abu-Mostafa et al., 2012, chap. 3.3)

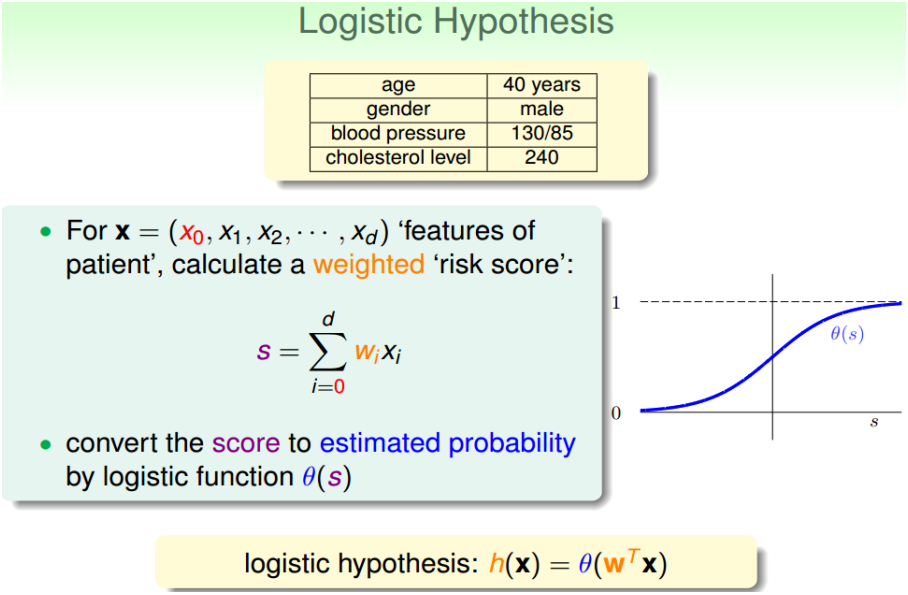


Figure 5.12: Hypothesis in Logistic Regression (Abu-Mostafa et al., 2012, chap. 3.3)

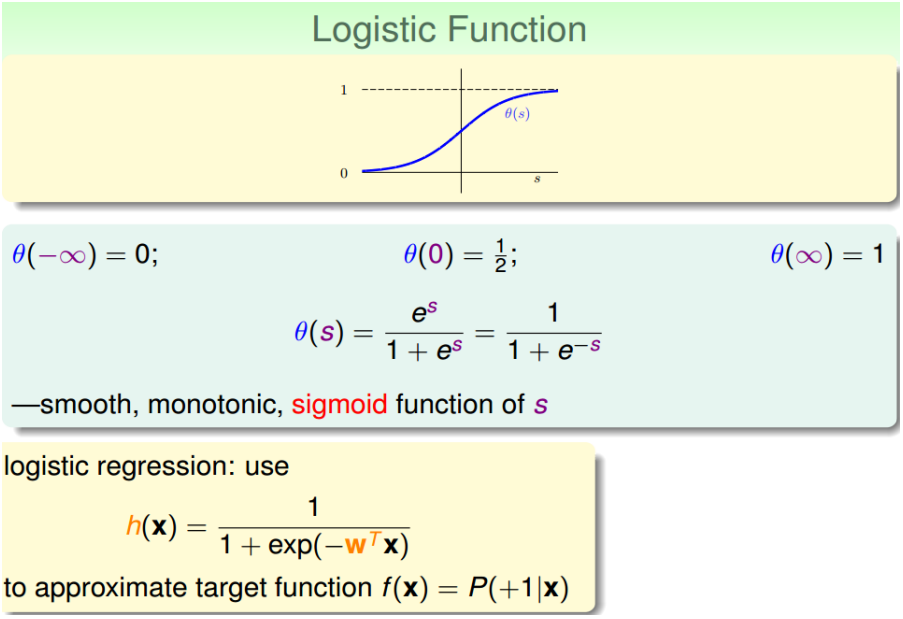


Figure 5.13: Logistic Function (Abu-Mostafa et al., 2012, chap. 3.3)

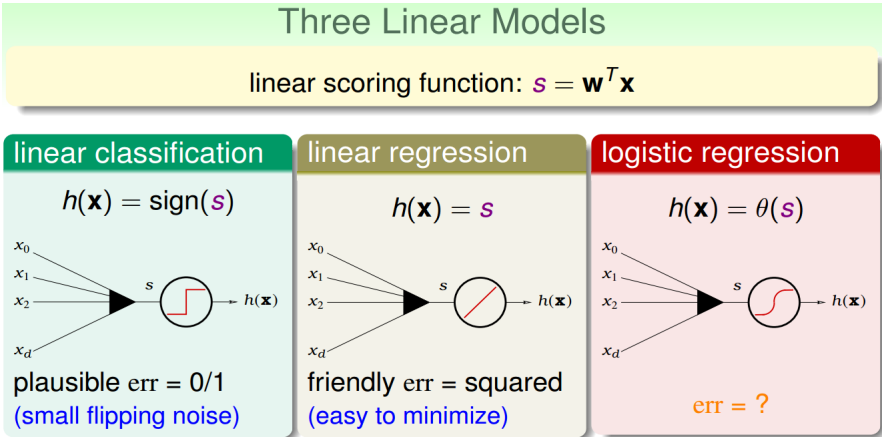


Figure 5.14: Linear models (Abu-Mostafa et al., 2012, chap. 3.3)

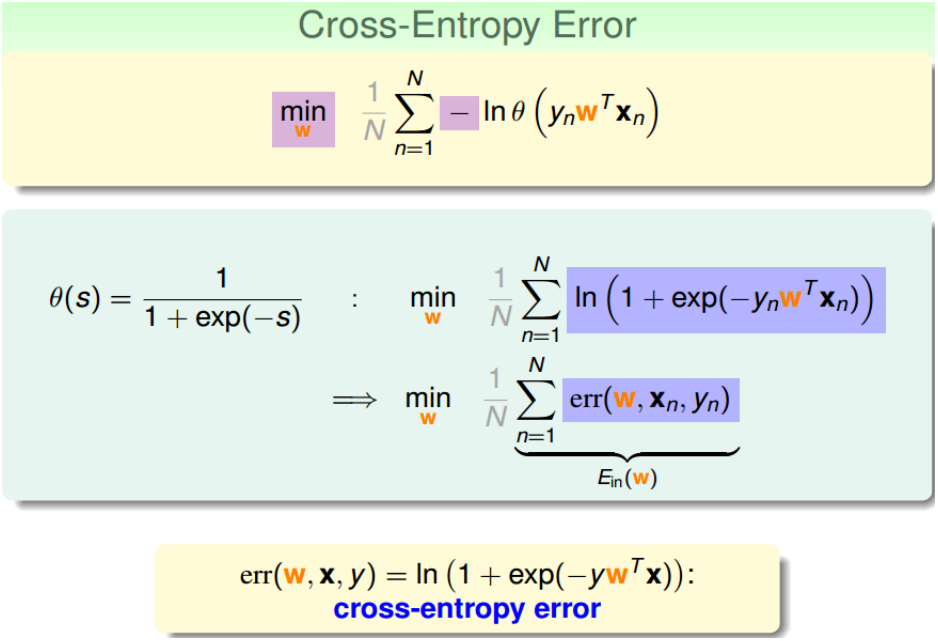


Figure 5.15: Cross Entropy Error (Abu-Mostafa et al., 2012, chap. 3.3)

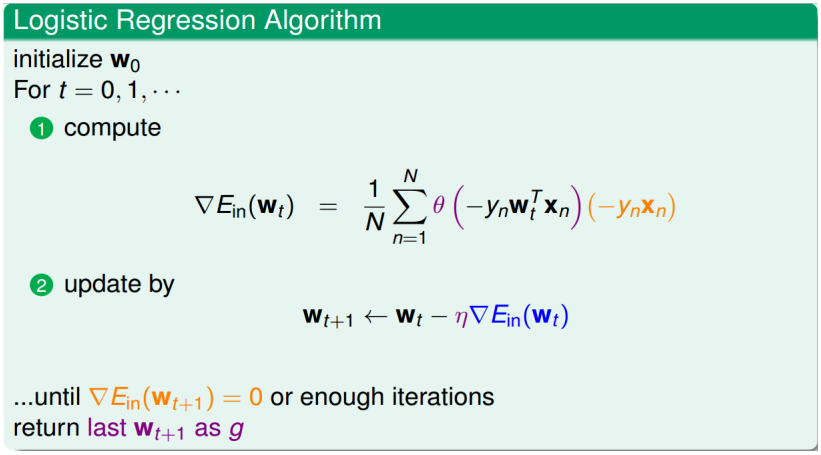


Figure 5.16: Logistic Regression work flow (Abu-Mostafa et al., 2012, chap. 3.3)

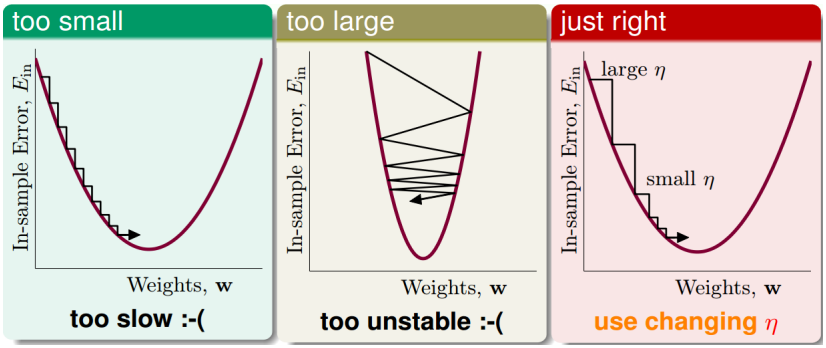


Figure 5.17: Choice of learning rate (Abu-Mostafa et al., 2012, chap. 3.3)

5.5 Ridge Regression

When using linear regression, especially with small datasets, the chance of overfitting your model is high. This is because although there is low bias, with a smaller training set, the variability in how your model performs on testing sets will increase. In order to counter overfitting, we can use various regularization techniques used by Ridge and LASSO regression. They pretty much work by introducing some bias in the model in order to lower the variability of how the model performs on test sets. This is especially useful when the coefficients in your linear regression model are far apart. This is where Ridge Regression comes in.

Ridge Regression uses something called $L2$ Regularization. Regularization is used to reduce overfitting. $L2$ Regularization reduces model complexity and helps bring the weights in our model closer to zero, in essence, decreasing variance and shifting us more left on the bias-variance curve. If you're confused, take a look here. Linear Regression is prone to overfitting and high variance, what we can do to counter that is to introduce bias. In return for adding bias, our model has decreased variance.

A traditional linear regression model minimizes the sum of squared residuals, whereas in ridge regression, the sum of squared residuals plus a λ times the sum of the coefficients squared is minimized. You can think of this as a penalty term. If $\lambda = 0$, our ridge cost function equals our linear regression cost function.

$$Cost = \sum_{i=1}^{M} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{d} w_j \times x_{ij} \right)^2 + \lambda \sum_{j=1}^{d} (w_j)^2 \qquad (5.3)$$

5.6 LASSO Regression

In ridge regression, we improved on linear regression by adding a penalty term to help make our coefficients more stable. Ridge regression utilized $L2$ regularization while LASSO regression uses $L1$ regularization. LASSO regression adds feature selection in the mix, which means it can make certain coefficients or weights zero, thereby removing them and selecting the others. To see how this works, let's look at the LASSO cost function.

$$Cost = \sum_{i=1}^{M} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{M} \left(y_i - \sum_{j=0}^{d} w_j \times x_{ij} \right)^2 + \lambda \sum_{j=0}^{d} (|w_j|) \tag{5.4}$$

The difference in the penalty terms in ridge and LASSO regression shows how $L1$ and $L2$ regularization differ. Like ridge regression, a λ value of 0 yields the linear regression cost function. By only looking at the magnitude of coefficients rather than the square, LASSO regression permits certain weights to reach 0 whereas in ridge regression, weights can only approach 0. This type of regularization is extremely useful when we have a lot of features in our data. $L1$ regularization uses feature selection to help reduce model complexity.

5.7 Performance Metrics for Regression

Following are the performance metrics used for evaluating a regression model:

1. Mean Absolute Error (MAE)

$$MAE = \frac{1}{n} \sum_{i=1}^{n} |(y_i - \hat{y}_i)| \tag{5.5}$$

   where y_i is the actual expected output and $\hat{y}_i$ is the model's prediction. It is the simplest evaluation metric for a regression scenario and is not much popular compared to the following metrics.

   Say, $y_i = [5, 10, 15, 20]$ and $\hat{y}_i = [4.8, 10.6, 14.3, 20.1]$

   Thus, $MAE = \frac{1}{4} * (|5 - 4.8| + |10 - 10.6| + |15 - 14.3| + |20 - 20.1|) = 0.4$

2. Mean Squared Error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.6}$$

Here, the error term is squared and thus more sensitive to outliers as compared to Mean Absolute Error (MAE).

Thus, $MSE = \frac{1}{4} * (|5-4.8|^2 + |10-10.6|^2 + |15-14.3|^2 + |20-20.1|^2) = 0.225$

3. Root Mean Squared Error (RMSE)

$$RMSE = \sqrt{\frac{1}{n} * \sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{5.7}$$

Since MSE includes squared error terms, we take the square root of the MSE, which gives rise to Root Mean Squared Error (RMSE).

Thus, $RMSE = (0.225)^{0.5} = 0.474$

4. R-Squared

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{5.8}$$

R-squared is calculated by dividing the sum of squares of residuals (SS_{res}) from the regression model by the total sum of squares (SS_{tot}) of errors from the average model and then subtract it from 1.

R-squared is also known as the **Coefficient of Determination**. It explains the degree to which the input variables explain the variation of the output / predicted variable.

A R-squared value of 0.81, tells that the input variables explains 81% of the variation in the output variable. The higher the R squared, the more variation is explained by the input variables and better is the model. There exists a limitation in this metric, which is solved by the Adjusted R-squared.

5. Adjusted R-squared

$$\text{Adjusted } R^2 = 1 - \frac{\left(1 - R^2\right)(n-1)}{n-p-1} \tag{5.9}$$

Here, n - total sample size (number of rows) and p - number of predictors (number of columns)

The limitation of R-squared is that it will either stay the same or increases with the addition of more variables, even if they do not have any relationship with the output variables. To overcome this limitation, Adjusted R-square comes into the picture as it penalizes you for adding the variables which do not improve your existing model.

Hence, if you are building Linear regression on multiple variables, it is always suggested that you use Adjusted R-squared to judge the goodness of the model.

If there exists only one input variable, R-square and Adjusted R squared are same.

Chapter 6

Classification

6.1 Support Vector Machine

A support vector machine (or SVM) is an algorithm that works as follows. It uses a nonlinear mapping to transform the original training data into a higher dimension. Within this new dimension, it searches for the linear optimal separating hyperplane (that is, a "decision boundary" separating the tuples of one class from another). With an appropriate non linear mapping to a sufficiently high dimension, data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors).

Margin is defined as the distance between the separating hyperplane (decision boundary) and the training samples that are closest to this hyperplane, which are the so-called support vectors. The hyperplane with the larger margin is more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase), the SVM searches for the hyperplane with the largest margin, that is, the maximum marginal hyperplane (MMH). This is illustrated in the Figures: 6.2, 6.1.

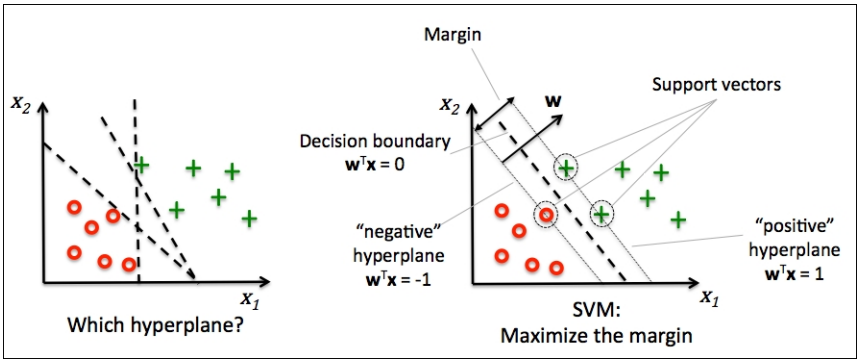


Figure 6.1: Support Vector Machine

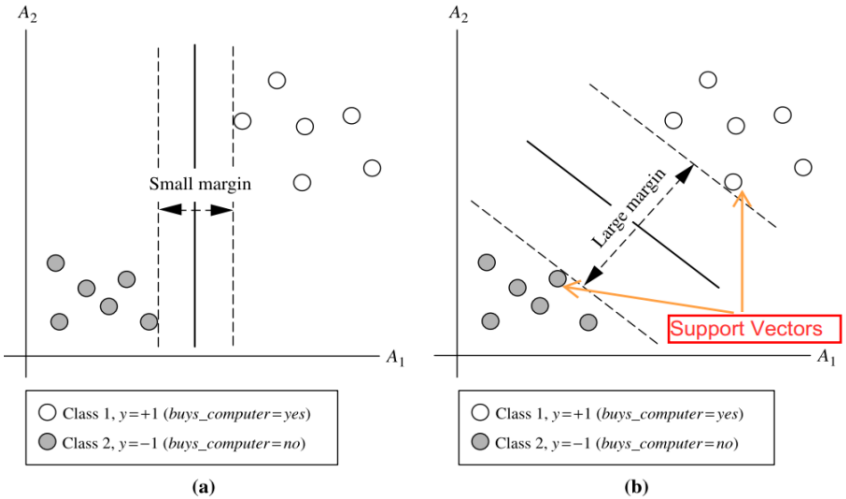


Figure 6.2: Two possible separating hyperplanes and their associated margins. Which one is better? The one with the larger margin (b) is called MMH.

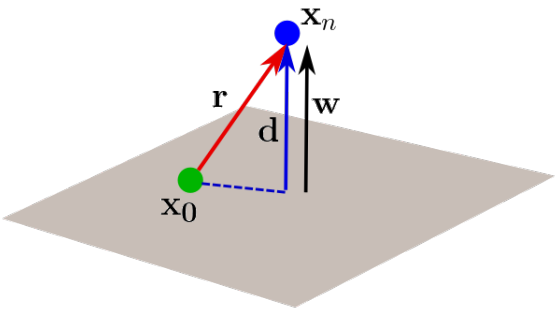


Figure 6.3: Margin in SVM

Here, x_n is one of the points closest to the hyperplane and it forms an orthogonal vector d that stems from it. As we can see, vector d is in the same direction as w. Besides, any point x_0 that lies on the hyperplane will form a vector r with x_n. Given these definitions, we can clearly see that d is the projection of r on w given by

$$\mathbf{d} = \frac{\mathbf{w}}{\|\mathbf{w}\|}\mathbf{r} = \frac{\mathbf{w}^\top (\mathbf{x}_n - \mathbf{x}_0)}{\|\mathbf{w}\|} = \frac{\mathbf{w}^\top \mathbf{x}_n - \mathbf{w}^\top \mathbf{x}_0}{\|\mathbf{w}\|} \tag{6.1}$$

Let's add b and $-b$ to the numerator in this equation. The magnitude of d will give us the distance of x to the hyperplane and is defined by:

$$d = \frac{\left|\mathbf{w}^\top \mathbf{x}_n + b - \left(\mathbf{w}^\top \mathbf{x}_0 + b\right)\right|}{\|\mathbf{w}\|} = \frac{\left|\mathbf{w}^\top \mathbf{x}_n + b\right|}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|} \tag{6.2}$$

Note that for x_0 that lies on the hyperplane, its corresponding prediction y_0 is zero, and that's why the expression inside of the parenthesis vanishes. Therefore, in order to find the optimal margin, what we want to do is to maximize d. With these considerations, we can finally state the optimization problem for the model:

$$d = \max\left(\frac{1}{\|\mathbf{w}\|}\right) \quad \text{subject to} \quad \min_n \left|\mathbf{w}^\top \mathbf{x}_n + b\right| = 1 \tag{6.3}$$

Well, from Calculus we know that in order to find a maximum value, we need to use derivatives. Given this scenario, it will be more convenient for us to work with expressions that do not involve the use of reciprocal terms. So how could we make the d equation easier for us to optimize? Yes, you guessed it right! Invert the reciprocal term inside of the parentheses and then turn it into a minimization problem. Let's do that:

$$d = \min\left(\frac{1}{2}\|\mathbf{w}\|^2\right) = \min\left(\frac{1}{2}\mathbf{w}^\top\mathbf{w}\right) \quad \text{subject to } y_n\left(\mathbf{w}^\top\mathbf{x}_n\right) \geq 1 \tag{6.4}$$

So now you may be asking, why did we raise w to the power 2 and then divide it by 2? This is just a mathematical trick we are applying to make things easier for us when computing derivatives. Basically, the main reason is that the derivative for this expression is simply w. However and most importantly, this is going to help us define a quadratic programming problem that we will be able to solve using popular optimization libraries.

So far so good. Now consider what if we had data as shown in image below?

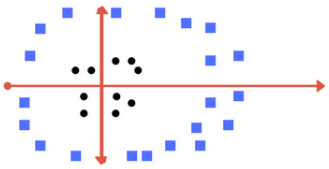


Figure 6.4: Non linearly separable problem in x-y plane

Clearly, there is no line that can separate the two classes in this x-y plane. So what do we do? We apply transformation and add one more dimension as we call it z-axis. In general terms, kernel functions allow us to transform non-linearly separable spaces to linearly-separable ones.

Let's assume value of points on z plane, w = x^2 + y^2. In this case we can manipulate it as distance of point from z-origin. Now if we plot in z-axis, a clear separation is visible and a line can be drawn.

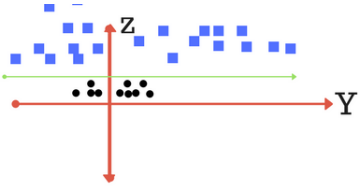


Figure 6.5: linearly separable problem in y-z plane

When we transform back this line to original plane, it maps to circular boundary as shown in image E. Thankfully, you don't have to guess / derive the transformation every time for your data set. The sklearn library's SVM implementation provides it inbuilt.

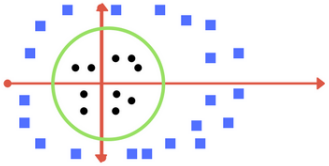


Figure 6.6: Decision boundary in x-y plane

It only works well when the categories can be separated completely. That's why this default version is also called **hard-margin SVM**. But as you can imagine, it's very unlikely for us to find datasets that are completely separable in real life. Therefore, we will next discuss a variation that allows us to extend SVMs to scenarios where some points could be misclassified, also known as **soft-margin SVM**.

6.1.1 Soft-Margin SVM

As we said earlier, hard-margin SVMs have limited use in real life applications. Here we will show that by making a slight change to the original dual problem we can extend SVMs to scenarios where some points could be misclassified. Let's illustrate our idea by means of a graph:

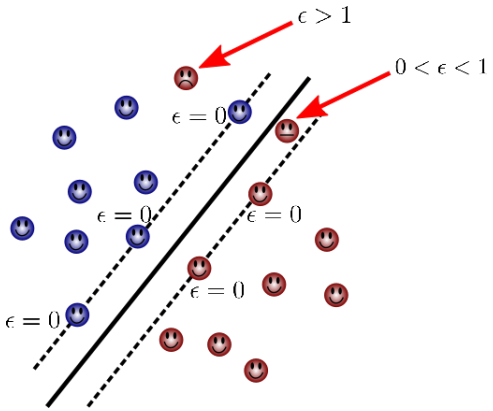


Figure 6.7: Misclassification scenarios for the soft-margin SVM

Here, we have defined a new tolerance variable ϵ. Points that lie on the margin will have $\epsilon = 0$. On the other hand, the farther we are from the correct boundary, the larger the value of ϵ will be. Ultimately, for points that are on the wrong side of the hyperplane, we expect ϵ to be greather than 1. Given this scenario, our objective becomes:

$$\min \left(\frac{1}{2} \mathbf{w}^\top \mathbf{w} + C \sum_{n=1}^{N} \epsilon_n \right) \quad \text{subject to } y_n \left(\mathbf{w}^\top \mathbf{x}_n \right) \geq 1 - \epsilon_n \tag{6.5}$$

As we can see, the second term here is a product of a new variable C we define and the sum of all epsilons. The Regularization parameter (often termed as C parameter in python's sklearn library) tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly. Conversely, a very small value of C will cause the optimizer to look for a larger-margin separating hyperplane, even if that hyperplane mis-classifies more points.

6.1.2 Advantages and Dis-advantages

- They are highly accurate, owing to their ability to model complex nonlinear decision boundaries.
- They are much less prone to overfitting than other methods.
- The support vectors found also provide a compact description of the learned model.
- SVMs can be used for prediction as well as classification.
- The training time of even the fastest SVMs can be extremely slow.

6.2 Classification and Regression Tree (CART)

6.2.1 Terminologies associated with decision tree

A decision tree has three main components :

- Root Node : The top most node is called Root Node. It implies the best predictor (independent variable).
- Decision / Internal Node : The nodes in which predictors (independent variables) are tested and each branch represents an outcome of the test
- Leaf / Terminal Node : It holds a class label (category) - Yes or No (Final Classification Outcome).

6.2.2 Types of Decision Tree

- Regression Tree is used when the dependent variable is continuous. The value obtained by leaf nodes in the training data is the mean response of observation falling in that region. Thus, if an unseen data observation falls in that region, its prediction is made with the mean value. This means that even if the dependent variable in training data was continuous, it will only take discrete values in the test set. A regression tree follows a top-down greedy approach. Classification Tree
- Classification tree is used when the dependent variable is categorical. The value obtained by leaf nodes in the training data is the mode response of observation falling in that region It follows a top-down greedy approach.

Together they are called as CART(Classification And Regression Tree)

We aim to build a decision tree where given a new record of chest pain, good blood circulation, and blocked arteries we should be able to tell if that person has heart disease or not. At the start, all our samples are in the root node. We will have to decide on which of the feature the root node should be divided first.

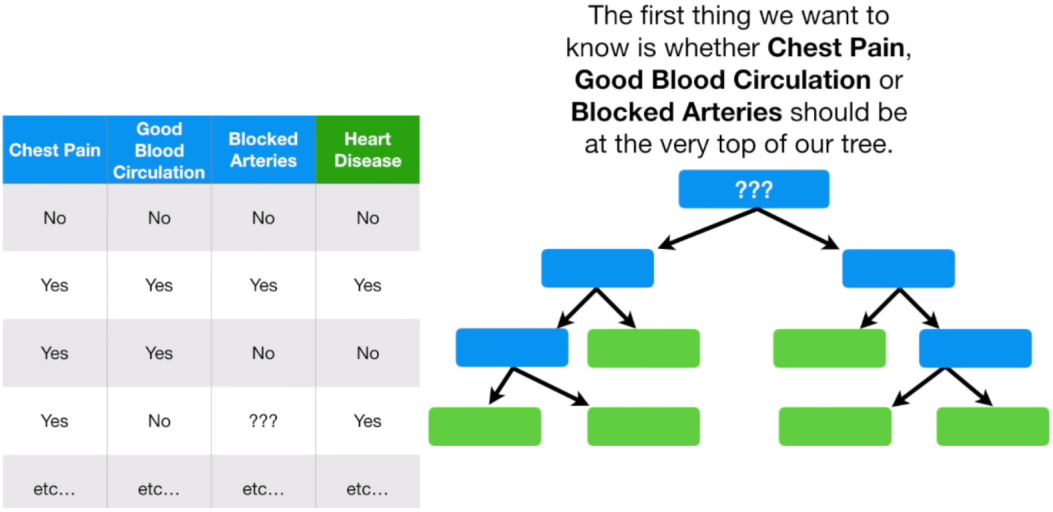


Figure 6.8: Data

Taking all three splits at one place in the below image. We can observe that, it is not a great split on any of the feature alone for heart disease yes or no which means that one of this can be a root node but its not a full tree, we will have to split again down the tree in hope of better split.

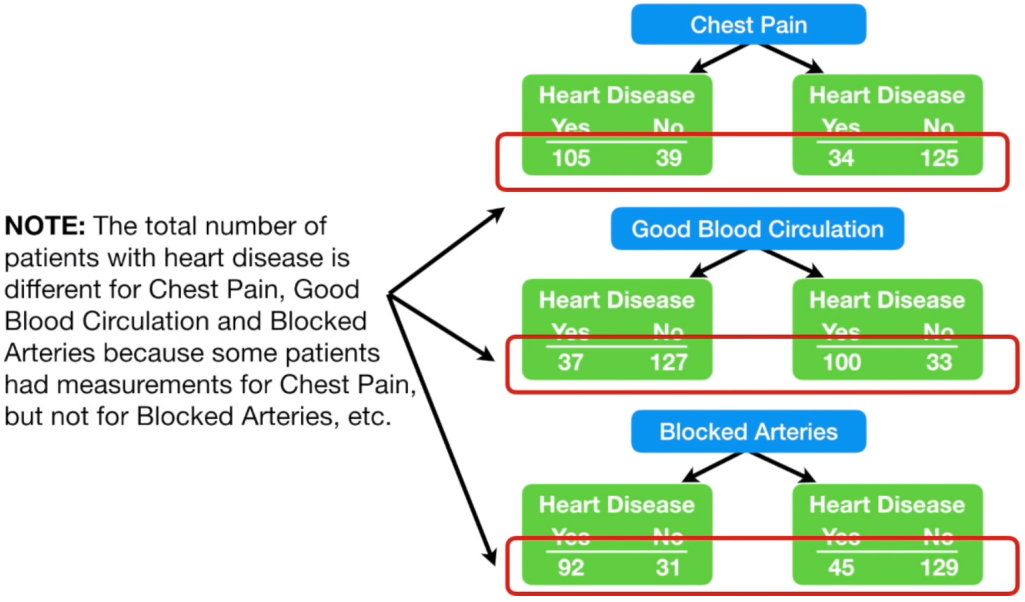
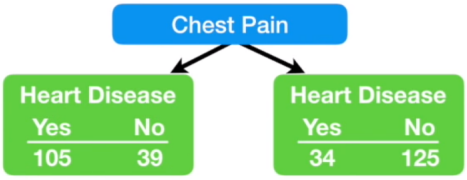


Figure 6.9: Possible splits

To decide on which one feature should the root node be split, we need to calculate the Gini impurity for all the leaf nodes as shown below. After calculating for leaf nodes, we take its weighted average to get Gini impurity about the parent node.

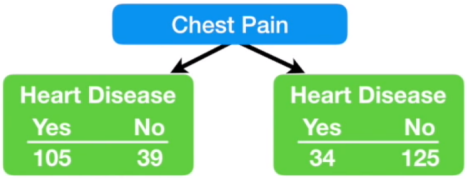


For this leaf, the Gini impurity = 1 - (the probability of "yes")^2 - (the probability of "no")^2

$$= 1 - \left(\frac{105}{105 + 39}\right)^2 - \left(\frac{39}{105 + 39}\right)^2$$

$$= 0.395$$

Figure 6.10: Gini index for left child of Chest pain



Gini impurity = 0.395 0.336

Gini impurity for Chest Pain = weighted average of Gini impurities for the leaf nodes

$$= \left(\frac{144}{144 + 159}\right) 0.395 + \left(\frac{159}{144 + 159}\right) 0.336$$

$$= 0.364$$

Figure 6.11: Gini index for Chest pain

We do this for all three features and select the one with the least Gini impurity as it is splitting the dataset in the best way out of three. Hence we choose good

blood circulation as the root node.

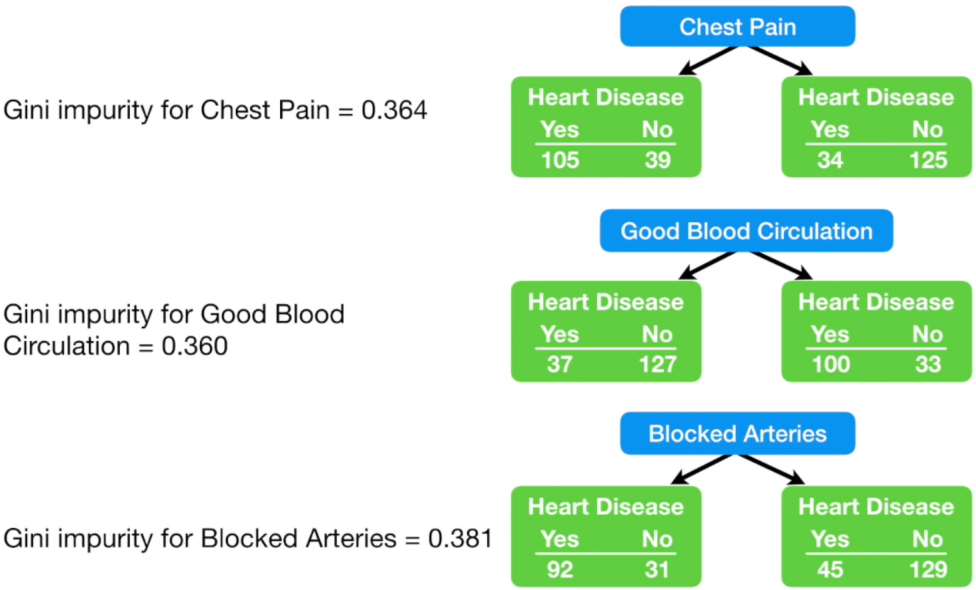


Figure 6.12: Gini indices for all features

We do the same for a child node of Good blood circulation now. In the below image we will split the left child with a total 164 sample on basis of blocked arteries as its gini impurity is lesser than chest pain(we calculate gini index again with the same formula as above, just a smaller subset of the sample — 164 in this case).

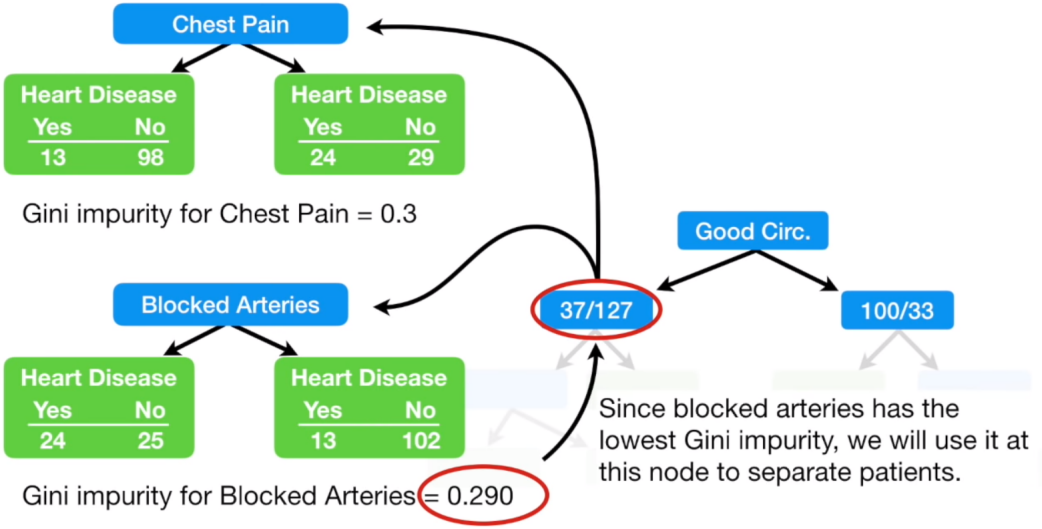


Figure 6.13: Gini indices for child chest pain and Blocked artery

One thing to note in the below image that, when we try to split the right child of blocked arteries on basis of chest pain, the gini index is 0.29 but the gini impurity of the right child of the blocked tree itself, it is 0.20. This means that splitting this node any further is not improving impurity. so this will be a leaf node.

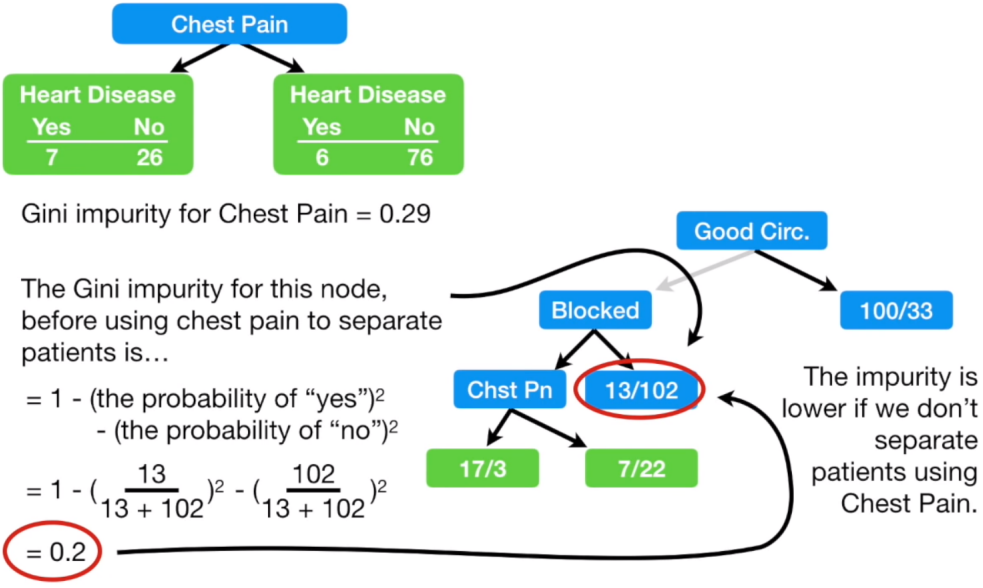


Figure 6.14: No improvement in Gini index of right child of Blocked artery

We repeat the same process for building right sub tree of root node.

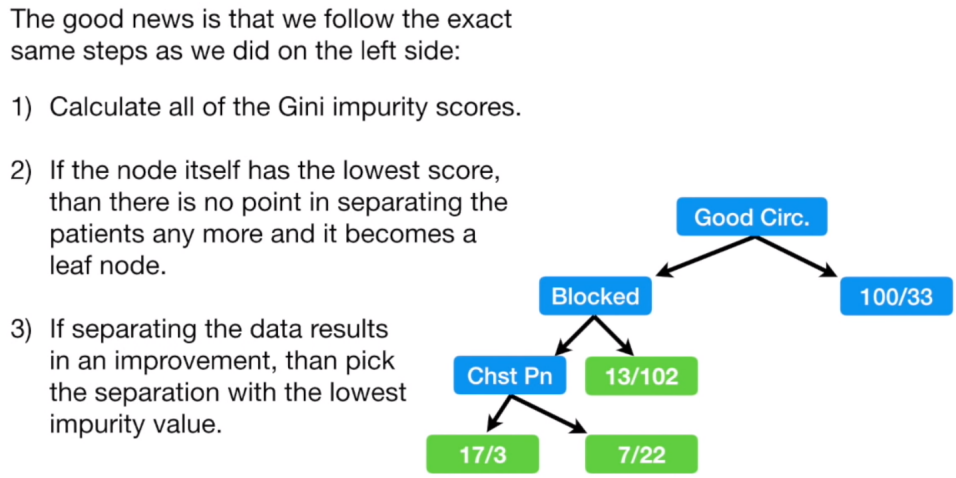


Figure 6.15: Steps for building Decision Tree

The final tree is as follows. This looks a good enough fit for our training data.

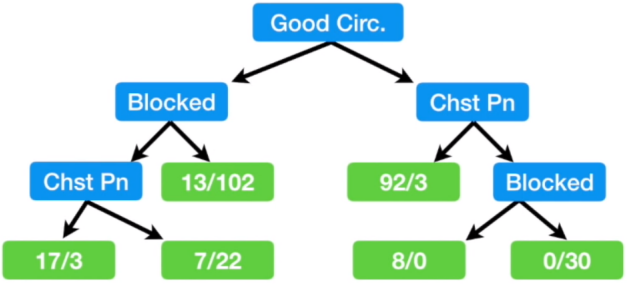


Figure 6.16: Complete Decision Tree

This is how we create a tree from data. What should we do if we have a column with numerical values? It is simple, order them in ascending order. Calculate the mean on every two consecutive numbers. make a split on basis of that and calculate gini impurity using the same method. We choose the split with the least gini impurity as always.

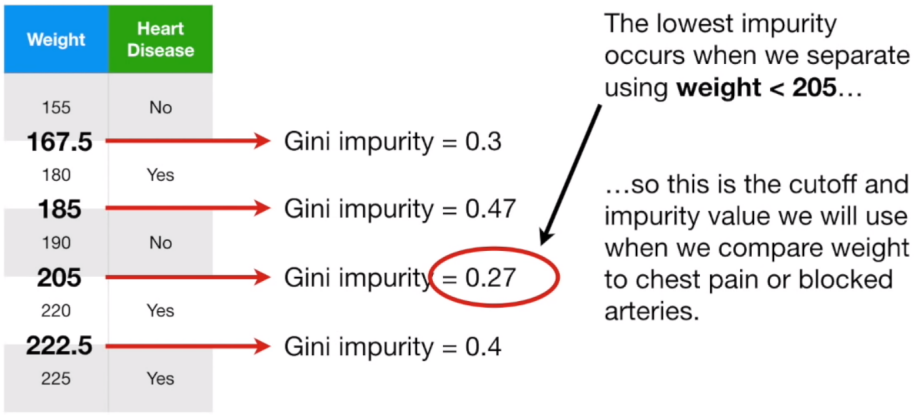


Figure 6.17: Split point if the feature is numerical type

If we have ranked the numerical column in the dataset, we split on every rating and calculate gini impurity for each split and select the one with the least gini impurity.

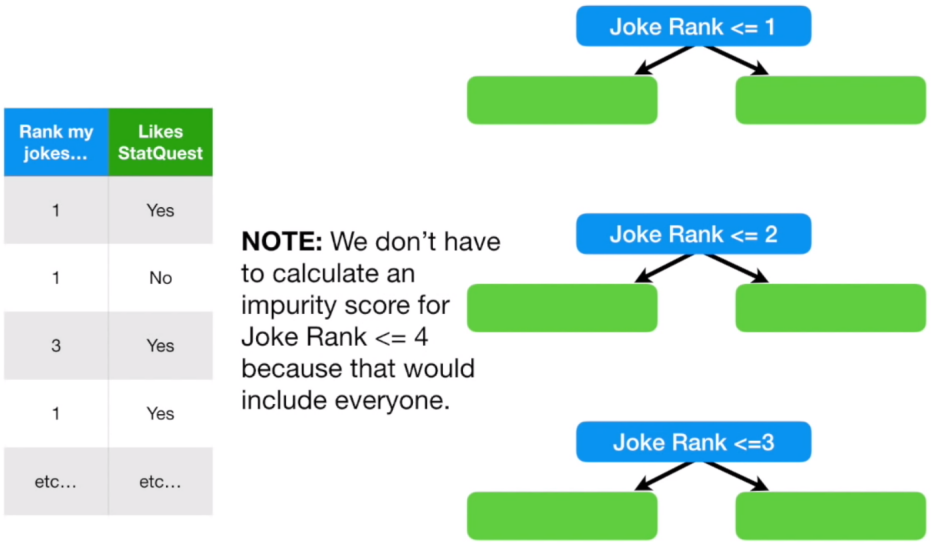


Figure 6.18: Split point if the feature is ordinal type

If we have categorical choices in our dataset then the split and gini impurity calculation needs to be done on every possible combination of choices

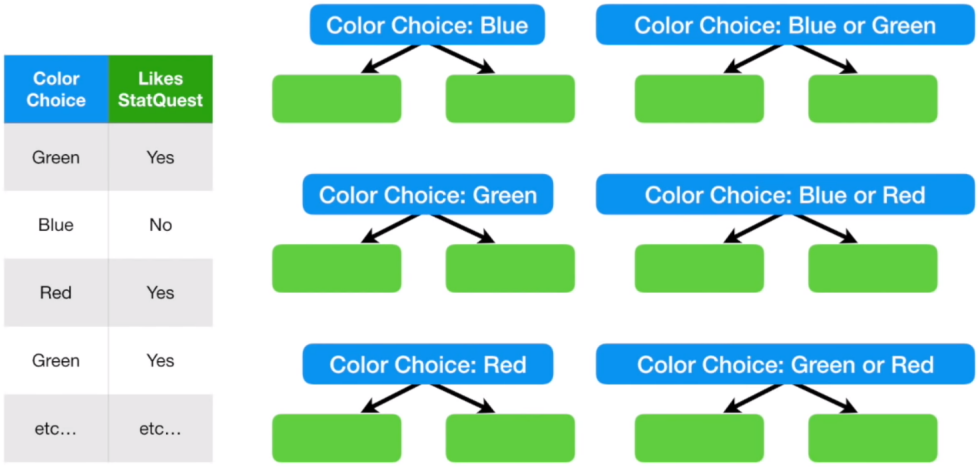


Figure 6.19: Split point if the feature is categorical type

6.2.3 Other attribute selection procedures

- Entropy is a measure of disorder or uncertainty and the goal of machine learning models and Data Scientists in general is to reduce uncertainty.

$$E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{6.6}$$

If we had a total of 100 data points in our dataset with 30 belonging to the positive class and 70 belonging to the negative class then 'P+' would be 3/10 and 'P-' would be 7/10. Pretty straightforward.

$$-\frac{3}{10} \times \log_2\left(\frac{3}{10}\right) - \frac{7}{10} \times \log_2\left(\frac{7}{10}\right) \approx 0.88$$

Figure 6.20: Entropy calculation

	Temperature	Outlook	Humidity	Windy	Played?
0	Mild	Sunny	80	No	Yes
1	Hot	Sunny	75	Yes	No
2	Hot	Overcast	77	No	Yes
3	Cool	Rain	70	No	Yes
4	Cool	Overcast	72	Yes	Yes
5	Mild	Sunny	77	No	No
6	Cool	Sunny	70	No	Yes
7	Mild	Rain	69	No	Yes
8	Mild	Sunny	65	Yes	Yes
9	Mild	Overcast	77	Yes	Yes
10	Hot	Overcast	74	No	Yes
11	Mild	Rain	77	Yes	No
12	Cool	Rain	73	Yes	No
13	Mild	Rain	78	No	Yes

Figure 6.21: Entropy calculation

Yes	No
10	4

Entropy(PlayTennis)

= Entropy(10,4)
=Entropy (0.71, 0.23)
= -(0.71$\log_2$ 0.71)- (0.23$\log_2$ 0.23)
= 0.84

Figure 6.22: Entropy calculation

		Yes	No	Total
Feature 2: Outlook	Sunny	3	2	5
	Overcast	4	0	4
	Rainy	3	2	5

E(PlayTennis, Outlook)

= P(Sunny)*E(3,2) + P(Overcast)*E(4,0) + P(Rainy)*E(3,2)

=(5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971

=0.693

Figure 6.23: Entropy calculation

Information gain is the decrease in entropy. Information gain computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values. ID3 (Iterative Dichotomiser) decision tree algorithm uses information gain.

$$\text{Info(D)} = -\sum_{i=1}^{m} p_i \log_2 p_i \tag{6.7}$$

Where, P_i is the probability that an arbitrary tuple in D belongs to class C_i.

$$\begin{aligned} \text{Info}_\text{A}(\text{D}) &= \textstyle\sum_{j=1}^{V} \frac{|Dj|}{|D|} \quad \text{X Info (D}_\text{j}) \\ \text{Gain(A)} &= \text{Info(D)} - \text{Info}_\text{A}(\text{D}) \end{aligned} \tag{6.8}$$

Where,

$Info(D)$ is the average amount of information needed to identify the class label of a tuple in D.
$\frac{|Dj|}{|D|}$ acts as the weight of the jth partition. $Info_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A.

The attribute A with the highest information gain, Gain(A), is chosen as the splitting attribute at node N

$$IG(Y,X) = E(Y) - E(Y|X)$$

Figure 6.24: Information Gain calculation

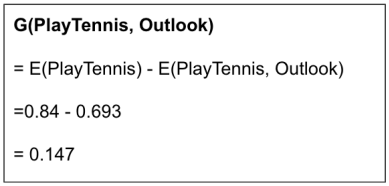


Figure 6.25: Information Gain calculation

Choose attribute with the largest Information Gain as the Root Node

6.2.4 Gain Ratio

Information gain is biased for the attribute with many outcomes. It means it prefers the attribute with a large number of distinct values. For instance, consider an attribute with a unique identifier such as customer_ID has zero info(D) because of pure partition. This maximizes the information gain and creates useless partitioning.

C4.5, an improvement of ID3, uses an extension to information gain known as the gain ratio. Gain ratio handles the issue of bias by normalizing the information gain using Split Info.

$$\text{SplitInfo }_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2 \left(\frac{|D_j|}{|D|} \right) \tag{6.9}$$

Where,
$\frac{|Dj|}{|D|}$ acts as the weight of the jth partition.
v is the number of discrete values in attribute A.

The gain ratio can be defined as

$$\text{GainRatio}(A) = \frac{\text{Gain}(A)}{\text{SplitInfo}_A(D)} \tag{6.10}$$

Advantages :

- Decision tree is easy to interpret.
- Decision Tree works even if there is nonlinear relationships between variables. It does not require linearity assumption.
- Decision Tree is not sensitive to outliers.

Disadvantages :

- Decision tree model generally overfits. It means it does not perform well on validation sample.
- It assumes all independent variables interact each other, It is generally not the case every time.

6.2.5 Avoid Overfitting in Decision Trees

Overfitting is one of the key challenges in a tree-based algorithm. If no limit is set, it will give 100% fitting, because, in the worst-case scenario, it will end up making a leaf node for each observation. Hence we need to take some precautions to avoid overfitting. It is mostly done in two ways:

- Pre Pruning
- Post Pruning

6.2.6 Pre pruning

Parameters play an important role in tree modeling. Overfitting can be avoided by using various parameters that are used to define a tree.

- Minimum samples for a node split
  - Defines the minimum number of observations that are required in a node to be considered for splitting. (this ensures above mentioned worst-case scenario).
  - A higher value of this parameter prevents a model from learning relations which might be highly specific to the particular sample selected for a tree.
  - Too high values can lead to under-fitting hence, it should be tuned properly using cross-validation.
- Minimum samples for a leaf node
  - Defines the minimum observations required in a leaf. (Again this also prevents worst-case scenarios)

– Generally, lower values should be chosen for imbalanced class problems as the regions in which the minority class will be in majority will be of small size.

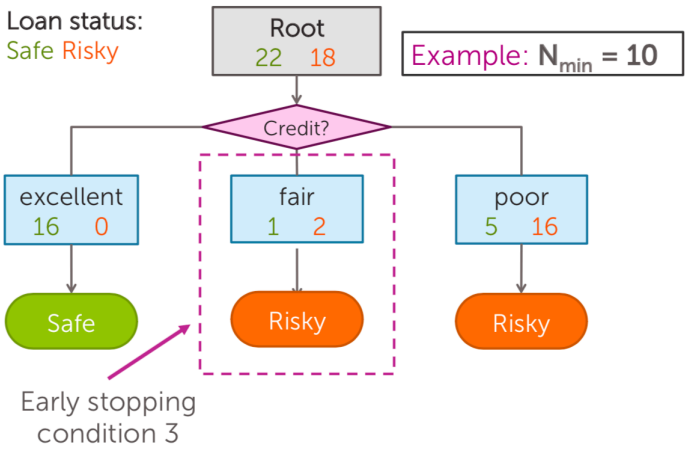


Figure 6.26: Minimum samples at a node constraint

- Maximum depth of the tree (vertical depth)
  - Used to control over-fitting as higher depth will allow the model to learn relations very specific to a particular sample.
  - Should be tuned properly using Cross-validation as too little height can cause underfitting.

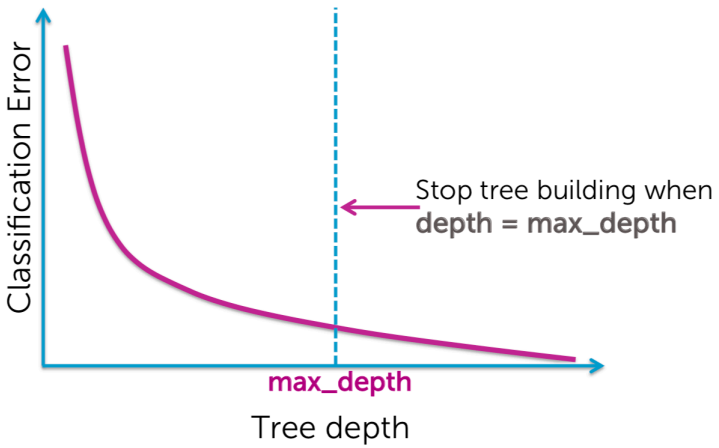


Figure 6.27: Maximum depth of a tree

- Maximum number of leaf nodes
  - The maximum number of leaf nodes or leaves in a tree.
  - Can be defined in place of max_depth. Since binary trees are created, a depth of n would produce a maximum of 2^n leaves.
- Maximum features to consider for a split
  - The number of features to consider while searching for the best split. These will be randomly selected.
  - As a thumb-rule, the square root of the total number of features works great but we should check up to $30 - 40\%$ of the total number of features.
  - Higher values can lead to over-fitting but depend on case to case.

6.2.7 Post Pruning

The cost function that is to be optimized in case of Decision tree is

$$\text{Total cost} = \text{measure of fit} + \text{measure of complexity}$$
$$C(T) = E(T) + \lambda \cdot L(T)$$

Where $L(T)$ is the number of leaf nodes, $E(T)$ is the mis-classification error of tree T and λ is a Regularization parameter.

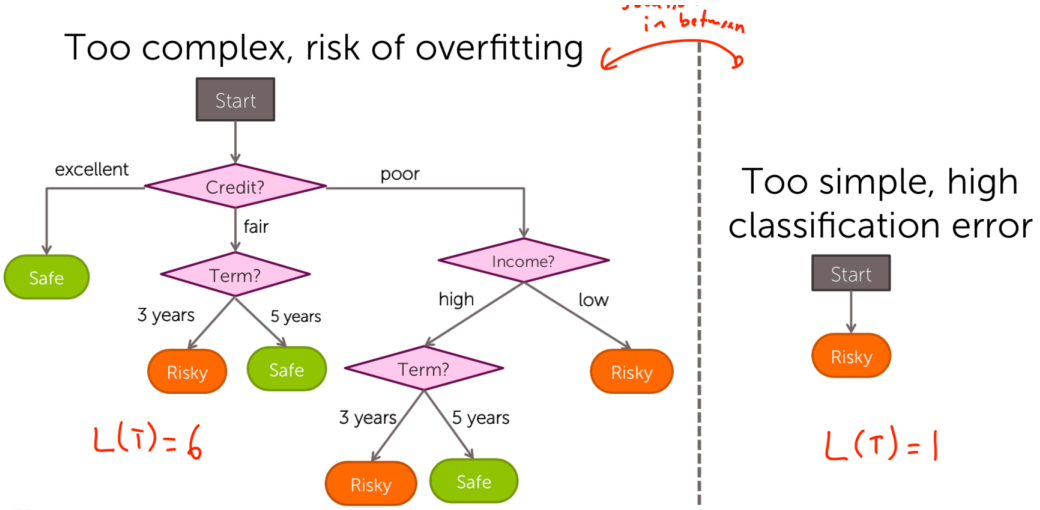


Figure 6.28: Cost function of a decision tree

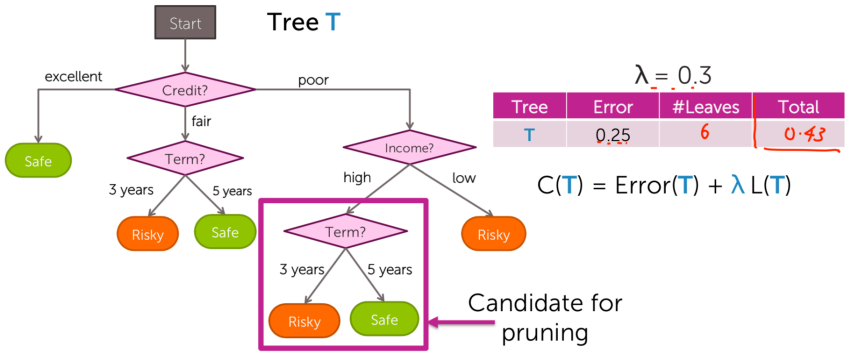


Figure 6.29: Tree pruning

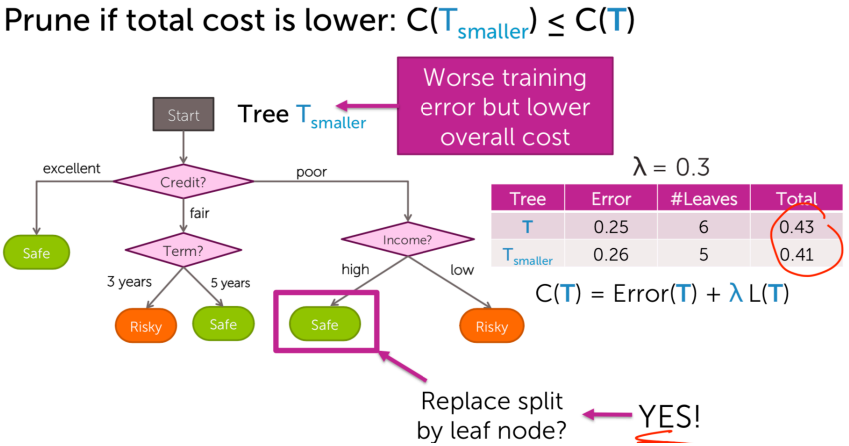


Figure 6.30: Tree pruning

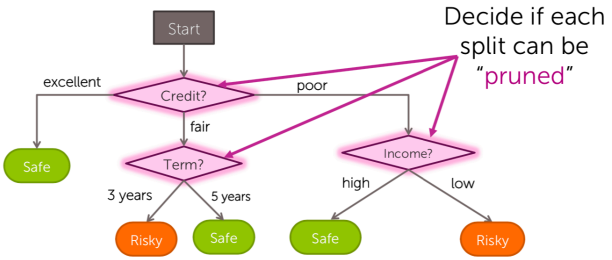


Figure 6.31: Tree pruning

6.3 Classifier performance evaluation

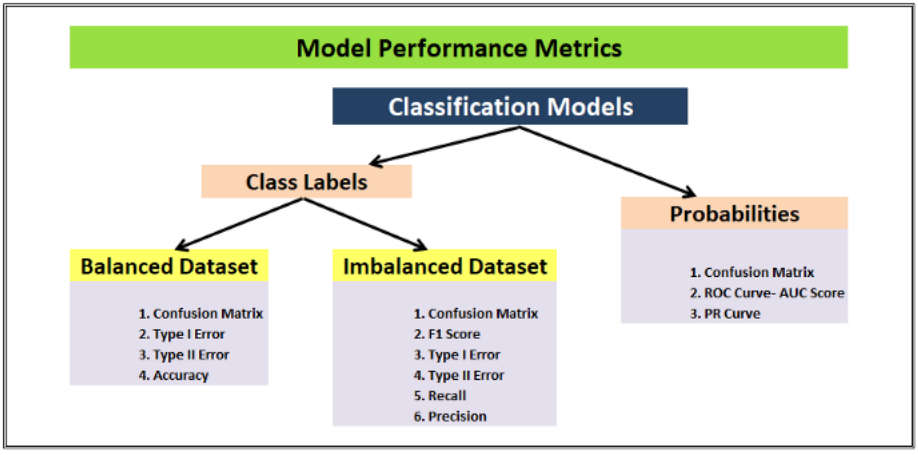


Figure 6.32: Classifier performance metrics

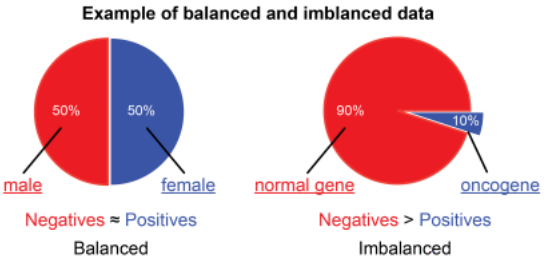


Figure 6.33: Balanced Vs Unbalanced dataset

Following is the list of 10 metrics which are used for performance evaluation of Classifiers:

1. Confusion Matrix: It is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known.

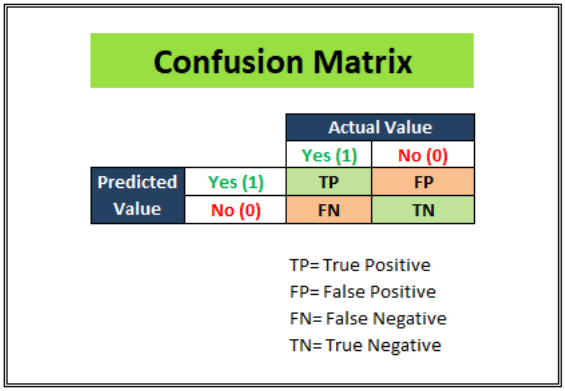


Figure 6.34: Confusion Matrix

2. Type I Error:

   A type 1 error is also known as a false positive and occurs when a classification model incorrectly predicts a true outcome for an originally false observation.

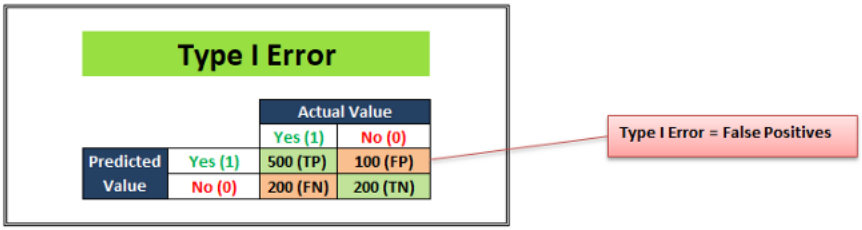


Figure 6.35: Type I Error

3. Type II Error:

   A type II error is also known as a false negative and occurs when a classification model incorrectly predicts a false outcome for an originally true observation.

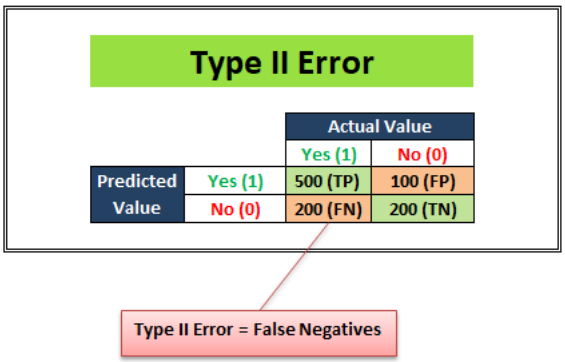


Figure 6.36: Type II Error

4. Accuracy:

   It is the proximity of measurement results to the true value. It tell us how accurate our classification model is able to predict the class labels given in the problem statement.

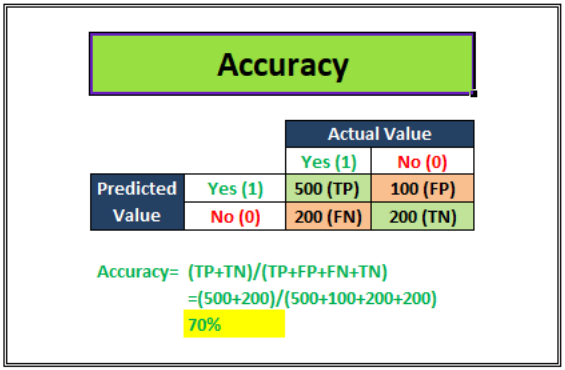


Figure 6.37: Accuracy

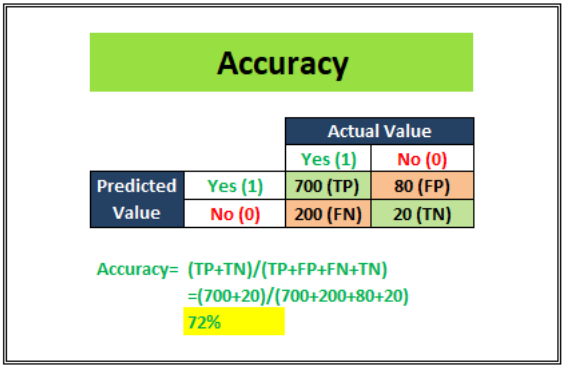


Figure 6.38: Accuracy

5. Recall or True Positive Rate or Sensitivity attempts to answer the following question: What proportion of actual positives was identified correctly?

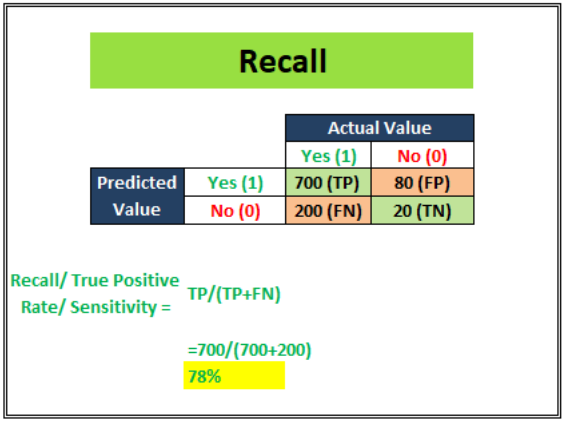


Figure 6.39: Recall

6. Precision attempts to answer the following question: What proportion of positive identifications was actually correct?

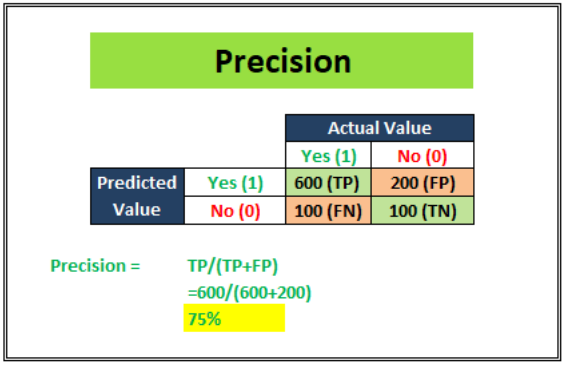


Figure 6.40: Precision

7. Specificity also called the true negative rate measures the proportion of actual negatives that are correctly identified as such.

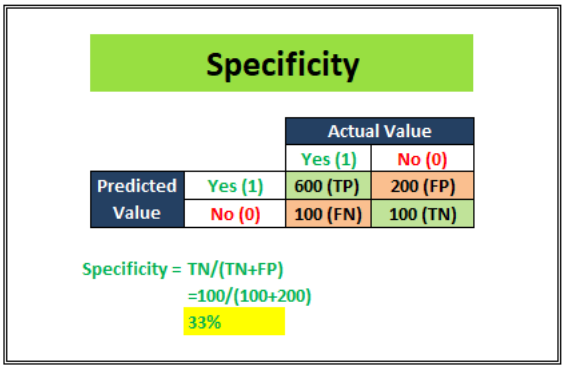


Figure 6.41: Specificity

8. F1 Score

   In a statistical analysis of binary classification, the F1 score (also F-score or F-measure) is a measure of a test's accuracy. It considers both the precision p and the recall r of the test to compute the score

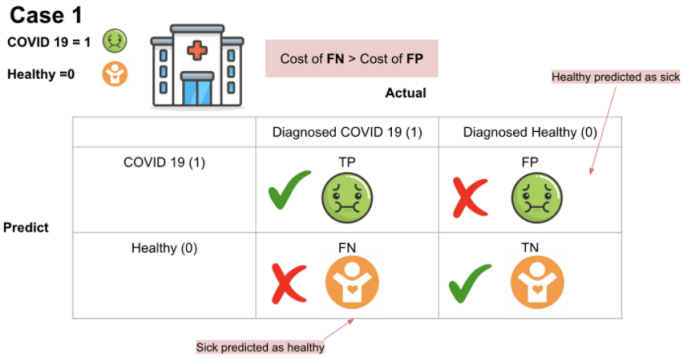


Figure 6.42: TypeII is more costly

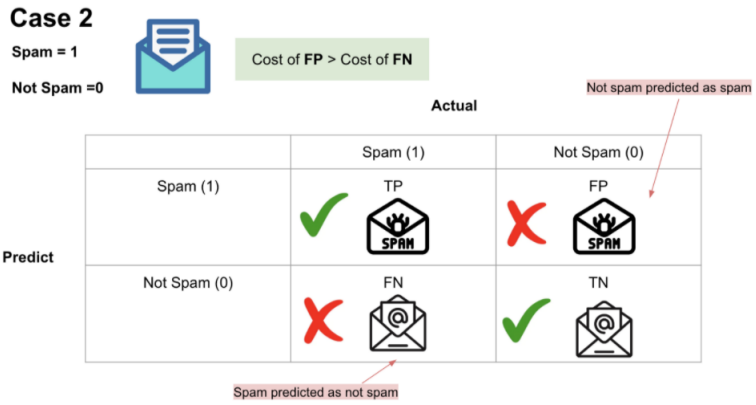


Figure 6.43: TypeI is more costly

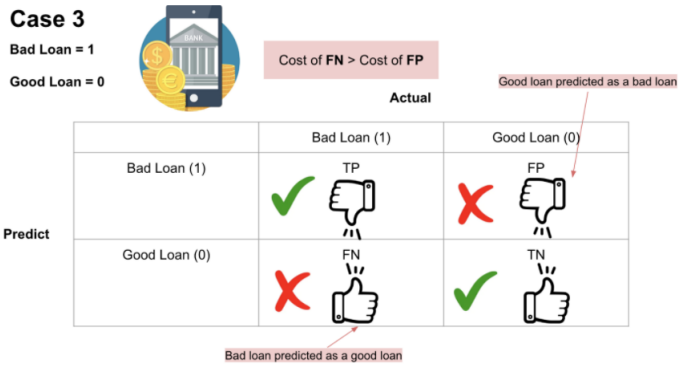


Figure 6.44: TypeII is more costly

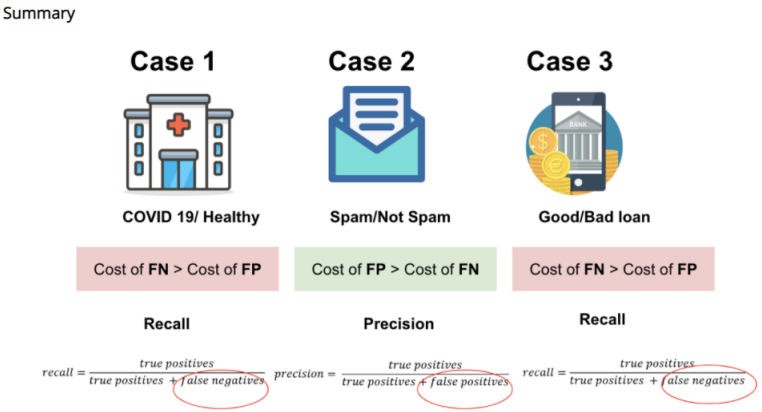


Figure 6.45: Precision Vs Recall

The traditional F-measure or balanced F-score (**F_1 score**) is the harmonic mean of precision and recall:

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} = 2 \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}} = \frac{\mathrm{tp}}{\mathrm{tp} + \frac{1}{2}(\mathrm{fp} + \mathrm{fn})}.$$

F_β [edit]

A more general F score, F_β, that uses a positive real factor β, where β is chosen such that recall is considered β times as important as precision, is:

$$F_\beta = (1 + \beta^2) \cdot \frac{\mathrm{precision} \cdot \mathrm{recall}}{(\beta^2 \cdot \mathrm{precision}) + \mathrm{recall}}.$$

In terms of Type I and type II errors this becomes:

$$F_\beta = \frac{(1 + \beta^2) \cdot \mathrm{true\ positive}}{(1 + \beta^2) \cdot \mathrm{true\ positive} + \beta^2 \cdot \mathrm{false\ negative} + \mathrm{false\ positive}}.$$

Two commonly used values for β are 2, which weighs recall higher than precision, and 0.5, which weighs recall lower than precision.

Figure 6.46: F1 Score

9. ROC Curve- AUC Score:

   Area under the Curve (AUC), Receiver Operating Characteristics curve (ROC) This is one of the most important metrics used for gauging the model performance and is widely popular among the data scientists.

   To plot a ROC curve, we have to plot (1-Specificity) i.e. False Positive Rate on x-axis and Sensitivity i.e. True Positive Rate on the y-axis. ROC (Receiver Operating Characteristic) Curve tells us about how good the model can distinguish between two things (e.g If a patient is obese or not). Better models can accurately distinguish between the two. Whereas, a poor model will have difficulties in distinguishing between the two.

Let's start understanding this with an example. We have a classification model that gives probability values ranging between 0–1 to predict the probability of a person being obese or not. Probability score near 0 indicates a very low probability that the person under consideration is obese whereas probability values near 1 indicate a very high probability of a person being obese. Now, by default if we consider a threshold of 0.5 then all the people assigned probabilities 0.5 will be classified as "Not Obese" and people assigned probabilities >0.5 will be classified as "Obese". But, we can vary this threshold. What if I make it 0.3 or 0.9. Let's see what happens.

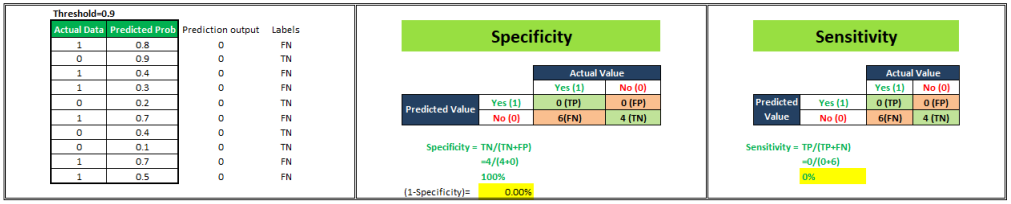


Figure 6.47: ROC calculations

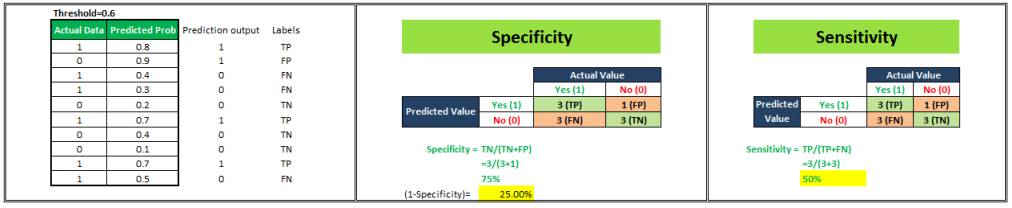


Figure 6.48: ROC calculations

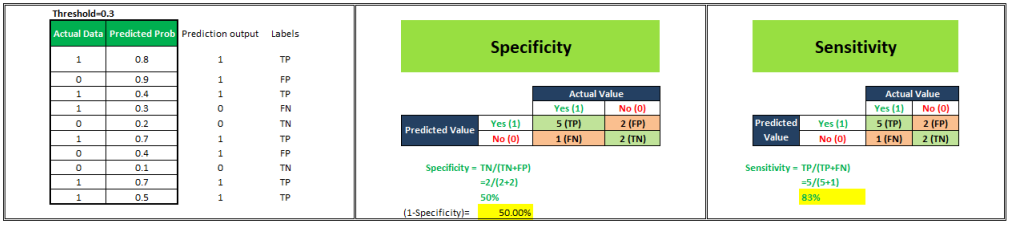


Figure 6.49: ROC calculations

Threshold=0

Actual Data	Predicted Prob	Prediction output	Labels
1	0.8	1	TP
0	0.9	1	FP
1	0.4	1	TP
1	0.3	1	TP
0	0.2	1	FP
1	0.7	1	TP
0	0.4	1	FP
0	0.1	1	FP
1	0.7	1	TP
1	0.5	1	TP

Specificity

		Actual Value	
		Yes (1)	No (0)
Predicted Value	Yes (1)	6 (TP)	4 (FP)
	No (0)	0 (FN)	0 (TN)

Specificity = TN/(TN+FP)
=0/(0+4)
0%
(1-Specificity)= 100.00%

Sensitivity

		Actual Value	
		Yes (1)	No (0)
Predicted Value	Yes (1)	6 (TP)	4 (FP)
	No (0)	0 (FN)	0 (TN)

Sensitivity = TP/(TP+FN)
=6/(6+0)
100%

Figure 6.50: ROC calculations

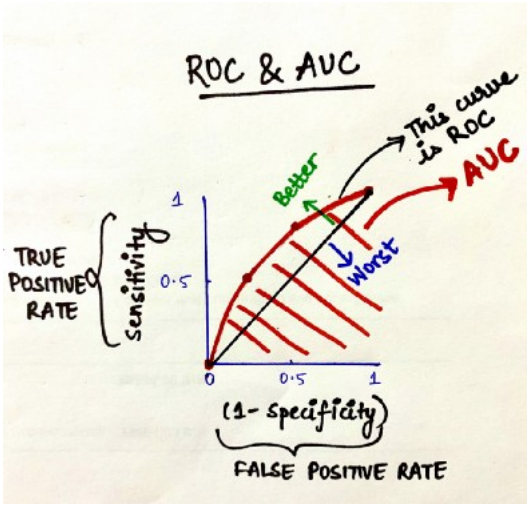


Figure 6.51: ROC curve

The area under the ROC curve is known as AUC. The more the AUC the better your model is. The farther away your ROC curve is from the middle linear line, the better your model is. This is how ROC-AUC can help us judge the performance of our classification models as well as provide us a means to select one model from many classification models.

If you care for a class which is smaller in number independent of the fact whether it is positive or negative, go for ROC-AUC score.

10. PR Curve:

    In cases where the data is located mostly in the negative label, the ROC-AUC will give us a result that will not be able to represent the reality much because we primarily focus on a positive rate approach, TPR on y-axis and FPR on the x-axis. For instance, look at the example below:

		Actual Value	
		Yes (1)	No (0)
Predicted Value	Yes (1)	30 (TP)	50 (FP)
	No (0)	10 (FN)	1000 (TN)

Figure 6.52: PR curve

Over here you can see that most of the data lie under the negative label and ROC-AUC will not capture that information. In these kinds of scenarios, we turn to PR curves which are nothing but the Precision-Recall curve. In a PR curve, we'll calculate and plot Precision on Y-axis and Recall on X-axis to see how our model is performing.

6.3.1 When will you prefer F1 over ROC-AUC?

When you have a small positive class, then F1 score makes more sense. This is the common problem in fraud detection where positive labels are few. We can understand this statement with the following example.

Chapter 7

Multiclassifiers

7.1 Introduction

When there are several classifiers with a common objective it is called a multiclassifier. In Machine Learning multiclassifiers are sets of different classifiers which make estimates and are fused together, obtaining a result that is a combination of them. Lots of terms are used to refer to multiclassifiers: multi-models, multiple classifier systems, combining classifiers, decision committe, etc.

They can be divided into two big groups:

1. **Ensemble methods**: Ensemble is a Machine Learning concept in which the idea is to train multiple models using the same learning algorithm. Bagging and Boosting are the most extended ones.

2. **Hybrid methods**: Takes a set of different learners and combines them using new learning techniques. Stacking (or Stacked Generalization) is one of the main hybrid multiclassifiers.

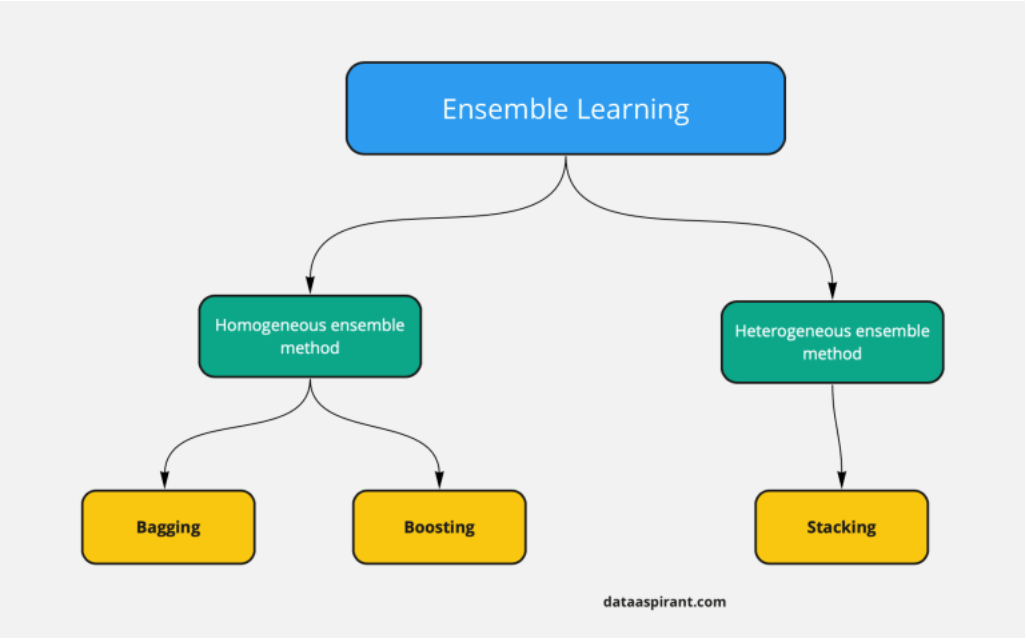


Figure 7.1: Multi Classifiers

The main causes of error in learning are due to noise, bias and variance. Ensemble helps to minimize these factors. These methods are designed to improve the stability and the accuracy of Machine Learning algorithms. Combinations of multiple classifiers decrease variance, especially in the case of unstable classifiers, and may produce a more reliable classification than a single classifier.

$$\begin{aligned} Err(x) &= (E[\hat{y}] - y)^2 + E[y - E(\hat{y})]^2 + \sigma_e^2 \\ Err(x) &= Bias^2 + Variance + Irreduciable\,Error \end{aligned} \tag{7.1}$$

- **Bias** error is useful to quantify how much on an average are the predicted values different from the actual value. A high bias error means we have a under-performing model which keeps on missing important trends.
- **Variance** on the other side quantifies how are the prediction made on same observation different from each other. A high variance model will over-fit on your training population and perform badly on any observation beyond training. Following diagram will give you more clarity (Assume that red spot is the real value and blue dots are predictions) :

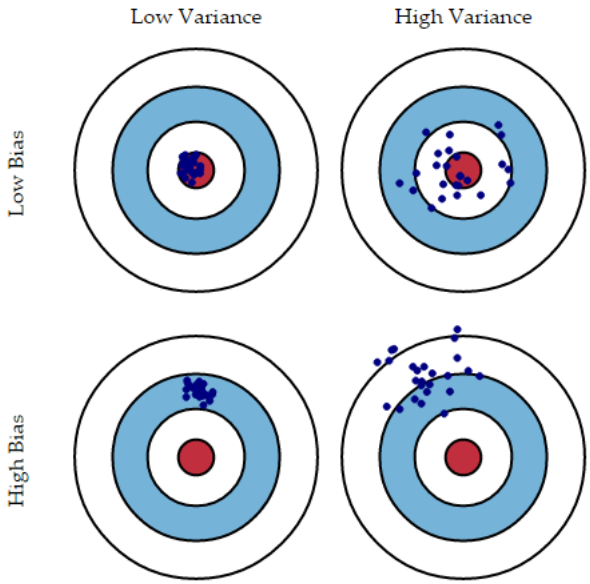


Figure 7.2: Difference between Bias and Variance

Normally, as you increase the complexity of your model, you will see a reduction in error due to lower bias in the model. However, this only happens till a particular point. As you continue to make your model more complex, you end up over-fitting your model and hence your model will start suffering from high variance.

A best model should maintain a balance between these two types of errors. This is known as the trade-off management of bias-variance errors. Ensemble learning is one way to execute this trade off analysis.

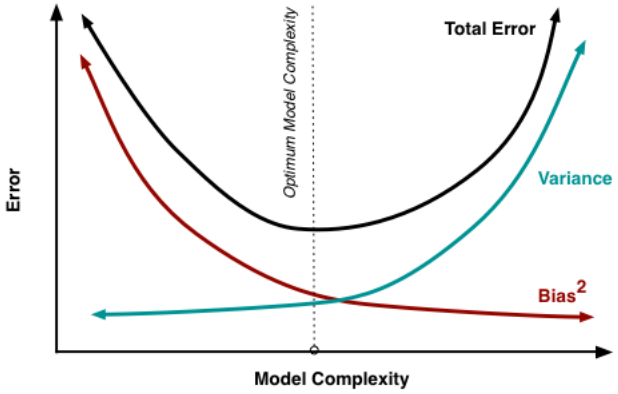


Figure 7.3: Relationship between Model complexity and Bias, Variance

7.2 Ensemble learning techniques

- **Max Voting** The max voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point. The predictions by each model are considered as a 'vote'. The predictions which we get from the majority of the models are used as the final prediction.

  For example, when you asked 5 of your friends to rate a movie, the calculation of final rating from individual ratings is as follows

	Friend 1	Friend 2	Friend 3	Friend 4	Friend 5	Final rating
rating	5	4	5	4	4	4

- **Averaging** Similar to the max voting technique, multiple predictions are made for each data point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for making predictions in regression problems or while calculating probabilities for classification problems.

  For example, when you asked 5 of your friends to rate a movie, the calculation of final rating from individual ratings is as follows

	Friend 1	Friend 2	Friend 3	Friend 4	Friend 5	Final rating
rating	5	4	5	4	4	4.4

- **Weighted Average** This is an extension of the averaging method. All models are assigned different weights defining the importance of each model for prediction. For instance, if two of your colleagues are critics, while others have no prior experience in this field, then the answers by these two friends are given more importance as compared to the other people.

  For example, when you asked 5 of your friends to rate a movie, the calculation of final rating from individual ratings is as follows

	Friend 1	Friend 2	Friend 3	Friend 4	Friend 5	Final rating
weight	0.23	0.23	0.18	0.18	0.18	
rating	5	4	5	4	4	4.41

- **BAGG**ing : or **B**ootstrap **AGG**regating.

  Bootstrapping is a sampling technique in which we create subsets of observations from the original dataset, with replacement. The size of the subsets is the same as the size of the original set.

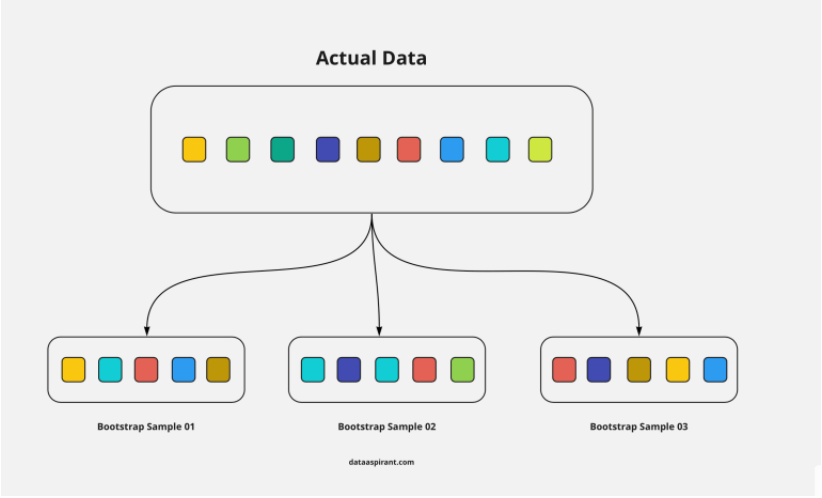


Figure 7.4: Bootstrap Sample

BAGGing gets its name because it combines Bootstrapping and Aggregation to form one ensemble model. Given a sample of data, multiple bootstrapped subsamples are pulled. The size of subsets created for bagging may be less than the original set. A model is formed on each of the bootstrapped subsamples. An algorithm is used to aggregate over the models to form the most efficient predictor. The image below will help explain:

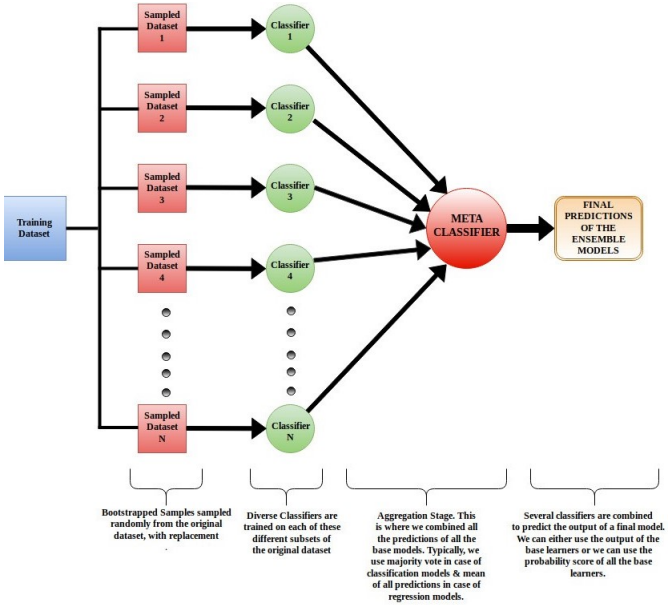


Figure 7.5: Bagging

- **Boosting** : Boosting is an iterative technique which adjust the weight of an observation based on the last classification. If an observation was classified incorrectly, it tries to increase the weight of this observation and vice versa. Boosting in general decreases the bias error and builds strong predictive models. However, they may sometimes over fit on the training data. There are several boosting methods like AdaBoost, LPBoost, XGBoost, Gradient-Boost, BrownBoost.

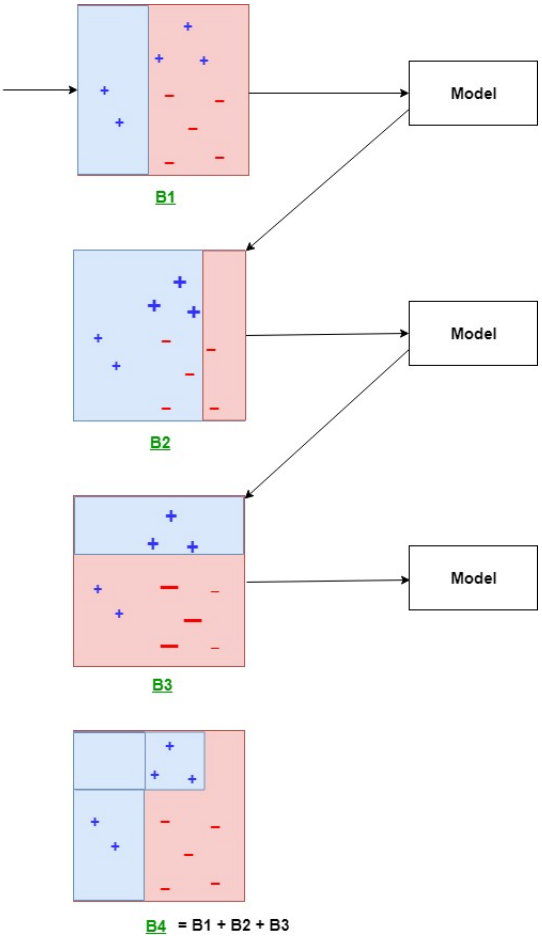


Figure 7.6: Boosting

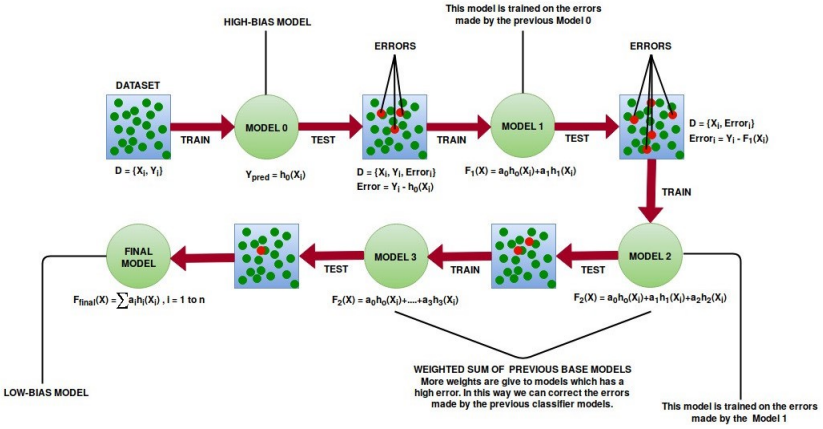


Figure 7.7: Boosting

- **Stacking** : This is a very interesting way of combining models. Here we use a learner to combine output from different learners. This can lead to decrease in either bias or variance error depending on the combining learner we use.

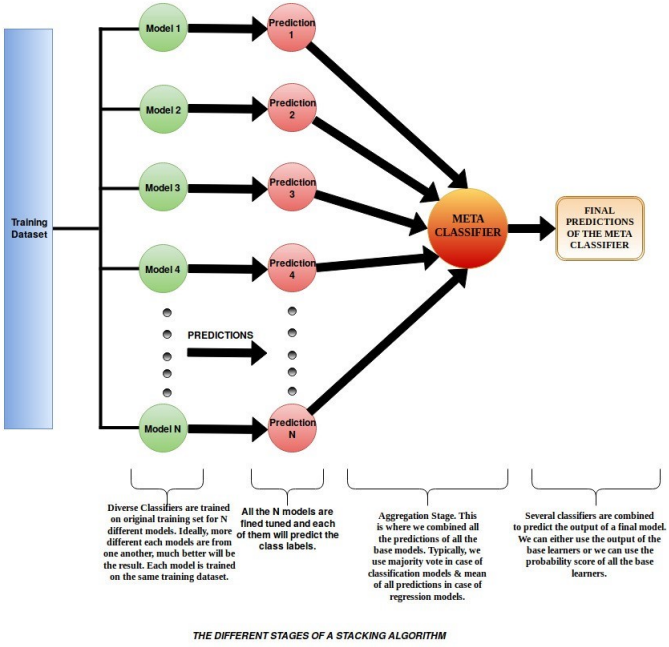


Figure 7.8: Stacking

7.3 Comparison of Bagging and Boosting

The details of comparison between Bagging and Boosting are as follows

Similarities	Differences
Both are ensemble methods to get N learners from 1 learner …	… but, while they are built independently for Bagging, Boosting tries to add new models that do well where previous models fail.
Both generate several training data sets by random sampling …	… but only Boosting determines weights for the data to tip the scales in favor of the most difficult cases.
Both make the final decision by averaging the N learners (or taking the majority of them)…	… but it is an equally weighted average for Bagging and a weighted average for Boosting, more weight to those with better performance on training data.
Both are good at reducing variance and provide higher stability …	… but only Boosting tries to reduce bias. On the other hand, Bagging may solve the over-fitting problem, while Boosting can increase it.

7.4 Random Forest Algorithm

7.4.1 What is Random Forest?

Random Forest is a versatile machine learning method capable of performing both regression and classification tasks. It also undertakes dimensional reduction methods, treats missing values, outlier values and other essential steps of data exploration, and does a fairly good job. It is a type of ensemble learning method, where a group of weak models combine to form a powerful model.

A decision tree model has high variance and low bias which can give us pretty unstable output unlike the commonly adopted logistic regression, which has high bias and low variance. That is the only point when Random Forest comes to the rescue.

In Random Forest, we grow multiple trees as opposed to a single tree in CART model. To classify a new object based on attributes, each tree gives a classification

and we say the tree "votes" for that class. The forest chooses the classification having the most votes (over all the trees in the forest) and in case of regression, it takes the average of outputs by different trees.

The reason that the random forest model works so well is: A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.

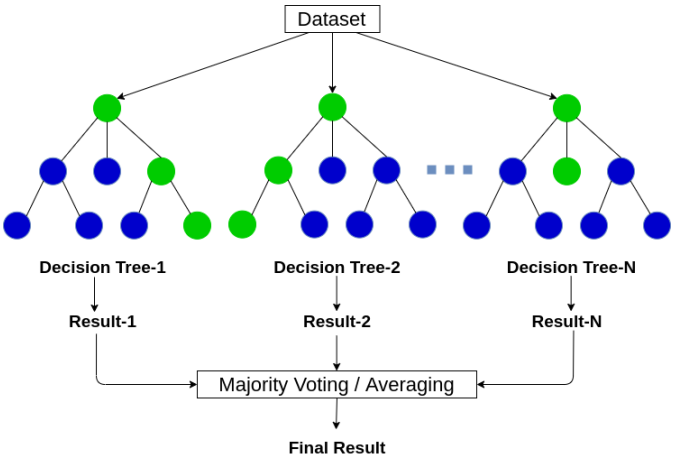


Figure 7.9: Random forest algorithm

7.4.2 How Random Forest algorithm works?

In Random Forest, each tree is planted & grown as follows:

1. Assume number of cases in the training set is N. Then, sample of these N cases $(\frac{2}{3})$ is taken at random but with replacement. This sample will be the training set for growing the tree.

2. If there are M input variables, a number $m < M$ is specified such that at each node, m variables ($\sqrt{M}$ for classification and $\frac{M}{3}$ for regression) are selected at random out of the M. The best split on these m is used to split the node. The value of m is held constant while we grow the forest.

3. Each tree is grown to the largest extent possible and there is no pruning.

4. Predict new data by aggregating the predictions of the n trees (i.e., majority votes for classification, average for regression).

7.4.3 Pros and Cons of Random Forest

7.4.3.1 Pros:

- This algorithm can solve both type of problems i.e. classification and regression and does a decent estimation at both fronts.
- Can handle large data set with higher dimensionality. It can handle thousands of input variables and identify most significant variables so it is considered as one of the dimensionality reduction methods. Further, the model outputs Importance of variable, which can be a very handy feature.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- It has methods for balancing errors in datasets where classes are imbalanced.
- Random Forest involves sampling of the input data with replacement called as bootstrap sampling. Here one third of the data is not used for training and can be used to testing. These are called the out of bag samples. Error estimated on these out of bag samples is known as out of bag error. Study of error estimates by Out of bag, gives evidence to show that the out-of-bag estimate is as accurate as using a test set of the same size as the training set. Therefore, using the out-of-bag error estimate removes the need for a set aside test set.

7.4.3.2 Cons:

- It surely does a good job at classification but not as good as for regression problem as it does not give continuous output. In case of regression, it doesn't predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.
- Random Forest can feel like a black box approach for statistical modelers – you have very little control on what the model does. You can at best – try different parameters and random seeds!

7.4.4 Random Forests vs Decision Trees

- Random forests is a set of multiple decision trees.

- Deep decision trees may suffer from overfitting, but random forests prevents overfitting by creating trees on random subsets.
- Decision trees are computationally faster.
- Random forests is difficult to interpret, while a decision tree is easily interpretable and can be converted to rules.

Chapter 8

Clustering

The classic definition for clustering is described as follows:

- Instances, in the same cluster, must be similar as much as possible;
- Instances, in the different clusters, must be different as much as possible;
- Measurement for similarity and dissimilarity must be clear and have the practical meaning;

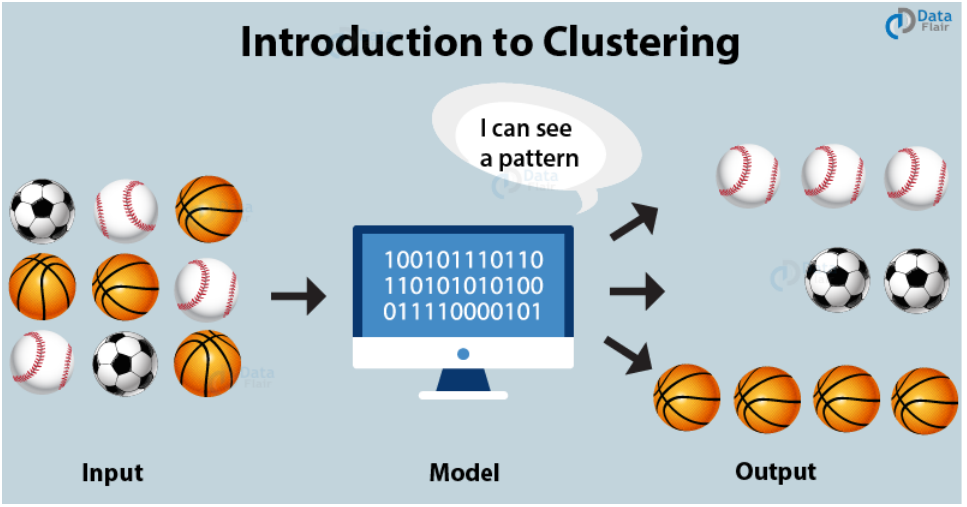


Figure 8.1: Clustering

Classification	Clustering
Uses labelled data as the input	Uses unlabelled data as the input
The output is known	The output is unknown
Uses supervised machine learning	Uses unsupervised machine learning
A training data set is provided and used to produce classifications	A training data set is not provided and used to produce clusters
Examples of algorithms: Decision-trees, Bayesian Classifiers and Support Vector Machines (SVM)	Examples of algorithms: Partition-based clustering (k-means), Hierarchical clustering (agglomerative & divisive) and DBSCAN
Can be more compex than clustering	Can be less compex than classification
Does not specify areas for improvement	Specifies areas for improvement
Two-phase	Single-phase
Boundary conditions must be specified	Boundary conditions do not always need to be specified

Figure 8.2: Classification_vs_Clustering

8.1 Distance and Similarity

Distance (dissimilarity) and similarity are the basis for constructing clustering algorithms. As for quantitative data features, distance is preferred to recognize the relationship among data. And similarity is preferred when dealing with qualitative data features. The common used distance functions for quantitative data feature are summarized in Table 8.1. The common used similarity functions for qualitative data feature are summarized in Table 8.2.

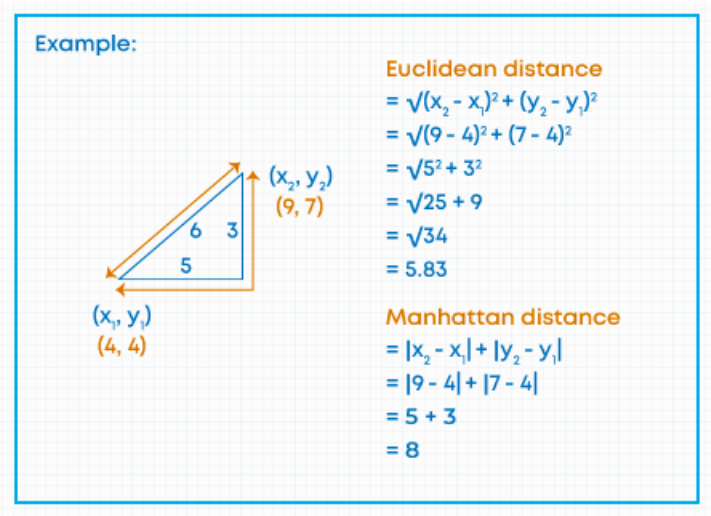


Figure 8.3: Euclidean and City block distance

	Allen Solly	Arrow	Peter England	US Polo	Van Heusen	Zodiac
Customer 1	4	5	3	5	2	1
Customer 2	1	2	4	3	3	5

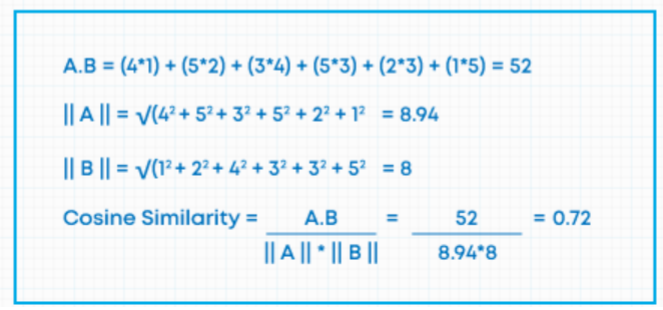


Figure 8.4: Cosine Similarity

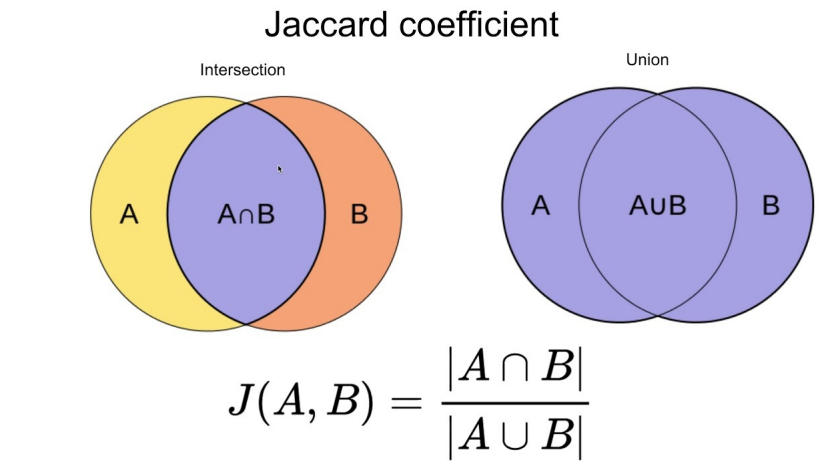


Figure 8.5: Jaccard Similarity

Store 1	Bread	Jam	Biscuits	Coke	Beer	Cake		
Store 2	Coke	Curd	Butter	Bread	Cake	Jam		
	Bread	**Jam**	**Biscuits**	**Coke**	**Beer**	**Cake**	**Curd**	**Butter**
Store 1	1	1	1	1	1	1	0	0
Store 2	1	1	0	1	0	1	1	1

Figure 8.6: Jaccard Similarity

Dist	A	B	C	D	E	F
A	0.00	0.71	5.66	3.61	4.24	3.20
B	0.71	0.00	4.95	2.92	3.54	2.50
C	5.66	4.95	0.00	2.24	1.41	2.50
D	3.61	2.92	2.24	0.00	1.00	0.50
E	4.24	3.54	1.41	1.00	0.00	1.12
F	3.20	2.50	2.50	0.50	1.12	0.00

Figure 8.7: Distance Matrix

Table 8.1: Distance functions

Name	Formula	Explanation
Minkowski distance	$\left(\sum_{l=1}^{d} \lvert x_{il} - x_{jl} \rvert^n\right)^{1/n}$	A set of definitions for distance: 1. City-block distance when $n = 1$ 2. Euclidean distance when $n = 2$ 3. Chebyshev distance when $n \to \infty$
Standardized Euclidean distance	$\left(\sum_{l=1}^{d} \left\lvert \frac{x_{il} - x_{jl}}{s_l} \right\rvert^2\right)^{1/2}$	1. S stands for the standard deviation 2. A weighted Euclidean distance based on the deviation
Cosine distance	$1 - \cos\alpha = \frac{\boldsymbol{x}_i^T \boldsymbol{x}_j}{\lVert \boldsymbol{x}_i \rVert \lVert \boldsymbol{x}_j \rVert}$	1. Stay the same in face of the rotation change of data 2. The most commonly used distance in document area

Table 8.2: Similarity functions

Name	Formula	Explanation
Jaccard similarity	$J(A,B) = \frac{\lvert A \cap B \rvert}{\lvert A \cup B \rvert}$	1. Measure the similarity of two sets 2. $\lvert X \rvert$ Stands for the number of elements of set X 3. Jaccard distance $= 1 -$ Jaccard similarity

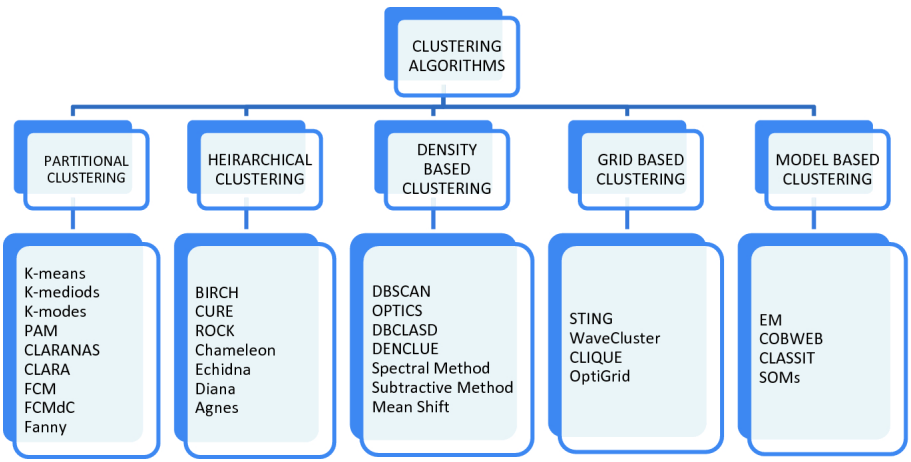


Figure 8.8: Clustering Types

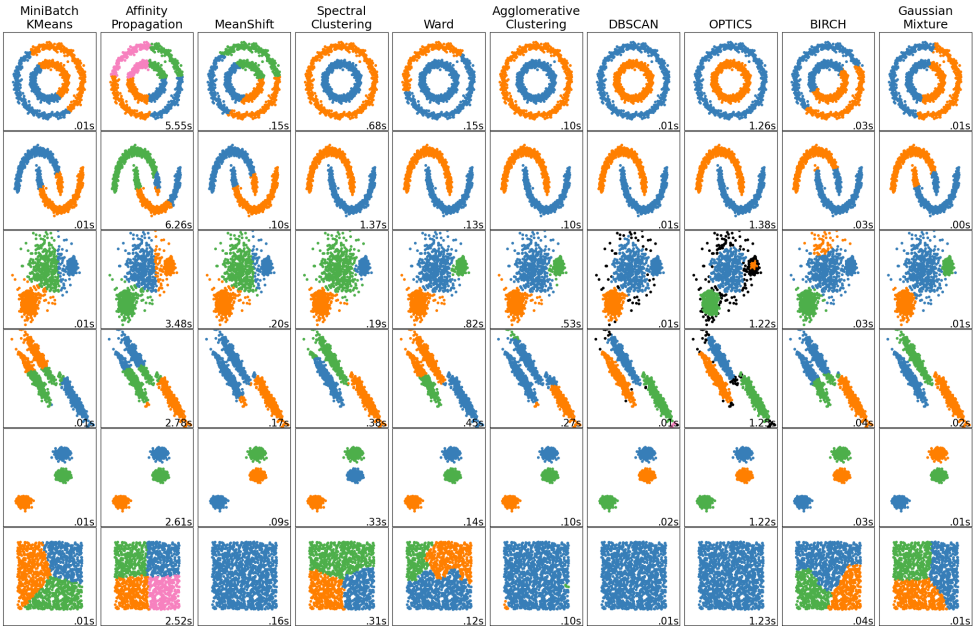


Figure 8.9: Clustering Algorithms' performance on Toy datasets

There are three popular categories of clustering algorithms Xu and Tian (2015):

1. **Partitional clustering** divides data objects into non overlapping groups. In other words, no object can be a member of more than one cluster, and every cluster must have at least one object. Two examples of partitional clustering algorithms are k-means and k-medoids. These algorithms are both non deterministic, meaning they could produce different results from two separate runs even if the runs were based on the same input.

    - Partitional clustering methods have several strengths:
        - They work well when clusters have a spherical shape.
        - They are scalable with respect to algorithm complexity.
    - They also have several weaknesses:
        - They are not well suited for clusters with complex shapes and different sizes.
        - They break down when used with clusters of different densities.

2. **Hierarchical clustering** determines cluster assignments by building a hierarchy.

- This is implemented by either a bottom-up or a top-down approach:
  - **Agglomerative clustering** is the bottom-up approach. It merges the two points that are the most similar until all points have been merged into a single cluster.
  - **Divisive clustering** is the top-down approach. It starts with all points as one cluster and splits the least similar clusters at each step until only single data points remain.

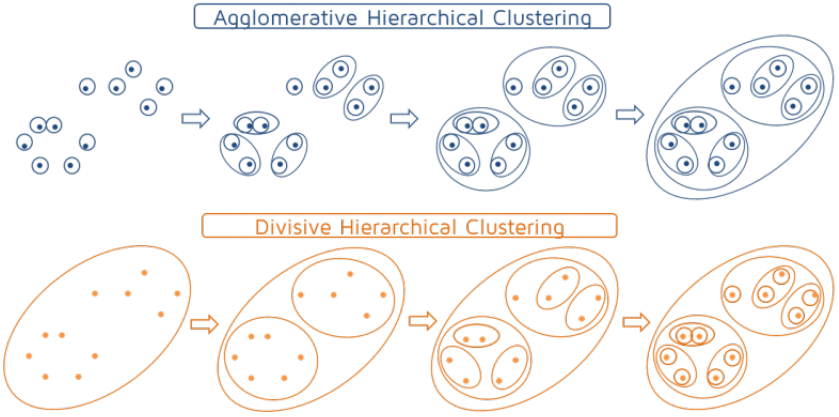


Figure 8.10: Agglomerative vs Divisive clustering

- These methods produce a tree-based hierarchy of points called a **dendrogram**. Similar to partitional clustering, in hierarchical clustering the number of clusters (k) is often predetermined by the user. Clusters are assigned by cutting the dendrogram at a specified depth that results in k groups of smaller dendrograms.

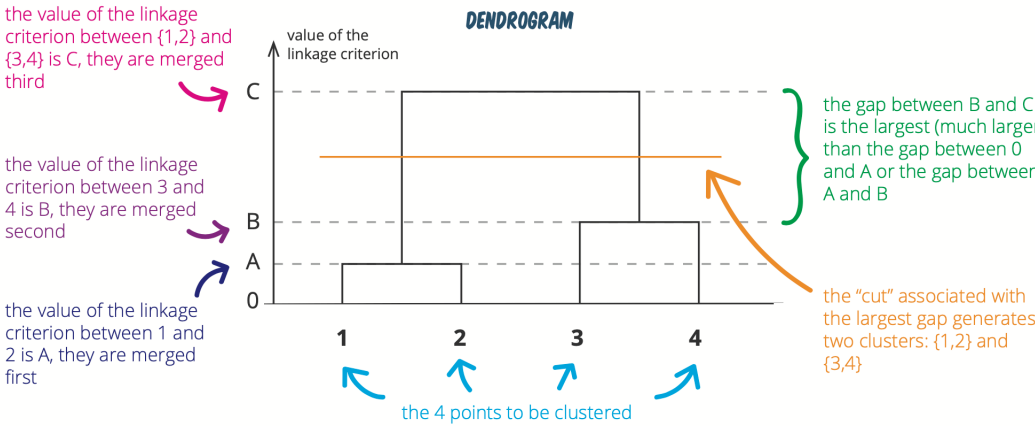


Figure 8.11: Dendrogram

- Unlike many partitional clustering techniques, hierarchical clustering is a deterministic process, meaning cluster assignments won't change when you run an algorithm twice on the same input data.
- Linkages used in Hierarchical Cluistering

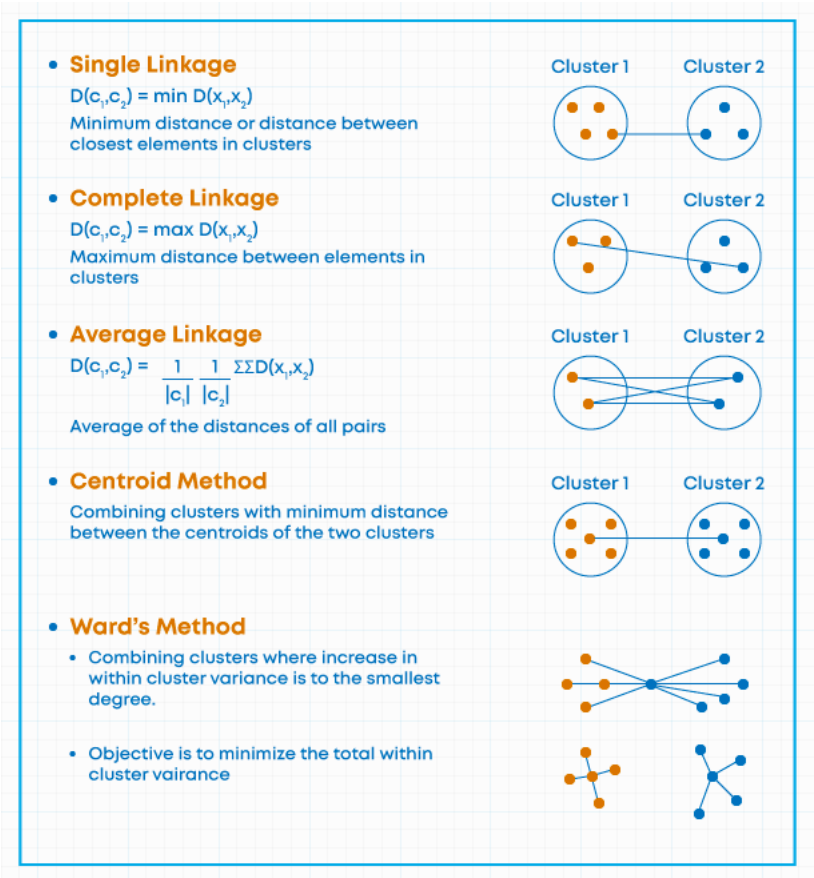


Figure 8.12: Linkages

- The strengths of hierarchical clustering methods include the following:
  - They often reveal the finer details about the relationships between data objects.
  - They provide an interpretable dendrogram.
- The weaknesses of hierarchical clustering methods include the following:
  - They're computationally expensive with respect to algorithm complexity.
  - They're sensitive to noise and outliers.

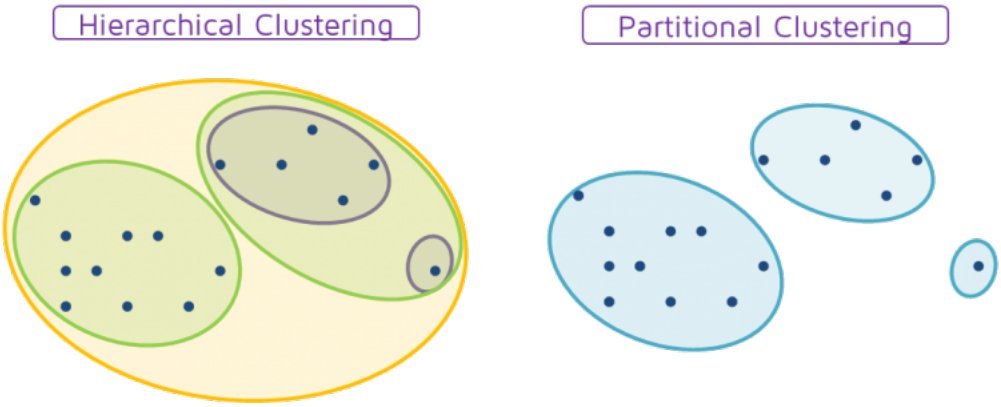


Figure 8.13: Partition vs Hierarchical clustering

3. **Density-based clustering** determines cluster assignments based on the density of data points in a region.

    - Clusters are assigned where there are high densities of data points separated by low-density regions.
    - Unlike the other clustering categories, this approach doesn't require the user to specify the number of clusters. Instead, there is a distance-based parameter that acts as a tunable threshold. This threshold determines how close points must be to be considered a cluster member.
    - Examples of density-based clustering algorithms include Density-Based Spatial Clustering of Applications with Noise, or DBSCAN, and Ordering Points To Identify the Clustering Structure, or OPTICS.
    - The strengths of density-based clustering methods include the following:
        - They excel at identifying clusters of nonspherical shapes.
        - They are resistant to outliers.
    - The weaknesses of density-based clustering methods include the following:
        - They are not well suited for clustering in high-dimensional spaces.
        - They have trouble identifying clusters of varying densities.

8.2 KMeans

The KMeans algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields.

The k-means algorithm divides a set of N samples X into k disjoint clusters C, each described by the mean μ_j of the samples in the cluster. The means are commonly called the cluster "centroids"; note that they are not, in general, points from , although they live in the same space.

Algorithm 8.1 k-means. The k-means algorithm for partitioning, where each cluster's center is represented by the mean value of the objects in the cluster.

Input:

- k: the number of clusters,
- D: a data set containing n objects.

Output: A set of k clusters.

Method:

arbitrarily choose k objects from D as the initial cluster centers;
repeat
(re)assign each object to the cluster to which the object is the most similar,

based on the mean value of the objects in the cluster; update the cluster means, that is, calculate the mean value of the objects for each cluster;
until no change;

The K-means algorithm aims to choose centroids that minimise the inertia, or within-cluster sum-of-squares criterion:

$$\text{WCSS or Inertia} = \sum_{i=1}^{k} \sum_{p \in C_i} \text{dist}\left(\boldsymbol{p}, \boldsymbol{\mu_i}\right)^2 \tag{8.1}$$

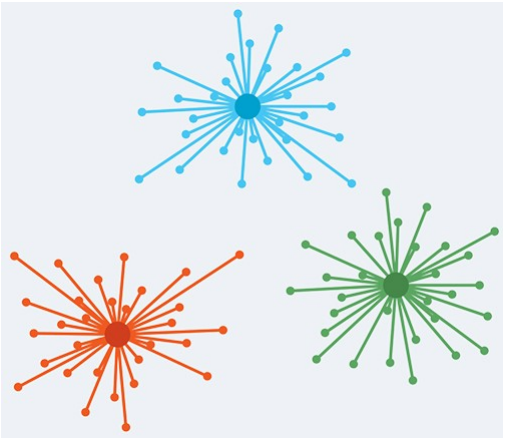


Figure 8.14: Calculation of inertia, With in Cluster Sum of Squares (WCSS)

Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks:

- Inertia makes the assumption that clusters are convex and isotropic, which is not always the case. It responds poorly to elongated clusters, or manifolds with irregular shapes.
- Inertia is not a normalized metric: we just know that lower values are better and zero is optimal. But in very high-dimensional spaces, Euclidean distances tend to become inflated (this is an instance of the so-called "curse of dimensionality"). Running a dimensionality reduction algorithm such as Principal component analysis (PCA) prior to k-means clustering can alleviate this problem and speed up the computations.
- The k-means problem is solved using either Lloyd's or Elkan's algorithm.
- The average complexity is given by $O(knT)$, were n is the number of samples and T is the number of iteration.
- The worst case complexity is given by $O(n^{(k+2/p)})$ with n = n_samples, p = n_features. Arthur and Vassilvitskii (2006)
- In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in local minima. That's why it can be useful to restart it several times.

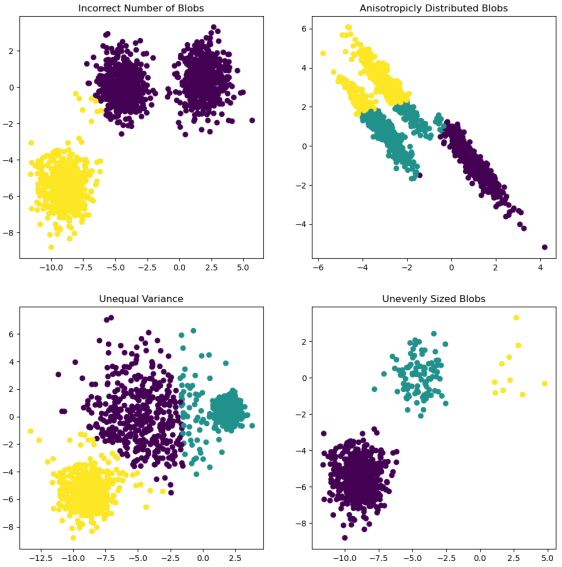


Figure 8.15: Situations where k-means will produce unintuitive and possibly unexpected clusters.

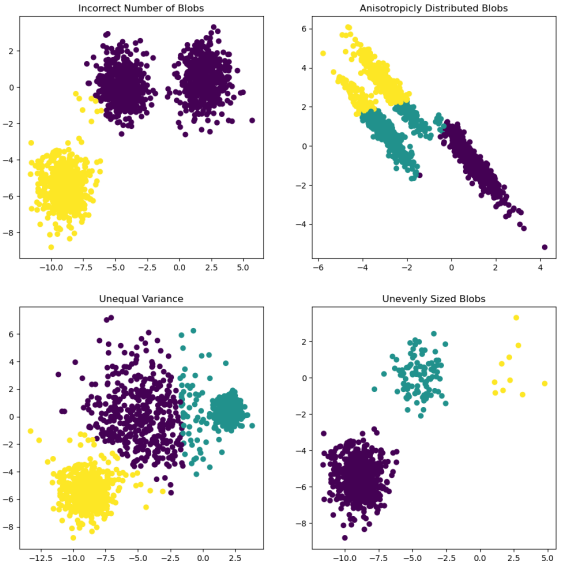


Figure 8.16: Situations where k-means will produce unintuitive and possibly unexpected clusters.

Chapter 9

ANN

9.1 The Neuron

The neuron is the basic unit inside a neural net. One neuron receives some signals that it combines and transforms to create a new single signal that is sent to the next level. Actually, a neuron is a very cool name for a function:

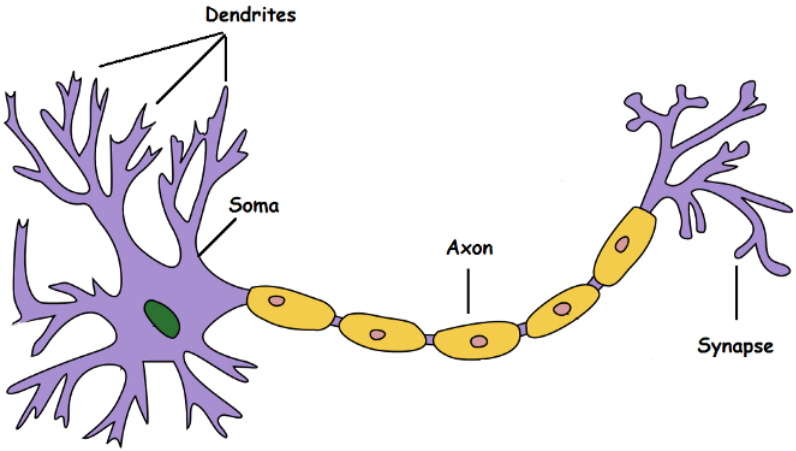


Figure 9.1: Biological Neuron

- Dendrite: Receives signals from other neurons
- Soma: Processes the information
- Axon: Transmits the output of this neuron

- Synapse: Point of connection to other neurons

9.1.1 McCulloch-Pitts Neuron

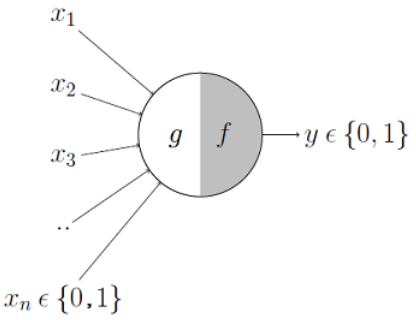


Figure 9.2: McCulloch-Pitts Neuron

- The first part, g takes an input, performs an aggregation and based on the aggregated value the second part, f makes a decision.
- These inputs can either be excitatory or inhibitory. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs.
- Excitatory inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together.

$$g(x_1, x_2, x_3, \ldots, x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$y = f(g(\mathbf{x})) = 1 \text{ if } \quad g(\mathbf{x}) \geq \theta$$

$$= 0 \quad \text{if } g(\mathbf{x}) < \theta$$

- We can see that $g(x)$ is just doing a sum of the inputs — a simple aggregation. And theta here is called thresholding parameter. For example, if the output is always 1 when the sum turns out to be 2 or more, the theta is 2 here. This is called the Thresholding Logic.
- Realization of simple functions using MP neuron

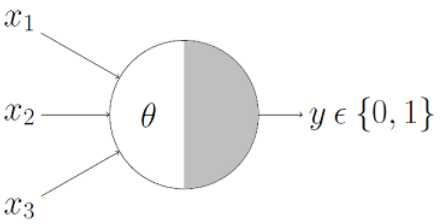


Figure 9.3: M-P Neuron: A Concise Representation

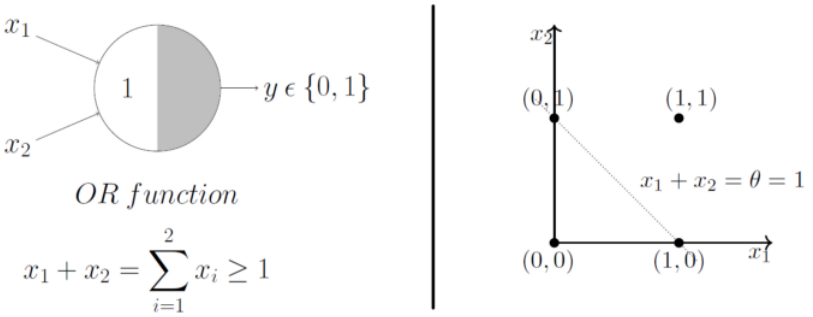


Figure 9.4: OR Function

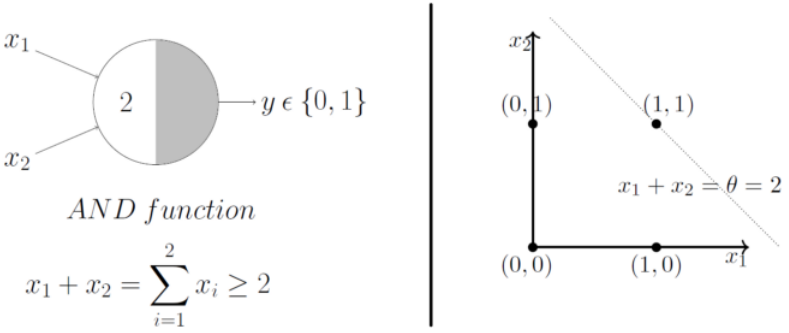
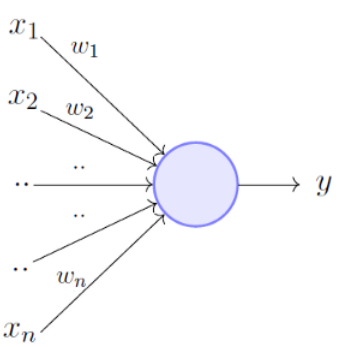


Figure 9.5: AND Function

- Limitations Of M-P Neuron
  - What about non-boolean (say, real) inputs?

– Do we always need to hand code the threshold?
– Are all inputs equal? What if we want to assign more importance to some inputs?
– What about functions which are not linearly separable? Say XOR function.

9.1.2 Perceptron



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

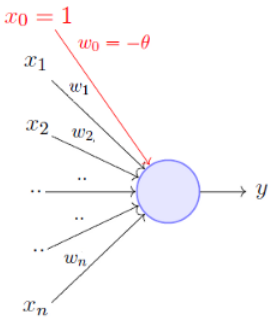
$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Figure 9.6: Perceptron



A more accepted convention,

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

$$where, \quad x_0 = 1 \quad and \quad w_0 = -\theta$$

Figure 9.7: Perceptron

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

Figure 9.8: MP Neuron vs Perceptron

x_1	x_2	OR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 > -w_0$$

One possible solution is
$w_0 = -1,\ w_1 = 1.1,\ w_2 = 1.1$

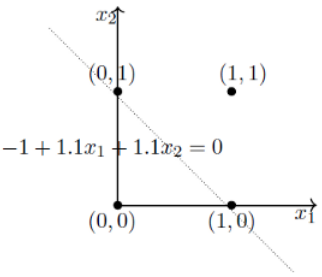


Figure 9.9: OR function using Perceptron

x_1	x_2	XOR	
0	0	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^{2} w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^{2} w_i x_i < 0$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$
$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$
$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 \geq 0 \implies w_1 + w_2 < -w_0$$

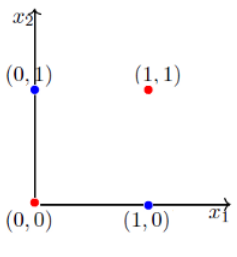


Figure 9.10: XOR function using Perceptron

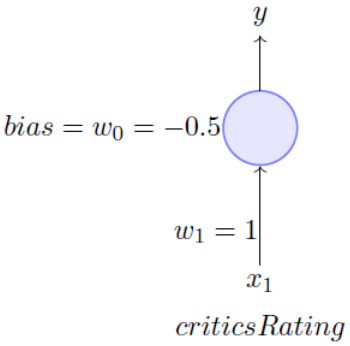


Figure 9.11: Perceptron behaviour

- What would be the decision for a movie with criticsRating = 0.51? Yes!
- What would be the decision for a movie with criticsRating = 0.49? No!
- Some might say that its harsh that we would watch a movie with a rating of 0.51 but not the one with a rating of 0.49.
- It is a characteristic of the perceptron function itself which behaves like a step function.

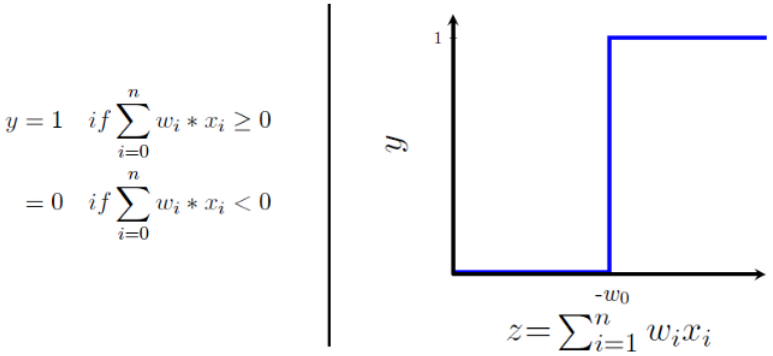


Figure 9.12: Perceptron behaviour

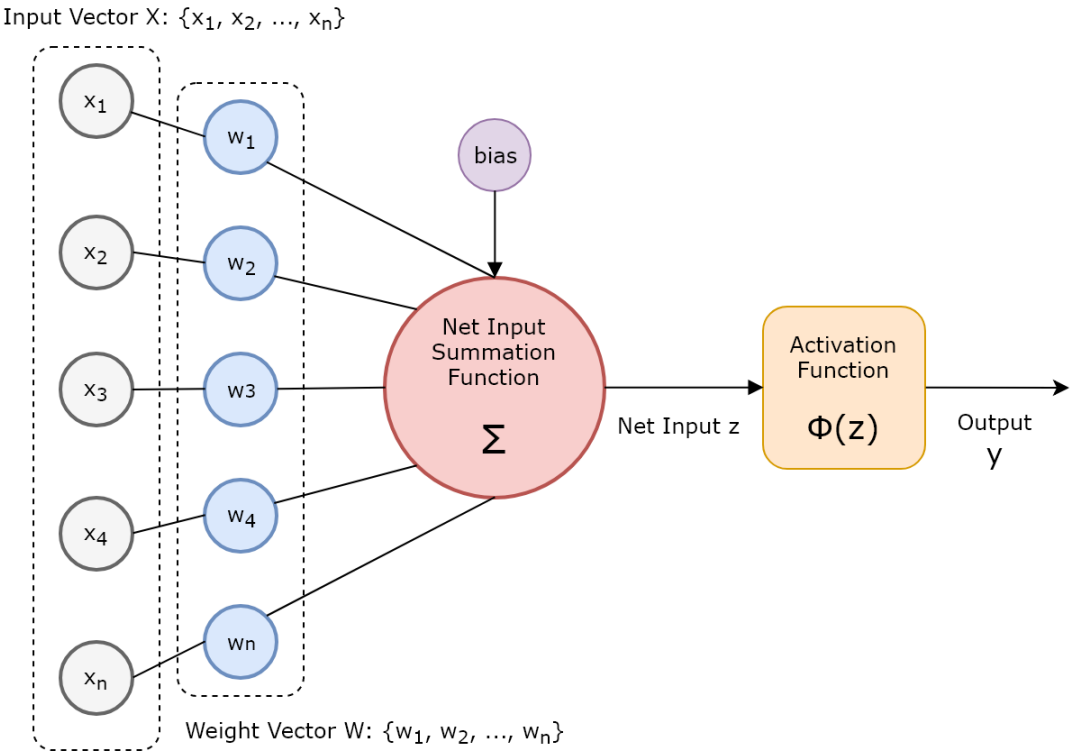


Figure 9.13: Perceptron

$$A = \frac{1}{1+e^{-x}}$$

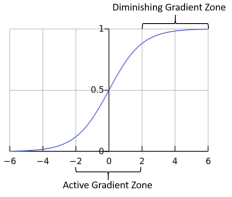


Figure 9.14: Sigmoid Function

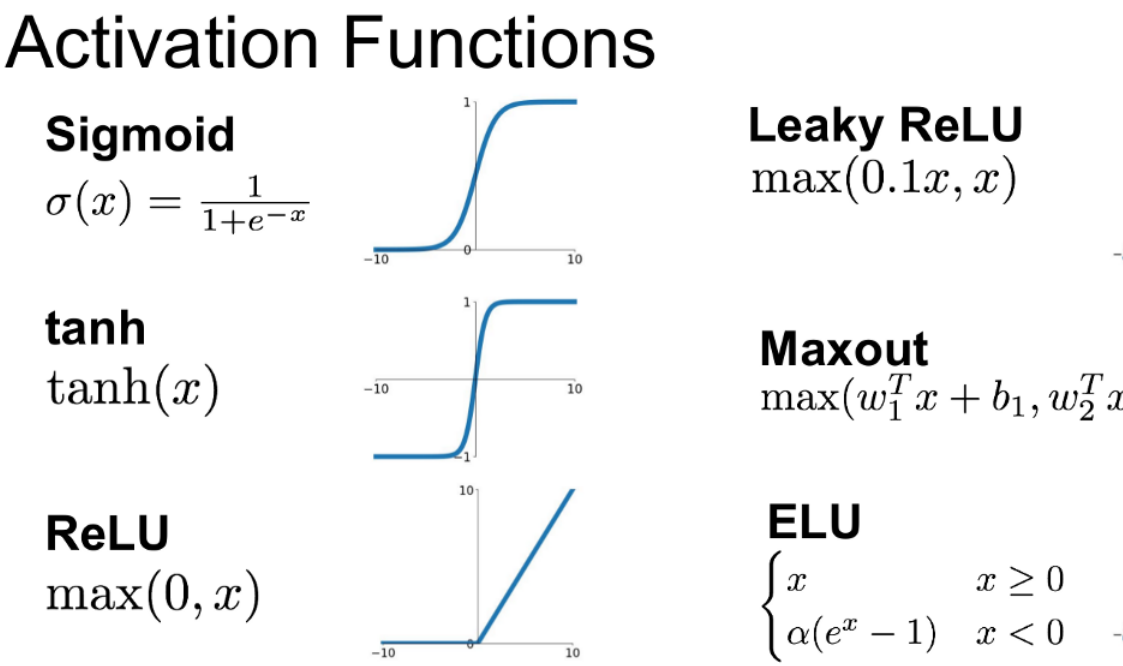


Figure 9.15: Sigmoid Function

Any of the inputs (other neurons) has a linked weight that can be interpreted as the intensity each signal comes to the particular neuron with.

The **propagation function** plays the role of combining the input data. It consists of the weighted sum of the input values. It includes a bias, an independent term that, in contrast with the signal provided by the rest of the equation, captures the existing noise in the relationship between x and y.

As you can see, the neuron seems to be very similar to linear regression, but it is not the same, due to the inclusion of the **activation function**. This activation distorts the previous combination and provides the network with the nonlinearity that is a key point when trying to find complex relationships in the data. The most common ones are:

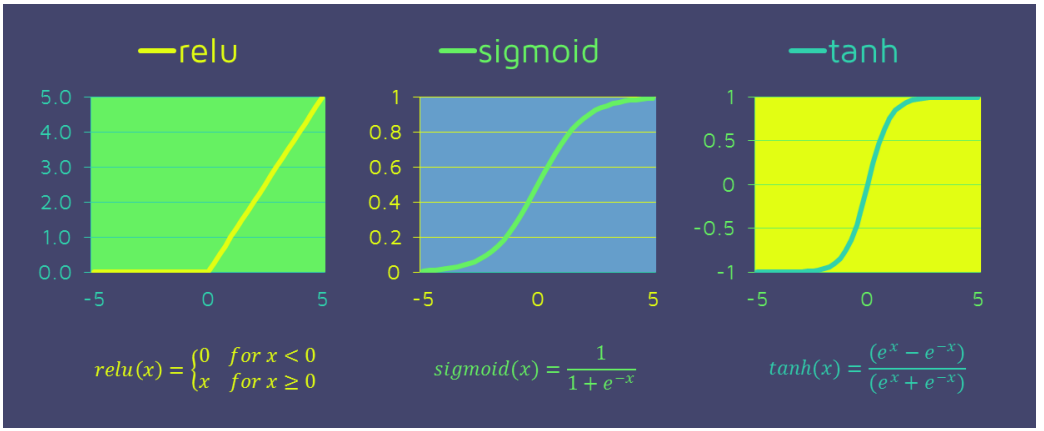


Figure 9.16: Activation functions

9.2 The Net

A neural network is a set of neurons. They are organized in levels, the layers. Neurons of one layer interact with neurons on the next layer through the weighted connections. The information flows from one level to the next one: The neurons of the same layer receive the same inputs, those processed in the neurons of the previous layer. The outputs of the neurons on the same layer will be combined in the neurons of the next one.

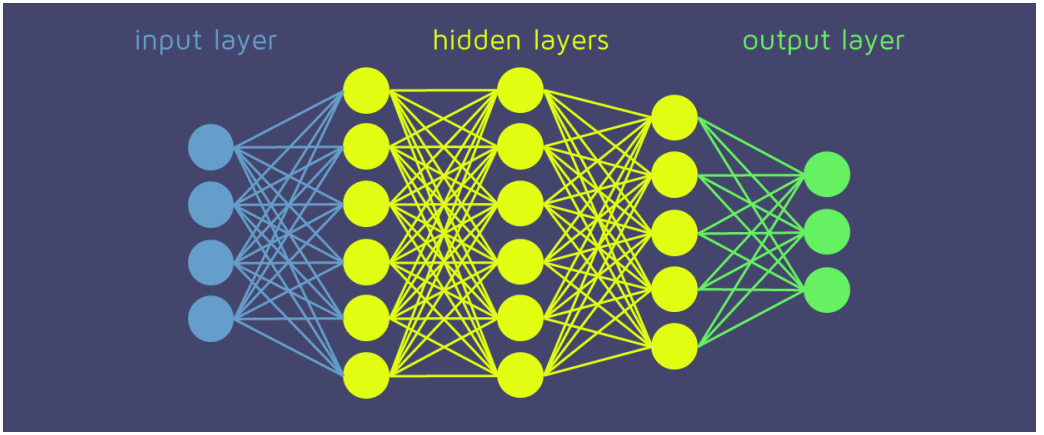


Figure 9.17: Neural network

The **first layer is the input layer** and has as many neurons as the number of available features that are used as inputs. The last one, the output layer, will have as many neurons as the needed outputs. When there are no more than these

two layers, we called it **perceptron**, the simplest net you can define. But the normal case it is to place between them the hidden layers, which can be as many as you prefer and can also contain as many neurons as you want.

This increment in the number of layers and the networks' complexity is known as deep learning, something like a conventional network on steroids.

Remember that any of these neurons is combining information by linear regression. It is easy to prove that the union of linear layers is also linear, as you can simplify it and obtain a unique equivalent layer. As mentioned before, the use of activation functions is necessary to introduce nonlinearity. These nonlinear distortions are what give the layers' sequence meaning and provide the neural nets with the power of modeling intricate relationships in data.

Find here a geometric interpretation of three types of nets, just a linear model, a simple net and a quite difficult one, and their success when trying to solve three different classification problems:

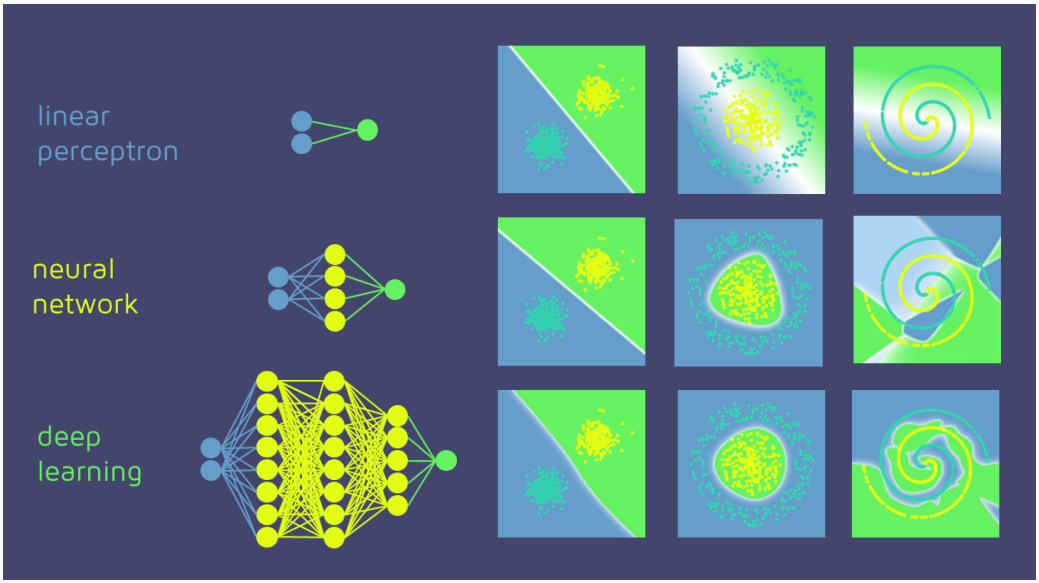


Figure 9.18: Neural networks comparison

9.3 Architecture of NN

In artificial neural networks, hidden layers are required if and only if the data must be separated non-linearly. In real-world problems, there is no way to determine the best number of hidden layers and neurons without trying.

Here are some guidelines to know the number of hidden layers and neurons per each hidden layer in a classification problem:

1. Based on the data, draw an expected decision boundary to separate the classes.
2. Express the decision boundary as a set of lines. Note that the combination of such lines must yield to the decision boundary.
3. The number of selected lines represents the number of hidden neurons in the first hidden layer.
4. To connect the lines created by the previous layer, a new hidden layer is added. Note that a new hidden layer is added each time you need to create connections among the lines in the previous hidden layer.
5. The number of hidden neurons in each new hidden layer equals the number of connections to be made.

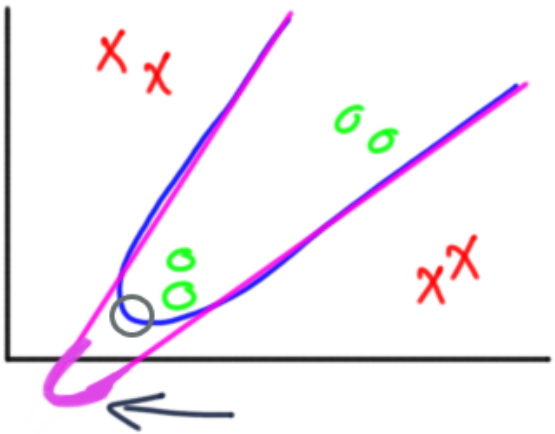


Figure 9.19: Sample problem 1

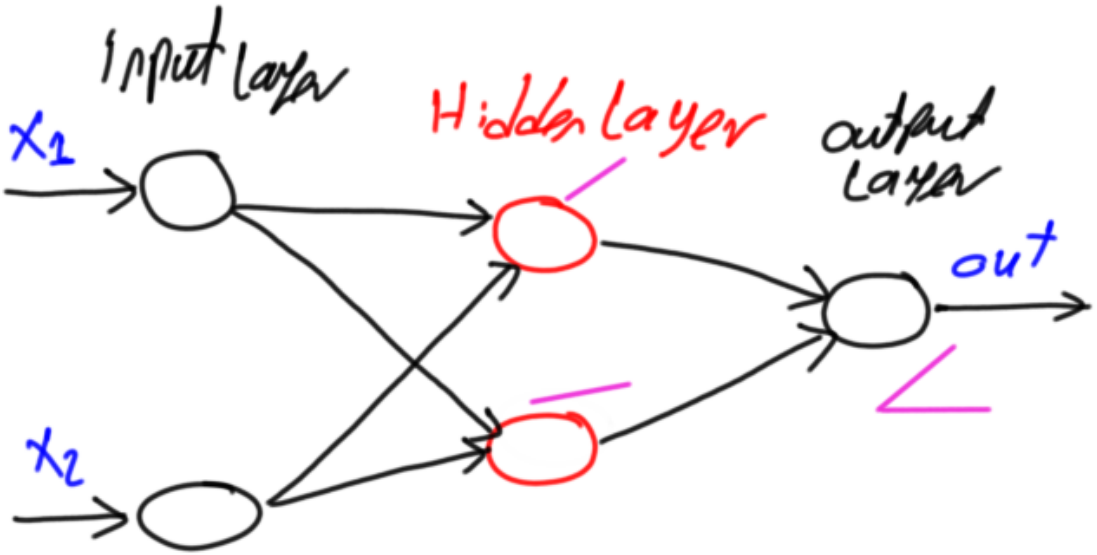


Figure 9.20: Architecture of NN for Problem 1

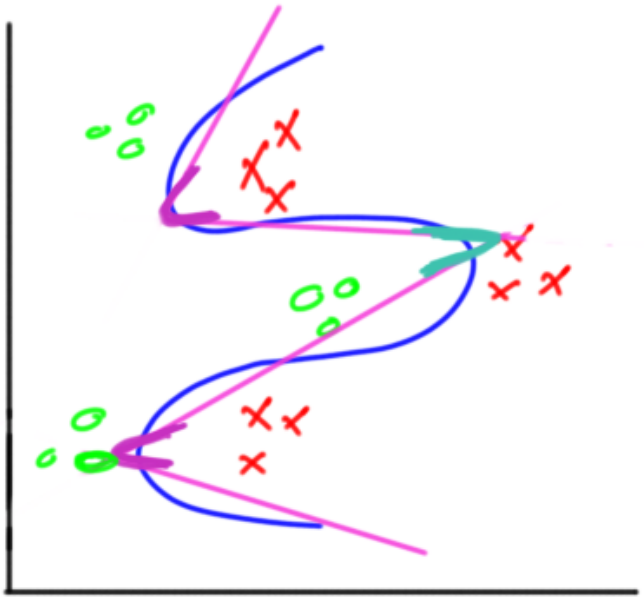


Figure 9.21: Sample problem 2

9.3.1 Solution to problem2

1. Because the first hidden layer will have hidden layer neurons equal to the number of lines, the first hidden layer will have four neurons. At the current time, the network will generate four outputs, one from each classifier.

2. Build a second hidden layer with two hidden neurons. The first hidden neuron will connect the first two lines and the last hidden neuron will connect the last two lines.

3. Up to this point, there are two separated curves. Thus there are two outputs from the network. Next is to connect such curves together in order to have just a single output from the entire network. In this case, the output layer neuron could be used to do the final connection rather than adding a new hidden layer.

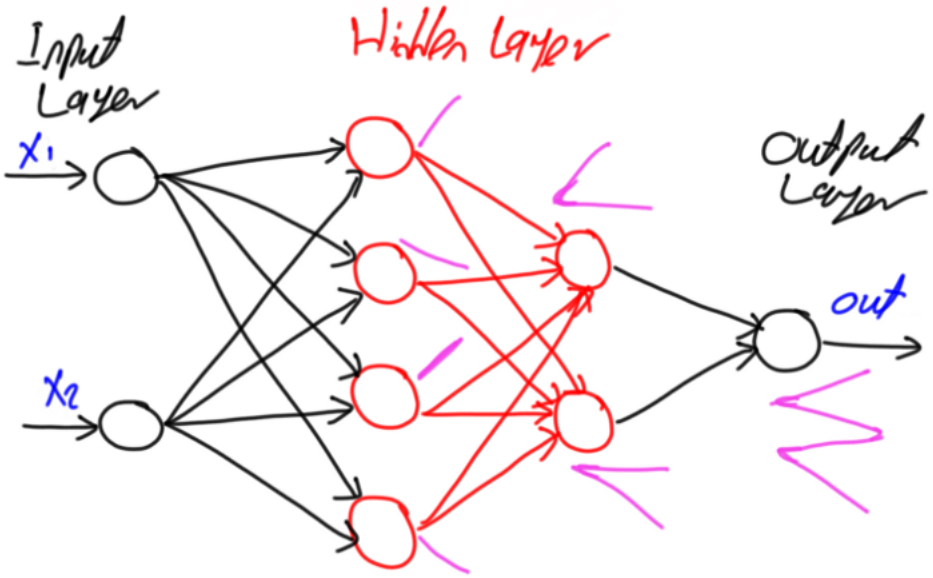


Figure 9.22: Architecture of NN for Problem 2

Number of layers and number of neurons are the two hyper-parameters of the net and so, a cross-validation optimization could help to set the most suitable values for them. What happens If the net has an inappropriate number of hidden neurons? Low complexity will lead to under-fitting, while too much complexity surely drives to over-fitting. The density of your net must be justified.

9.4 What activation function may I choose?

Although there are no golden rules to answer this, the ReLu function is often the best choice. It is computationally cheaper and fixes the Vanishing Gradient problem, which is an inconvenience when using Sigmoid and Tanh. ReLu presents disadvantages too, such as the Dying Gradient problem, and some versions exist to mitigate its weaknesses. It is quite frequent to find architectures that use the ReLu function for hidden layers and another one, usually a version of ReLu for the output layer.

9.4.1 Sigmoid Function

$$A = \frac{1}{1 + e^{-x}} \tag{9.1}$$

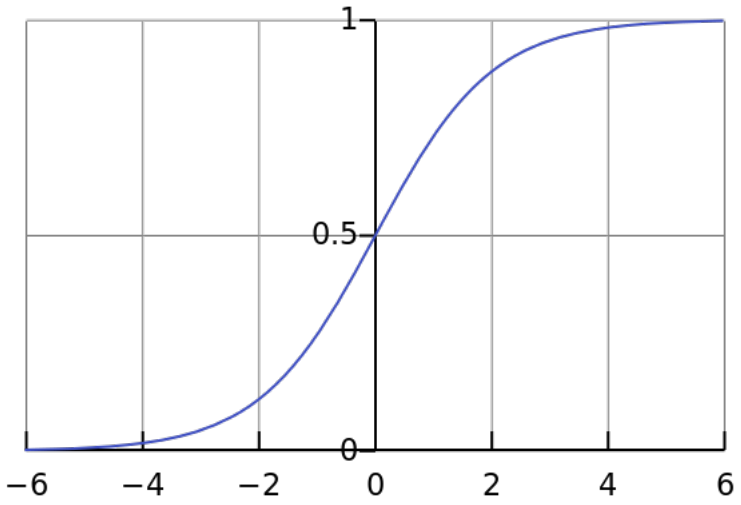


Figure 9.23: Sigmoid

It is nonlinear in nature. Combinations of this function are also nonlinear. Now we can stack layers.

What about non binary activations? Yes, that too!. It will give an analog activation unlike step function.

It has a smooth gradient too.

Between X values -2 to 2, Y values are very steep. Which means, any small changes in the values of X in that region will cause values of Y to change signifi-

cantly. That means this function has a tendency to bring the Y values to either end of the curve. It's good for a classifier considering its property? Yes ! It indeed is. It tends to bring the activations to either side of the curve (above x = 2 and below x = -2 for example). Making clear distinctions on prediction.

Another advantage of this activation function is, unlike linear function, the output of the activation function is always going to be in range (0,1) compared to (-inf, inf) of linear function. So we have our activations bound in a range. Nice, it won't blow up the activations then.

This is great. Sigmoid functions are one of the most widely used activation functions today. Then what are the problems with this?

If you notice, towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. What does that mean? The gradient at that region is going to be small. It gives rise to a problem of "vanishing gradients". So what happens when the activations reach near the "near-horizontal" part of the curve on either sides? Gradient is small or has vanished (cannot make significant change because of the extremely small value). The network refuses to learn further or is drastically slow (depending on use case and until gradient /computation gets hit by floating point value limits).

9.4.2 Tanh Function

Another activation function that is used is the tanh function.

$$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{9.2}$$

$$\tanh(x) = 2\,\text{sigmoid}(2x) - 1 \tag{9.3}$$

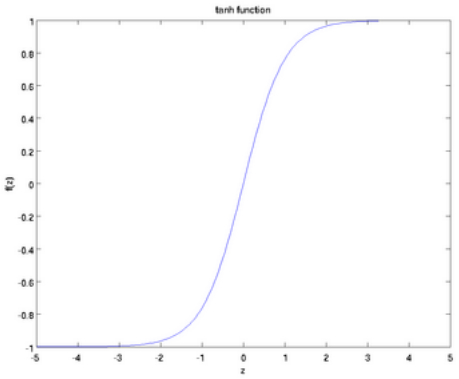


Figure 9.24: Tanh function

This has characteristics similar to sigmoid that we discussed above. One point to mention is that the gradient is stronger for tanh than sigmoid (derivatives are steeper). Deciding between the sigmoid or tanh will depend on your requirement of gradient strength. Like sigmoid, tanh also has the vanishing gradient problem.

9.4.3 ReLu function

The ReLu function is as shown above. It gives an output x if x is positive and 0 otherwise.

$$f(x) = max(0, x) \tag{9.4}$$

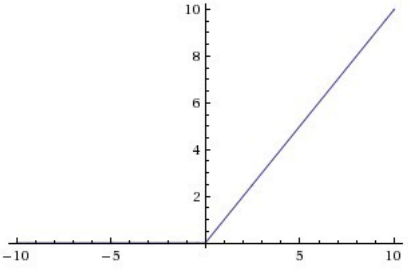


Figure 9.25: ReLu function

ReLu is nonlinear in nature. And combinations of ReLu are also non linear! (in fact it is a good approximator. Any function can be approximated with combinations of ReLu). Great, so this means we can stack layers. It is not bound though. The range of ReLu is [0, inf). This means it can blow up the activation.

Another point that I would like to discuss here is the sparsity of the activation.

Imagine a big neural network with a lot of neurons. Using a sigmoid or tanh will cause almost all neurons to fire in an analog way (remember?). That means almost all activations will be processed to describe the output of a network. In other words the activation is dense. This is costly. We would ideally want a few neurons in the network to not activate and thereby making the activations sparse and efficient.

ReLu give us this benefit. Imagine a network with random initialized weights (or normalised) and almost 50% of the network yields 0 activation because of the characteristic of ReLu (output 0 for negative values of x). This means a fewer neurons are firing (sparse activation) and the network is lighter. Woah, nice! ReLu seems to be awesome! Yes it is, but nothing is flawless. Not even ReLu.

Because of the horizontal line in ReLu(for negative X), the gradient can go towards 0. For activations in that region of ReLu, gradient will be 0 because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is 0, nothing changes). This is called dying ReLu problem. This problem can cause several neurons to just die and not respond making a substantial part of the network passive. There are variations in ReLu to mitigate this issue by simply making the horizontal line into non-horizontal component . for example $y = 0.01x$ for $x < 0$ will make it a slightly inclined line rather than horizontal line. This is leaky ReLu. There are other variations too. The main idea is to let the gradient be non zero and recover during training eventually.

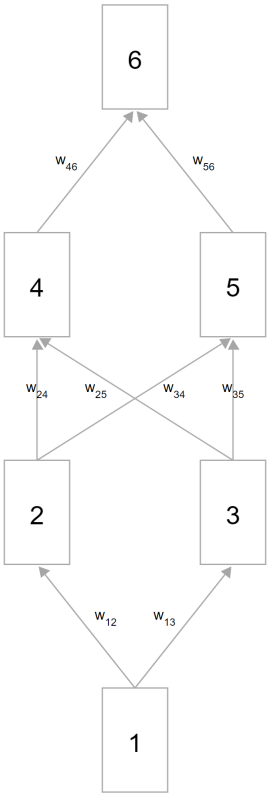
ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

A sigmoid works well for a classifier, because approximating a classifier function as combinations of sigmoid is easier than maybe ReLu, for example. Which will lead to faster training process and convergence.

If you don't know the nature of the function you are trying to learn, then maybe I would suggest start with ReLu, and then work backwards.

9.5 Backpropagation algorithm

Simple Neural Network



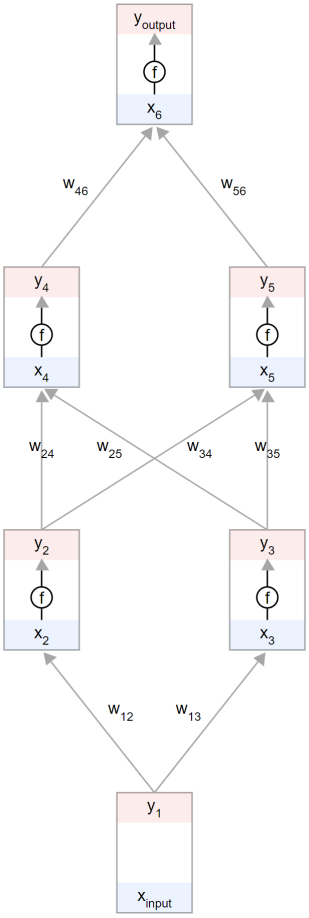
The backpropagation algorithm is essential for training large neural networks quickly. In the figure you can see a neural network with one input, one output node and two hidden layers of two nodes each.

Nodes in neighboring layers are connected with weights w_{ij}, which are the **network parameters**.

Activation function

As shown in the figure, each node has a total input $\boldsymbol{x}$, an activation function $f(\boldsymbol{x})$ and an output $\boldsymbol{y} = f(\boldsymbol{x})$. $f(\boldsymbol{x})$ has to be a non-linear function, otherwise the neural network will only be able to learn linear models.

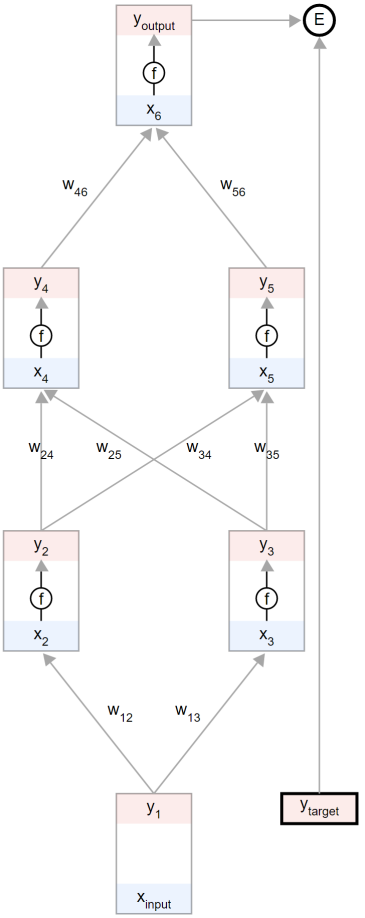
A commonly used activation function is the Sigmoid function: $f(\boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{x}}}$.



Error function

The goal is to learn the weights of the network automatically from data such that the predicted output $\boldsymbol{y}_{output}$ is close to the target $\boldsymbol{y}_{target}$ for all inputs $\boldsymbol{x}_{input}$.

To measure how far we are from the goal, we use an error function E. A commonly used error function is $E(\boldsymbol{y}_{output}, \boldsymbol{y}_{target}) = \frac{1}{2}(\boldsymbol{y}_{output} - \boldsymbol{y}_{target})^2$.



Forward propagation

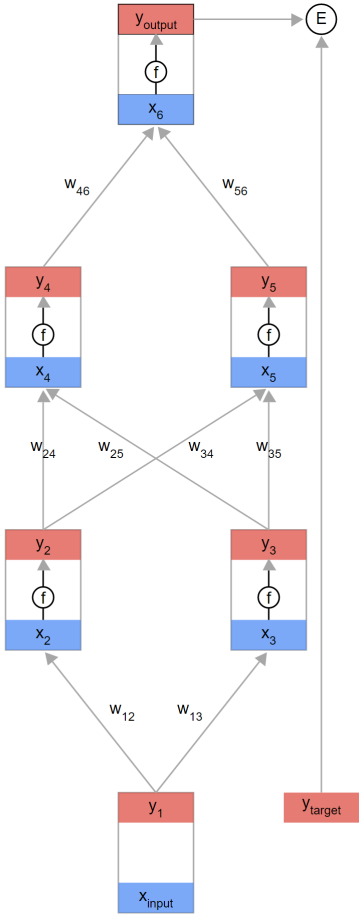
We begin by taking an input example $(\boldsymbol{x}_{input}, \boldsymbol{y}_{target})$ and updating the input layer of the network.

For consistency, we consider the input to be like any other node but without an activation function so its output is equal to its input, i.e. $\boldsymbol{y}_1 = \boldsymbol{x}_{input}$.

Using these 2 formulas we propagate for the rest of the network and get the final output of the network.

$$\boldsymbol{y} = f(\boldsymbol{x})$$

$$\boldsymbol{x}_j = \sum_{i \in in(j)} w_{ij} \cdot \boldsymbol{y}_i + b_j$$



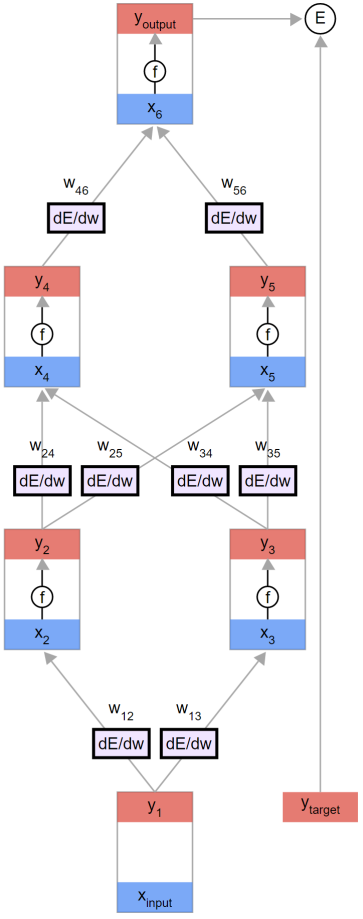
Error derivative

The backpropagation algorithm decides how much to update each weight of the network after comparing the predicted output with the desired output for a particular example. For this, we need to compute how the error changes with respect to each weight $\frac{dE}{dw_{ij}}$.

Once we have the error derivatives, we can update the weights using a simple update rule:

$$w_{ij} = w_{ij} - \alpha \frac{dE}{dw_{ij}}$$

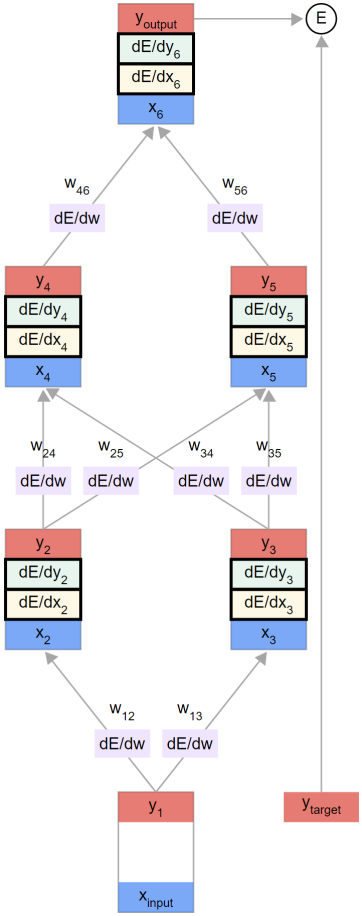
where α is a positive constant, referred to as the **learning rate**, which we need to fine-tune empirically. The update rule is very simple: if the error goes down when the weight increases ($\frac{dE}{dw_{ij}} < 0$), then increase the weight, otherwise if the error goes up when the weight increases ($\frac{dE}{dw_{ij}} > 0$), then decrease the weight.



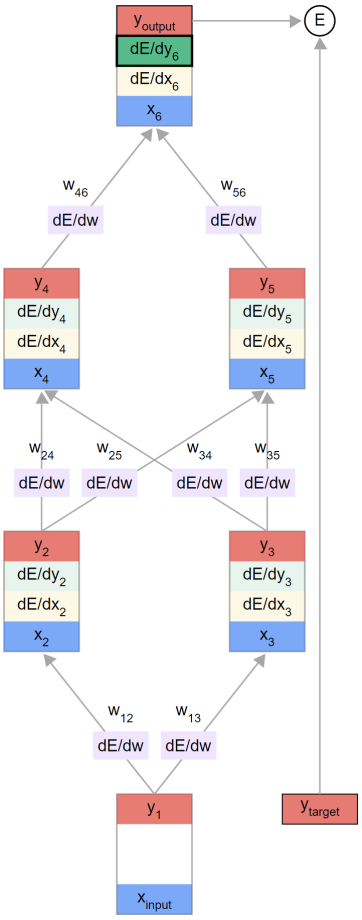
Additional derivatives

To help compute $\frac{dE}{dw_{ij}}$, we additionally store for each node two more derivatives: how the error changes with:

- the total input of the node $\frac{dE}{dx}$ and
- the output of the node $\frac{dE}{dy}$.



Back propagation



Let's begin backpropagating the error derivatives. Since we have the predicted output of this particular input example, we can compute how the error changes with that output.

Given our error function

$$E(\boldsymbol{y}_{output}, \boldsymbol{y}_{target}) = \frac{1}{2}(\boldsymbol{y}_{output} - \boldsymbol{y}_{target})^2$$

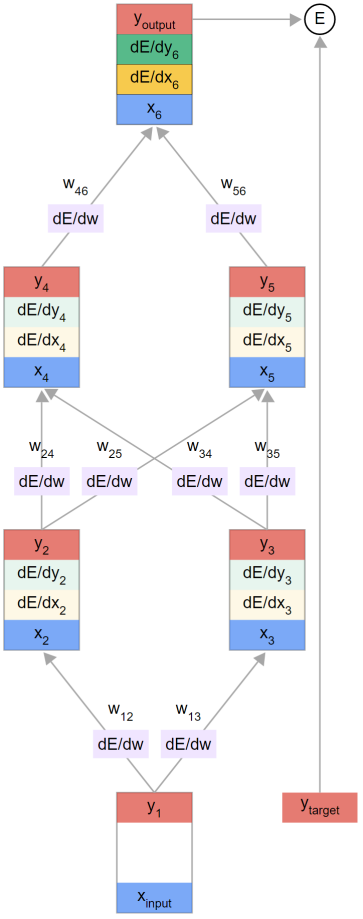
$$\frac{\partial E}{\partial \boldsymbol{y}_{output}} = \boldsymbol{y}_{output} - \boldsymbol{y}_{target}$$

Back propagation

Now that we have $\frac{dE}{dy}$ we can get $\frac{dE}{dx}$ using the chain rule.

$$\frac{\partial E}{\partial x} = \frac{dy}{dx}\frac{\partial E}{\partial y} = \frac{d}{dx}f(x)\frac{\partial E}{\partial y}$$

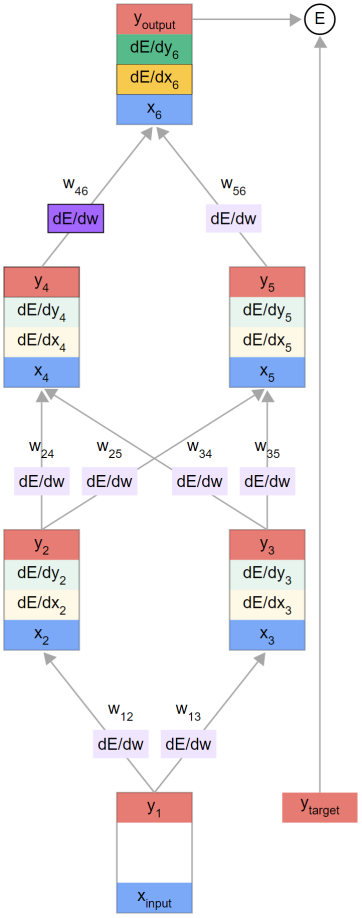
where $\frac{d}{dx}f(x) = f(x)(1 - f(x))$ when $f(x)$ is the Sigmoid activation function.



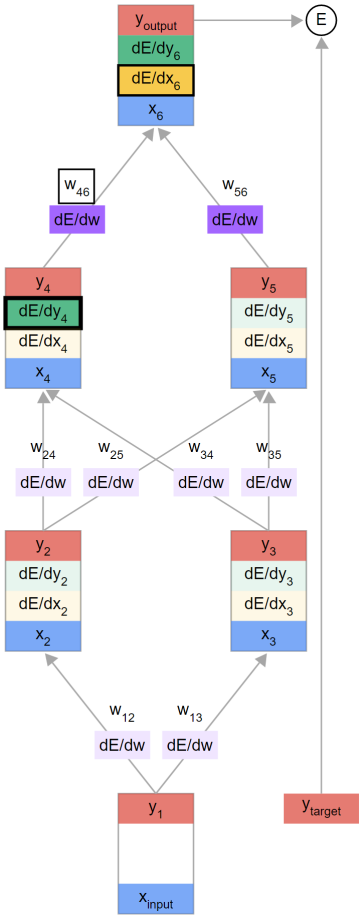
Back propagation

As soon as we have the error derivative with respect to the total input of a node, we can get the error derivative with respect to the weights coming into that node.

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial x_j}{\partial w_{ij}} \frac{\partial E}{\partial x_j} = y_i \frac{\partial E}{\partial x_j}$$



Back propagation



And using the chain rule, we can also get $\frac{dE}{dy}$ from the previous layer. We have made a full circle.

$$\frac{\partial E}{\partial y_i} = \sum_{j \in \text{out}(i)} \frac{\partial x_j}{\partial y_i} \frac{\partial E}{\partial x_j} = \sum_{j \in \text{out}(i)} w_{ij} \frac{\partial E}{\partial x_j}$$

All that is left to do is repeat the previous three formulas until we have computed all the error derivatives.

Bibliography

Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data A short course*. AML publishers, 2012.

David Arthur and Sergei Vassilvitskii. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry*, pages 144–153, 2006.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer, 2017.

Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.