DSA
Data Science Africa

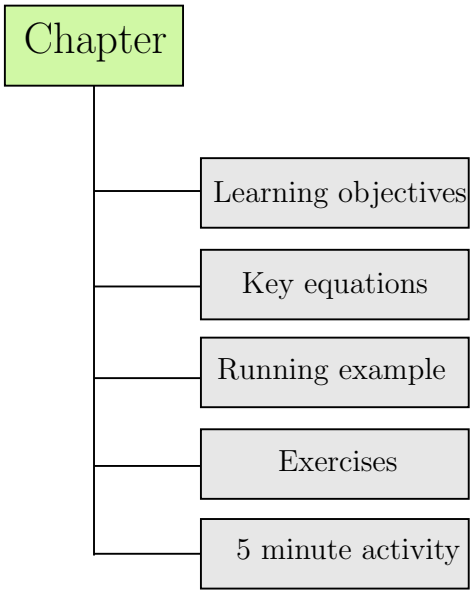
DSA 2026
Machine Learning Fundamentals

Harry D. Mafukidze

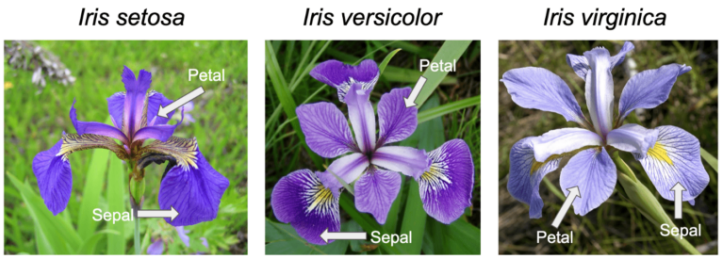
Contents

Preface

These notes are designed for an introductory course in machine learning. We assume basic Python and neural network knowledge - see Appendix. Each chapter includes learning objectives, key equations, exercises, and a running example. Special emphasis is placed on the machine learning pipeline and generalization problem – some of the biggest challenges in ML.



Running Example: Iris Flower Classification



Throughout these notes, we shall use the Iris dataset as a running example to illustrate the main stages of a machine learning workflow. The Iris dataset contains measurements of iris flowers, including sepal length, sepal width, petal length, and petal width. The goal is to classify each flower into one of three species: *setosa*, *versicolor*, or *virginica*. This is a supervised classification problem because the input features are known and the correct class labels are provided.

1 Machine learning pipeline

Learning Objectives

- Describe the seven stages of an ML pipeline.
- Explain why problem definition and data preparation are critical.
- Distinguish between training, validation, and test sets.

A machine-learning pipeline is a structured sequence of processes that takes raw data through a chain of transformation and decision-making to produce a deployed machine-learning model. These processes include data acquisition, cleaning, feature engineering, model training, evaluation, deployment, and continuous monitoring. Unlike traditional data pipelines, which only move and transform data, ML pipelines incorporate model-specific tasks such as training and inference, ensuring that data science efforts translate into production-ready solutions.

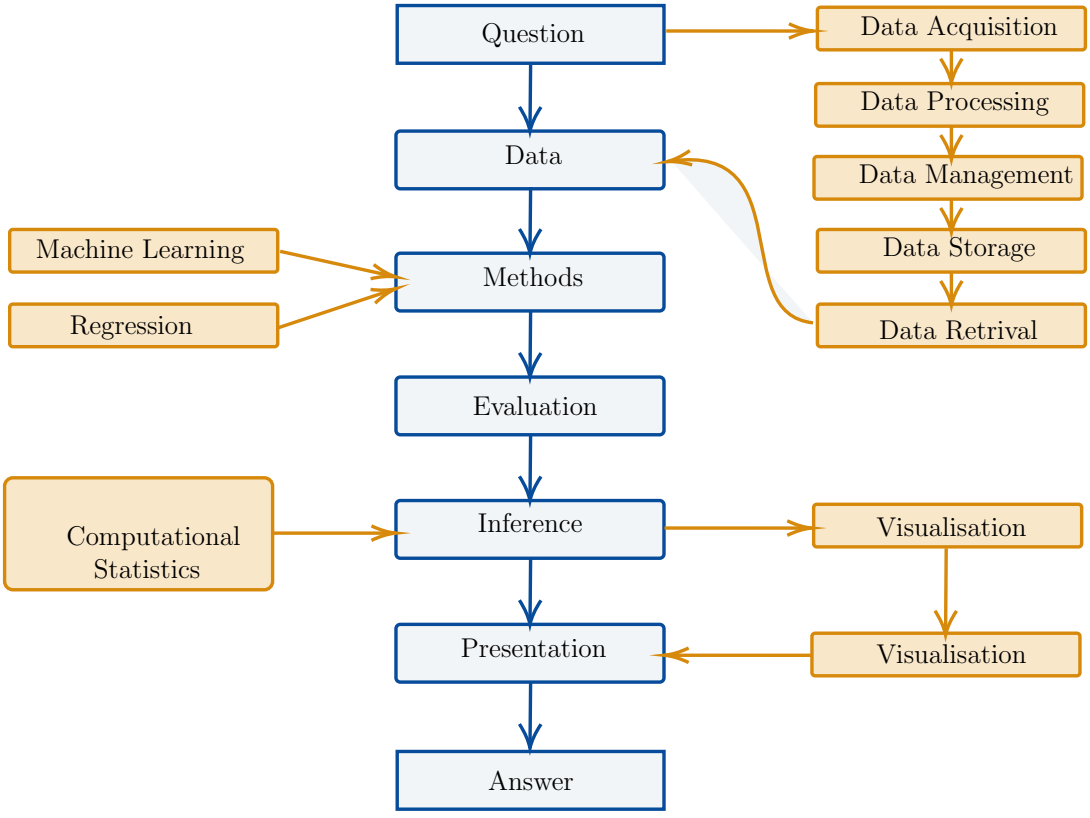


Figure 1: ML pipeline.

1.1 Problem Definition

The process of data analysis actually begins long before the collection of raw data. In fact, a data analysis always starts with a problem to be solved, which needs to be defined. The problem is defined only after you have well-focused the system you want to study: this may be a mechanism, an application, or a process in general. Generally this study

can be in order to better understand its operation, but in particular the study will be designed to understand the principles of its behavior in order to be able to make predictions, or to make choices (defined as an informed choice).
The definition step and the corresponding documentation (deliverables) of the scientific problem or business are both very important in order to focus the entire analysis strictly on getting results. Once the problem has been defined and documented, you can move to the project planning of a data analysis.

1.2 Data Extraction and Data Collection

Once the problem has been defined, the first step is to obtain the data in order to perform the analysis. The data must be chosen with the basic purpose of building the ML model, and so their selection is crucial for the success of the analysis as well. The sample data collected must reflect as much as possible the real world, that is, how the system responds to stimuli from the real world. In fact, even using huge data sets of raw data, often, if they are not collected competently, these may portray false or unbalanced situations compared to the actual ones. Thus, a poor choice of data, or even performing analysis on a data set which is not perfectly representative of the system, will lead to models that will move away from the system under study.

1.3 Data Preparation

Among all the steps involved in data analysis, data preparation, though seemingly less problematic, is in fact one that requires more resources and more time to be completed. The collected data are often collected from different data sources, each of which will have the data in it with a different representation and format. So, all of these data will have to be prepared for the process of data analysis.
The preparation of the data is concerned with obtaining, cleaning, normalizing, and transforming data into an optimized data set, that is, in a prepared format, normally tabular, suitable for the methods of analysis that have been scheduled during the design phase. Many are the problems that must be avoided, such as invalid, ambiguous, or missing values, replicated fields, or out-of-range data.

This step consumes 60–80% of project time. Key tasks:

- Handling missing values (imputation, deletion).
- Encoding categorical variables (one-hot, label encoding).
- Feature scaling – crucial for gradient-based methods.
- Train/validation/test split (typically 70/15/15 or 80/10/10).

1.4 Data Exploration/Visualization

Exploring the data is essentially the search for data in a graphical or statistical presentation in order to find patterns, connections, and relationships in the data. Data visualization is the best tool to highlight possible patterns.

In recent years, data visualization has been developed to such an extent that it has become a real discipline in itself. In fact, numerous technologies are utilized exclusively for the display of data, and equally many are the types of display applied to extract the best possible information from a data set. Data exploration consists of a preliminary examination of the data, which is important for understanding the type of information that has been collected and what they mean. In combination with the information acquired during the definition problem, this categorization will determine which method of data analysis will be most suitable for arriving at a model definition. Generally, this phase, in addition to a detailed study of charts through the visualization data, may consist of one or more of the following activities:

- Summarizing data
- Grouping data
- Exploration of the relationship between the various attributes
- Identification of patterns and trends
- Construction of regression models
- Construction of classification models

1.5 Predictive Modeling

Predictive modeling is a process used in data analysis to create or choose a suitable statistical model to predict the probability of a result. After exploring data you have all the information needed to develop the mathematical model that encodes the relationship between the data. These models are useful for understanding the system under study. It is possible to divide the models according to the type of result that they produce, such as :

- **Regression**: numeric output (e.g., price, temperature).
- **Classification**: categorical output (e.g., spam/not spam).
- **Clustering**: no labels, find groups.

Simple methods to generate these models include techniques such as linear regression, logistic regression, classification and regression trees, and k-nearest neighbors. But the methods of analysis are numerous, and each has specific characteristics that make it excellent for some types of data and analysis.

1.6 Model Validation

Validation of the model, that is, the test phase, is an important phase that allows you to validate the model built on the basis of starting data. That is important because it allows you to assess the validity of the data produced by the model by comparing them directly with the actual system. But this time, you are coming out from the set of starting data on which the entire analysis has been established.
Generally, you will refer to the data as the **training set**, when you are using them for

building the model, and as the **validation set**, when you are using them for validating the model. Thus, by comparing the data produced by the model with those produced by the system you will be able to evaluate the error, and using different test datasets, you can estimate the limits of validity of the generated model. In fact the correctly predicted values could be valid only within a certain range, or have different levels of matching depending on the range of values taken into account.
This process allows you not only to numerically evaluate the effectiveness of the model but also to compare it with any other existing models. There are several techniques in this regard; the most famous is the **cross-validation**. This technique is based on the division of the training set into different parts. Each of these parts, in turn, will be used as the validation set and any other as the training set. In this iterative manner, you will have an increasingly perfected model.

1.7 Deployment

This is the final step of the analysis process, which aims to present the results, that is, the conclusions of the analysis. In the deployment process, in the business environment, the analysis is translated into a benefit for the client who has commissioned it. In technical or scientific environments, it is translated into design solutions or scientific publications. That is, the deployment basically consists of putting into practice the results obtained from the data analysis.

1.8 Common pitfalls to look out for

1.8.1 Data leakage

Data leakage occurs when a model accidentally uses information during training that would not be available during real-world predictions. This also occurs when testing data are leaked (directly or indirectly, deliberately or unintentionally) to the training process, leading to unrealistically optimistic performance during testing but poor results in practice, like a student cheating on a practice exam but failing the real one.

1. Target Leakage
    - **What Happens?** The model uses features that are only available after the target (prediction) is known.
    - **Example**: A hospital builds a model to predict diabetes, using a feature like "insulin treatment." But insulin is only prescribed after a diabetes diagnosis. The model "cheats" by using this post-diagnosis data.
    - **Solution**: Remove features that are influenced by or created after the target. Use domain knowledge to filter them out.
2. Train-Test Contamination
    - **What Happens?** Test data "leaks" into the training process, making the model overfit.
    - **Example**: Normalizing (scaling) a dataset before splitting into train/test sets. The test data's distribution influences the training, leading to false confidence.

- **Solution**: Always split data first into train/test sets. Preprocess (e.g., scale) using only training data statistics.

3. Preprocessing Leakage

   - Arises when preprocessing steps like scaling, normalization or encoding are applied before splitting the dataset. This allows information from the test set to influence the training process.
   - **Solution**: Use pipelines: compute means/medians from the training set only, then apply them to the test set.

4. Time-Based Leakage

   - Time-Based Leakage
     (a) **What Happens?** Future data is used to predict past events, violating the time sequence.
     (b) **Example**: Predicting stock prices with a model trained on data from 2020–2023 but tested on 2019 data. The model sees the "future" during training.
     (c) **Solution**: Avoid shuffling time-series datasets.

1.8.2 Class imbalance

Class imbalance occurs when some classes in a dataset appear much more frequently than others. This is a common problem in real-world machine learning applications. For example, a Health dataset can have 95% healthy class and 5% diabetic class. With imbalanced data, standard metrics like accuracy might not make sense. For example, a classifier that always predicts "Healthy" would have 95% accuracy in detecting health status. Over-sampling the minority class and under-sampling or removing data points from the majority class can help when working with unbalanced datasets.

1.8.3 Poor train-test splitting

Train-test splitting is the process of dividing a dataset into **a training set** used to train the model, and a **test set** used to evaluate performance on unseen data. Poor train-test splitting can produce misleading evaluation results and incorrect conclusions about model performance and generalisation ability. Poor splitting can accidentally allow information from the test set to appear in the training process. Sometimes duplicate or nearly identical samples appear in both training and test sets.

- Split before pre-processing
- Avoid duplicate samples
- Use temporal splits for time-series data

Five-minute activity: Map the Pipeline Stages

1. Suppose you are asked to build a model using the Iris dataset, but no problem statement, objective, or stakeholder requirements are provided. What business value or scientific questions could we draw from this dataset?

2. Suppose the Iris dataset was collected from only one geographic region:

   (a) What hidden biases might be introduced?

   (b) Would a model trained on such data generalize to flowers from other regions?

   (c) How could you test whether the collected data truly represent the real-world population?

3. *Lets vote:* Which stage of the ML pipeline is ***most*** responsible for project failure?

   (a) Problem Definition

   (b) Data Collection

   (c) Data Preparation

   (d) Model Selection

   (e) Validation

   (f) Deployment

2 Functions

Learning Objectives

- Explain why functions improve code readability and maintainability.
- Convert repetitive machine-learning code into reusable functions.
- Create functions that accept arguments and return outputs.
- Build reusable preprocessing functions for loading Iris, feature scaling, and training a logistic regression model.

This section discusses functions in Python, and assumes the reader has knowledge on the basic structure of a Python code, running Python, variables, expressions, and statements.

In the context of programming, a function is a named sequence of statements that perform a computation. When you define a function, you specify the name and the sequence of statements. Later, you can "call" the function by name. If you need to perform that task multiple times throughout your program, you don't need to type all the code for the same task again and again; you just call the function dedicated to handling that task, and the call tells Python to run the code inside the function. You'll find that using functions makes your programs easier to write, read, test, and fix.

2.1 Why functions

1. Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.
2. Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
3. Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
4. Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

```
>>> print("Hello DSA 2026")
<Hello DSA 2026>

>>> type(50)
<class 'int'>

>>> abs(10)
<10>
```

The function has a *name*, and the expression in parentheses is called the *argument* of the function. It is, therefore, common to say that a function *"takes"* an argument and *"returns"* a result. The result is also called the return value.

2.2 In-built Python functions

Python provides a rich set of built-in functions that are always available for use. These functions perform a variety of tasks and can be used on different data types such as strings, lists, dictionaries, tuples, sets, and more. There is a comprehensive list of some of the most commonly used built-in functions in Python[1]

```
>>> type() #Returns the type of an object
>>> abs() #Returns the absolute value of a number
>>> min() #Returns the smallest item in an iterable
>>> max() #Returns the highest item in an iterable
>>> len() #Returns the length of an object
>>> hex() #Converts a number into a hexadecimal value
```

> Always check for parameters of the function.

2.3 User-defined functions

So far, we have only been using the functions that come with Python, but it is also possible to add new functions. A *function definition* specifies the name of a new function and the sequence of statements that run when the function is called. To define a function in Python, use the ***def*** keyword to declare the function name. The general syntax for defining a function is as follows:

```
def functionName([parameter1, parameter2, ..., parameterN]):

  """Documentation for the function."""

  <block_of_instructions>
```

In the function definition, the first string of characters (called a *docstring*) serves as documentation for the function, accessible via the interpreter using, for example, *help(functionName)*, or functionName? in Jupyter. It should be relevant, concise, and comprehensive. It may also include usage examples.

2.3.1 Definition of a simple function without arguments

Here's a simple function named *greet_members()* that prints a greeting.

```
def greet_members():
  """Greets members attending the conference."""
  print("Hello members!")


greet_members()
```

1. *def* is a keyword that indicates that this is a function definition. The name of the function is *greet_members*.

2. The empty parentheses after the name indicate that this function doesn't take any arguments.

3. The first line of the function definition is called the header, and the rest is called the body. The header has to end with a colon and the body has to be indented.

4. The strings in the print statements are enclosed in double quotes. Single quotes and double quotes do the same thing

5. The syntax for calling the new function is the same as for built-in functions.

2.3.2 Definition of a simple function with one or multiple arguments

Some of the functions we have seen require *arguments*. In other words, information is passed on to the function. For example, when you call *math.sin* you pass a number as an argument. Some functions take more than one *argument*: *math.pow* takes two, the base and the exponent. Inside the function, the arguments are assigned to variables called parameters. Here is a definition for a function that takes an argument

```
def print_twice(bruce):
  print(bruce)
  print(bruce)
```

This function assigns the argument to a parameter named bruce. When the function is called, it prints the value of the parameter (whatever it is) twice.

```
>>>print_twice('DSA 26')
<DSA 26>
<DSA 26>

>>>print_twice(27)
<27>
<27>
```

The same rules of composition that apply to built-in functions also apply to user defined functions. In the preceding print_twice() function, we defined print_user() to require a value for the variable bruce. Once we called the function and gave it the information, it printed that information twice. The variable bruce in the definition of print_user() is an example of a parameter, a piece of information the function needs to do its job. The value 'DSA 26' in print_twice('DSA 26') is an example of an argument. An argument is a piece of information that is passed from a function call to a function. When we call the function, we place the value we want the function to work with in parentheses. In this case the argument 'DSA 26' was passed to the function print_twice(), and the value was stored in the parameter bruce.

2.4 Return values

A function doesn't always have to display its output directly. Instead, it can process some data and then return a value or set of values. The value the function returns is called a *return value*. The return statement takes a value from inside a function and sends it back to the line that called the function. Return values allow you to move much of your program's grunt work into functions, which can simplify the body of your program.

2.4.1 Single-return values

Let's look at a function that takes the length of a side a square and returns the area of a square.

```
def squareArea(l):
  """
  Calculate the area of a square given the length of its side.

  Args:
  l (float): The length of the side of the square.

  Returns:
  float: The area of the square.
  """
  return l**2

# Input for the side length and display of the area
length = float(input('length: '))
print("Square area =", squareArea(length))
```

2.4.2 Multiple-return values

A function can also return multiple values as shown below:

```
 Pi = 3.14

def surfaceVolumeSphere(r):
  """
  Calculate the surface area and volume of a sphere.

  Args:
  r (float): The radius of the sphere.

  Returns:
  tuple: A tuple containing the surface area and volume of the
     sphere.
  """
  surf = 4.0 * Pi * r**2
  vol = surf * r / 3
  return surf, vol

# Main program
radius = float(input('Radius: '))
s, v = surfaceVolumeSphere(radius)
print(f"Sphere with surface {s:.2f} and volume {v:.2f}".format(s,
    v))
```

From the above script, the function *surfaceVolumeSphere*() takes the radius r of a sphere as an argument and calculates the surface area and the volume of the sphere using $4\pi r^2$ and $\frac{4}{3}\pi r^3$ respectively. The line *s, v = surfaceVolumeSphere(radius)* calls

the function *surfaceVolumeSphere(radius)* and stores the two returned values into the variables *s* and *v*.

function	A named sequence of statements that performs some useful operation. Func tions may or may not take arguments and may or may not produce a result.
function definition	A statement that creates a new function, specifying its name, parameters, and the statements it contains.
function object	A value created by a function definition. The name of the function is a variable that refers to a function object.
header	The first line of a function definition.
body	The sequence of statements inside a function definition.
parameter	A name used inside a function to refer to the value passed as an argument.
function call	A statement that runs a function. It consists of the function name followed by an argument list in parentheses.
argument	A value provided to a function when the function is called. This value is as signed to the corresponding parameter in the function.
local variable	A variable defined inside a function. A local variable can only be used inside its function.
return value	The result of a function. If a function call is used as an expression, the return value is the value of the expression
fruitful function	A function that returns a value.
void function	A function that always returns None
module	A file that contains a collection of related functions and other definitions
import statement	A statement that reads a module file and creates a module object.
module object	A value created by an import statement that provides access to the values defined in a module.
dot notation	The syntax for calling a function in another module by specifying the mod ule name followed by a dot (period) and the function name.

Five-minute activity: Reusable Functions

1. Which lines of code would you need to repeat if you wanted to train three different models, such as LogisticRegression, DecisionTree, KNN)?
2. What would be the risk of copy-pasting those lines instead of using functions? *NB: data leakage or inconsistent scaling.*
3. Write the following functions in a new cell using Python or Pseudocode:
   (a) *def scale_data(X_train, X_test):*
      - Takes training and test sets
      - Fits a *StandardScaler* on the training set
      - Returns scaled training set, scaled test set, and the fitted scaler

   (b) *def train_and_evaluate(model_class, X_train, y_train, X_test, y_test):*
      - Takes a *LogisticRegression* model
      - Instantiates it with default parameters
      - Trains it on the scaled training data
      - Returns the test accuracy

   (c) Call your functions to reproduce the original accuracy.

3 Model Evaluation

Learning Objectives

- Compute accuracy, precision, recall, F1, and confusion matrix.
- Explain why accuracy can be misleading for imbalanced data.
- Implement k-fold cross-validation.

Model evaluation is the process of assessing how well a machine learning model performs on unseen data using different metrics and techniques. It ensures that the model not only memorises training data but also generalises to new situations. By applying various techniques, we can identify whether a model has truly learned patterns or not.

3.1 Cross-Validation

Cross-validation ensures that the model is tested on multiple subsets of data making it less likely to overfit and improving its generalisation ability. The dataset is split into train and test sets (commonly 7:3 or 8:2).

3.2 K-Fold Cross Validation

In K-Fold Cross Validation, the dataset is divided into k equally sized folds. The model is trained on $k-1$ folds and validated on the remaining fold. This process repeats until every fold has been used once as a validation set.

The final model performance is computed as the average across all folds:

$$Accuracy_{avg} = \frac{1}{k} \sum_{i=1}^{k} Accuracy_i$$

This approach reduces overfitting and provides a more reliable estimate of model performance.

3.3 Evaluation Metrics for Classification Tasks

Classification models assign inputs to predefined labels. Their performance can be measured using *accuracy*, *precision*, *recall*, *F1 score*, *confusion matrix* and *AUC-ROC*.

3.3.1 Accuracy

Accuracy measures the overall correctness of a classification model. A high accuracy means the model predicts most classes correctly.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

where
TP = True positives

TN = True Negatives
FP = False Positives
FN = False Negatives

3.3.2 Precision

Precision measures how many predicted positive cases are actually positive. High precision means the model produces few false positives.

$$Precision = \frac{TP}{TP + FP}$$

3.3.3 Recall

Recall measures how many actual positive cases are correctly identified. High recall means few false negatives.

$$Recall = \frac{TP}{TP + FN}$$

3.3.4 F1-score

The F1-score combines precision and recall into a single metric using the harmonic mean. A high F1-score indicates a good balance between precision and recall.

$$F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

3.3.5 Confusion Matrix

A confusion matrix is a table used to measure how well a classification model is performing. It compares the predictions made by the model with the actual results and shows where the model was right or wrong. This helps you understand where the model is making mistakes so you can improve it.

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Figure 2: Confusion matrix.

An example
Imagine we have a *binary classification* model that predicts whether a patient has a certain disease (Positive) or not (Negative). We test the model on 145 patients. The results are summarised in the confusion matrix below:
From this example

	Predicted Positive	**Predicted Negative**
Actual Positive	TP = 50	FN = 5
Actual Negative	FP = 10	TN = 80

- True Positives (TP) = 50: The model correctly said "disease" to 50 patients who actually have the disease.
- False Negatives (FN) = 5: The model missed 5 patients who have the disease (predicted "no disease").
- False Positives (FP) = 10: The model incorrectly said "disease" to 10 healthy patients.
- True Negatives (TN) = 80: The model correctly said "no disease" to 80 healthy patients.

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{50}{50 + 10} = \frac{50}{60} \approx .0.833$$

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{50}{50 + 5} = \frac{50}{55} \approx .0.909$$

$$F_1 = 2 * \frac{\text{Precision * Recall}}{\text{Precision + Recall}} = 2 * \frac{0.833 * 0.909}{0.800 + 0.909} = \approx .0.869$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{50 + 80}{145} = \approx .0.897$$

Interpreting the results
Precision (0.833) – When the model predicts "disease", it is correct about 83% of the time.

Recall (0.909) – The model finds 91% of all actual diseased patients (only 5 missed).

F1-score (0.869) – A balanced measure of precision and recall.

3.4 More Evaluation Metrics

ROC-AUC and PR-AUC are two critical metrics for evaluating binary classification models, especially in imbalanced datasets. While both provide valuable insights about model performance, they have distinct characteristics and are suited for different scenarios. These metrics provide insights into different elements of model performance, such as the trade-off between precision and recall, the ability to handle imbalanced datasets, and the ability to classify samples correctly.

3.4.1 ROC-AUC (Receiver Operating Characteristic — Area Under Curve)

ROC AUC measures a model's ability to distinguish between positive and negative classes across different classification thresholds. These are ML metrics used to evaluate the performance of binary classification models. The ROC curve is a two-dimensional plot of the true positive rate against the false positive rate, showing the trade-off of both axes at various thresholds. The ROC curve is a plot of the true positive rate (TPR) against the false positive rate (FPR), as defined in Equation 1 and Equation 2 at various threshold settings, and it is created by varying the threshold to predict a positive or negative outcome and plotting the TPR against the FPR for each threshold. The TPR is the proportion of actual positive samples that are correctly identified as positive by the model.

$$TPR = \frac{TP}{TP + FN} \tag{1}$$

$$FPR = \frac{FP}{FP + TN} \tag{2}$$

In contrast, the FPR is the proportion of actual negative samples that are incorrectly identified as positive by the model. In the figure below, each colored line represents the ROC curve of a different binary classifier system. The axes represent the FPR and TPR. The diagonal line represents a random classifier, while the top-left corner represents a perfect classifier with TPR=1 and FPR=0. The AUC is the area under the curve made by the ROC curve. The AUC formula is to integrate the area under the ROC curve using the trapezoidal rule.

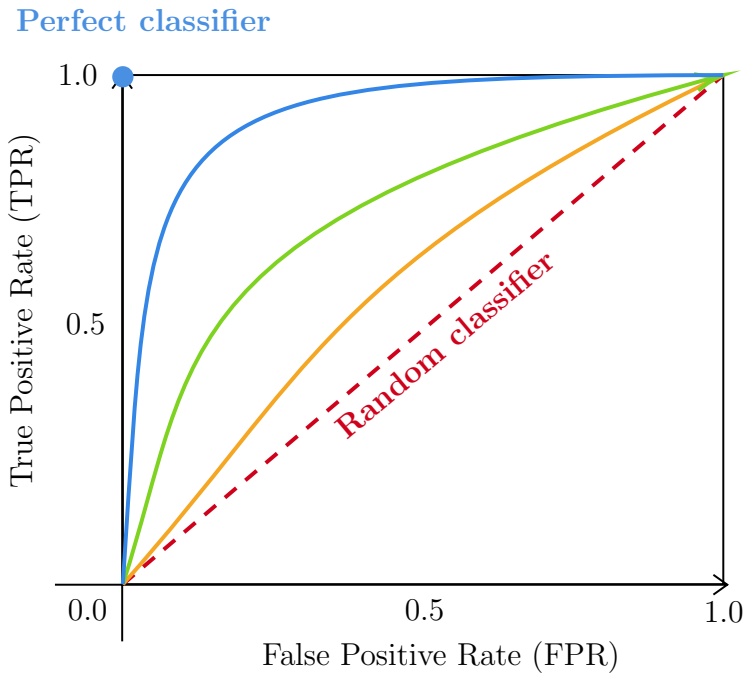


Figure 3: ROC curve.

The AUC (Area Under the Curve) summarizes the ROC curve into a single number between 0 and 1, where:

- 1.0 (100%) – Perfect classifier (completely separates positive and negative classes).
- 0.5 (50%) – Random guessing (no better than chance).
- < 0.5 – Worse than random (can be fixed by inverting predictions).

Key Characteristics of ROC-AUC

- Threshold-independent: Evaluates performance across all possible classification thresholds
- Scale-invariant: Focuses on the ranking of predictions rather than absolute values
- Insensitive to class imbalance in calculation: The curve shape doesn't change with class distribution
- Intuitive interpretation: Higher AUC = better separation between classes

When to Use ROC-AUC

- When both classes are equally important
- When the dataset is relatively balanced
- When false positives and false negatives have similar costs
- When you need a general overview of model performance

3.4.2 PR-AUC (Precision-Recall Area Under the Curve)

PR-AUC is an ML metric used to evaluate the performance of binary classification models, mainly when the classes are imbalanced. Unlike the ROC curve and AUC, which plot the TPR against the FPR, the PR curve plots the precision against the recall at different threshold settings, Equation 3 and 4, where its value is the integration of the area under the curve.

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

Precision is the proportion of true positive predictions out of all positive predictions made by the model, while recall is the proportion of true positive predictions from all actual positive samples in the dataset. The PR curve is created by varying the threshold for predicting a positive or negative outcome and plotting the precision against the recall for each threshold.

Unlike ROC-AUC, the baseline for PR-AUC depends on class distribution:

- In a balanced dataset (50% positive), random guessing gives PR-AUC = 0.5

- In an imbalanced dataset (1% positive), random guessing gives PR-AUC = 0.01

Key Characteristics of PR-AUC

- Focuses on the positive class: Particularly important for imbalanced datasets
- Sensitive to class imbalance: Reflects the challenge of identifying rare positive instances
- More informative in imbalanced scenarios: Can reveal issues that ROC-AUC might hide
- Directly relates to business objectives: Precision and recall often map directly to business costs

When to Use PR-AUC

- On imbalanced datasets
- When identifying positive instances is critical (e.g., fraud detection, medical diagnosis)
- When false positives and false negatives have different costs
- When you care more about the quality of positive predictions than overall accuracy

These metrics are important, especially for African data contexts where the datasets may be imbalanced, such as disease detection, fraud, crop disease, and low-resource health applications.

3.5 Python Example: Classification Evaluation

3.6 Regression Metrics

For regression, common metrics are:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{6}$$

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{7}$$

Five-minute activity: Model Evaluation

Using the Iris notebook, train a LogisticRegression model, and then:

1. Print the confusion matrix for the test set.
2. For class "Versicolour" (label 1) , compute by hand or using code the following, aand compare your numbers with *classification_report* output:
   (a) Precision
   (b) Recall
   (c) F1-score.
3. Suppose a botanist wants to ensure that every Versicolor flower is identified correctly, even if some other flowers are incorrectly labelled as Versicolor, which metric should be prioritized between **Accuracy**, **Precision**, **Recall**, and **F1 Score**.

4 Optimization

Learning Objectives

- Define a loss function and compute gradients.
- Compare Batch, Stochastic, and Mini-batch Gradient Descent.
- Explain the role of learning rate and momentum.

The optimization challenge is to find the optimal configuration of our models' parameters that minimizes loss.

Hyperparameters are configuration settings defined before training a machine learning model. Unlike model parameters, which are learned automatically from data during training, hyperparameters are chosen by the designer or via automated search, to control the learning process and model behavior. These settings influence important factors such as model complexity, convergence speed, training stability, and generalization performance. Proper hyperparameter tuning is essential because it helps minimize prediction errors while ensuring that the model performs well on unseen data. For example, in a neural network, the learning rate determines the magnitude of weight updates during each training iteration. If the learning rate is too high, training may become unstable and fail to converge. Conversely, if the learning rate is too low, convergence becomes slow, resulting in longer training times and potentially suboptimal performance (e.g., getting stuck in poor local minima). The choice of hyperparameters directly affects how well an optimization algorithm can minimize the **loss/cost function**, which quantifies the discrepancy between predictions and true values.

4.1 Cost/Loss Function

A machine learning model learns by minimizing a *clost function*, also called a *loss function*. The loss function quantifies how far the model's predictions are from the true values. During training, optimization algorithms such as Gradient Descent iteratively adjust the model parameters to minimize this loss. In general, the loss function can be expressed as

$$L(y, \hat{y})$$

where y is the true target value and $\hat{y}$ is the predicted value produced by the model.

The choice of loss function depends on the machine learning task being solved.

4.2 Mean Squared Error (MSE) for Regression

For regression problems, one of the most commonly used loss functions is the *Mean Squared Error* (MSE):

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

where:

N is the number of training examples,
(y_i) is the actual value,
($\hat{y}_i$ is the predicted value.

MSE penalizes large prediction errors more heavily because the errors are squared. A lower MSE indicates better predictive performance.

Example

Suppose the true values are:

$$[10, 20, 30]$$

and the model predicts:

$$[12, 18, 33]$$

Then

$$MSE = \frac{(10-12)^2 + (20-18)^2 + (30-33)^2}{3} = \frac{4+4+9}{3} = 5.67$$

4.3 Cross-Entropy Loss for Classification

For classification problems, a commonly used loss function is *Cross-Entropy Loss*.

For binary classification:

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

where:

$y_i \in 0, 1$ is the true class label, $\hat{y}_i$ is the predicted probability of the positive class.

Cross-entropy penalizes incorrect predictions based on their confidence. Predictions that are confidently wrong incur a large penalty, encouraging the model to produce accurate probability estimates.

4.4 Why Loss Functions Matter

The loss function defines the objective that the learning algorithm seeks to optimize. During training, Gradient Descent computes the gradient of the loss function with respect to the model parameters and updates the parameters in the direction that reduces the loss. Consequently, selecting an appropriate loss function is critical because it directly influences the learning behaviour and final performance of the model.

4.5 Learning Rate

The learning rate η is a crucial hyperparameter in gradient descent. It controls the step size for each parameter update and determines how aggressively the model moves toward minimizing the cost function C at each iteration.

The learning rate controls how large a step the optimizer takes when moving downhill on the loss surface

With a small learning rate (η too low), the machine learning model updates its parameters in very small steps, leading to slow convergence. If there is insufficient momentum, it may take excessive iterations to reach the minimum or get stuck in local minima.
With a high learning rate (η too high), the model takes big steps, which can cause it to overshoot the minimum and oscillate without settling. The loss might even diverge in extreme cases of large learning rates, preventing convergence.

While a single parameter gives us a nice curve, most machine learning models have at least two, and often many more, adjustable parameters. With two parameters, our loss function transforms from a curve into a three-dimensional surface, which we call the loss landscape, as shown in Figure 4. Each point on this landscape represents a unique combination of our two parameters, and its height indicates the corresponding loss value. The goal remains the same, to navigate this landscape and find the lowest point, the deepest valley, where the loss is minimized.

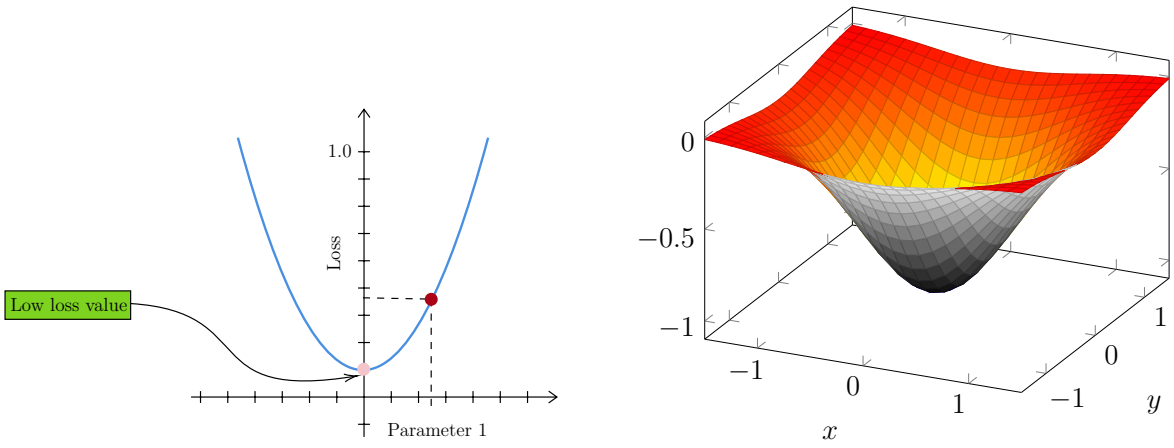


Figure 4: Visualizing the cost function. The primary objective of optimization is to find the model parameters that minimize a chosen loss function computed on the training data.

Gradient Descent is one of the most widely used optimization algorithms in machine learning and deep learning. It helps models minimize the cost function by updating parameters step by step. Two widely used variants of Gradient Descent are Batch Gradient Descent and Stochastic Gradient Descent (SGD). These variants differ mainly in how they process data and optimize the model parameters.

4.6 Batch Gradient Descent

Batch Gradient Descent computes the gradient of the cost function using the entire training dataset for each iteration. This approach ensures that the computed gradient is precise, but it can be computationally expensive when dealing with very large datasets.

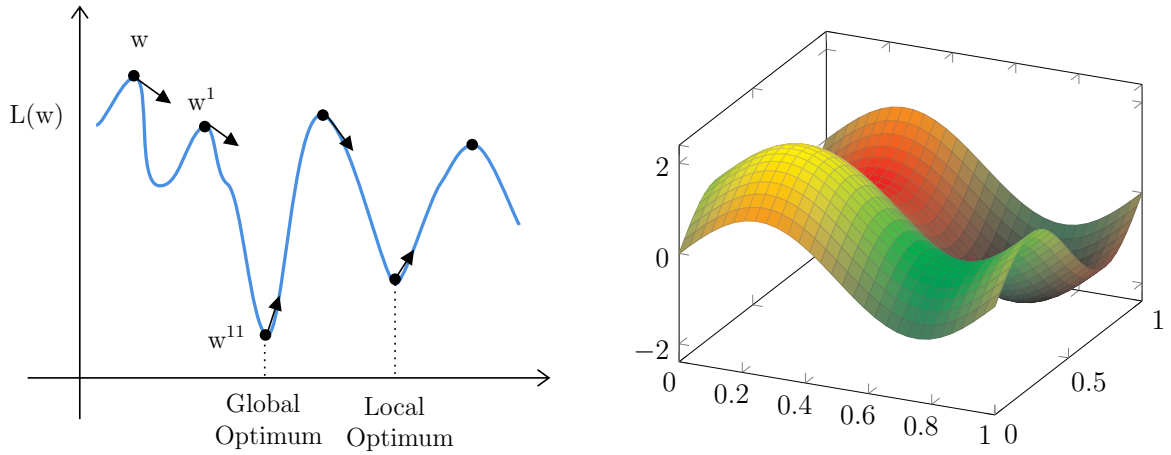


Figure 5: The optimizer looks to minimize the loss function by tuning the weights of the neurons.

Advantages

- Accurate Gradient Estimates: Since it uses the entire dataset, the gradient estimate is precise.
- Good for Smooth Error Surfaces: It works well for convex or relatively smooth error manifolds.

Disadvantages

- Slow Convergence: Because the gradient is computed over the entire dataset, it can take a long time to converge, especially with large datasets.
- High Memory Usage: Requires significant memory to process the whole dataset in each iteration, making it computationally intensive.
- Inefficient for Large Datasets: With large-scale datasets, Batch Gradient Descent becomes impractical due to its high computation and memory requirements.

When to Use Batch Gradient Descent?
Batch Gradient Descent is ideal when the dataset is small to medium-sized and when the error surface is smooth and convex. It is also preferred when we can afford the computational cost.

4.7 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) addresses the inefficiencies of Batch Gradient Descent by computing the gradient using only a single training example (or a small subset) in each iteration. This makes the algorithm much faster since only a small fraction of the data is processed at each step.

Advantages

- Faster Convergence: Since the gradient is updated after each individual data point, the algorithm converges much faster than Batch Gradient Descent.
- Lower Memory Requirements: As it processes only one data point at a time, it requires significantly less memory, making it suitable for large datasets.
- Escape Local Minima: Due to its stochastic nature, SGD can escape shallow local minima and typically aims to find a sufficiently good solution that generalises well to unseen data, especially for non-convex functions.

Disadvantages

- Noisy Gradient Estimates: Since the gradient is based on a single data point, the estimates can be noisy, leading to less accurate results.
- Convergence Issues: While SGD may converge quickly, it tends to oscillate around the minimum and does not settle exactly at the global minimum. This can be mitigated by gradually decreasing the learning rate.
- Requires Shuffling: To ensure randomness, the dataset should be shuffled before each epoch.

When to Use Stochastic Gradient Descent?
SGD is particularly useful when dealing with large datasets where processing the entire dataset at once is computationally expensive. It is also effective when optimizing non-convex loss functions.

Aspect	Batch Gradient Descent	Stochastic Gradient Descent
Data Processing	Uses the whole training dataset to compute the gradient.	Uses a single training sample to compute the gradient.
Convergence Speed	Slower, takes longer to converge.	Faster, converges quicker due to frequent updates.
Convergence Accuracy	More accurate, gives precise gradient estimates.	Less accurate due to noisy gradient estimates.
Computational and Memory Requirements	Requires significant computation and memory.	Requires less computation and memory.
Optimization of Non-Convex Functions.	Can get stuck in local minima.	Can escape shallow local minima find a sufficiently good solution that generalises well to unseen data.
Suitability for Large Datasets	Not ideal for very large datasets due to slow computation.	Can handle large datasets effectively.
Nature	Deterministic: Same result for the same initial conditions.	Stochastic: Results can vary with different initial conditions.
Learning Rate	Fixed learning rate.	Learning rate can be adjusted dynamically.
Shuffling of Data	No need for shuffling.	Requires shuffling of data before each epoch.
Overfitting	Can overfit if the model is too complex.	Can reduce overfitting due to more frequent updates.
Escape Local Minima	Cannot escape shallow local minima.	Can escape shallow local minima more easily.
Computational Cost	High due to processing the entire dataset at once.	Low due to processing one sample at a time.

4.8 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent is an optimization technique used in machine learning to update model parameters. It strikes a balance between the noise reduction of Batch Gradient Descent and efficiency the of Stochastic Gradient Descent. In this method, parameters are updated after computing the gradient of the error with respect to a subset of the training set - small subsets called mini-batches allowing the model to update its parameters more frequently compared to using the entire dataset at once.
Instead of updating weights after calculating the error for each data point (in stochastic gradient descent) or after the entire dataset (in batch gradient descent), mini-batch gradient descent updates the model's parameters after processing a mini-batch of data. This provides a balance between computational efficiency and convergence stability.

4.9 Why to use Mini-Batch Gradient Descent?

To improve both the computational efficiency and the convergence rate of the training process. By processing smaller subsets of data at a time we can update the weights more frequently than batch gradient descent while avoiding the noisiness of stochastic gradient descent.

- Batch gradient descent uses the entire dataset for each iteration which can be computationally expensive.
- Stochastic gradient descent updates the model after each training sample but this can lead to noisy updates and fluctuations in the training process.
- Mini-batch gradient descent offers a middle ground making it more efficient and often leading to faster convergence.

How Mini-Batch Gradient Descent Works

- **Splitting the Data**: The training dataset is divided into smaller mini-batches. Each mini-batch contains a subset of data points. For example if the dataset has 10,000 examples we might split it into 100 mini-batches each containing 100 data points.
- **Computing the Gradient:** For each mini-batch the gradient of the loss function is computed and used to update the model's parameters. The loss is averaged over the mini-batch which helps in reducing the noise compared to the SGD approach.
- **Updating the Parameters:** Once the gradient is computed for a mini-batch the model's parameters are updated using the learning rate and the gradient. This step is repeated for each mini-batch and the process continues until all mini-batches have been processed.
- **Epochs:** An epoch refers to one complete pass through the entire dataset. After each epoch the mini-batches are typically reshuffled to ensure that the model does not overfit to the specific order of the data.

As a general guideline, start with a batch size of *64* or *128* and adjust based on the training behavior. If training is too slow or unstable then experiment with smaller or larger batch sizes.

Five-minute activity: Optimization

Three students train a Logistic Regression model on the Iris dataset using different learning rates: [0.0001, 0.01, 10].

Student	Learning Rate	Final Accuracy
A	0.0001	83%
B	0.01	97%
C	10	63%

1. Run the code and observe:
   (a) Which learning rate converged fastest?
   (b) Which gave the highest test accuracy?
   (c) Sketch a loss-vs-iteration graph for a learning rate that is too small vs. one that is too large.
2. Why did Student B obtain the best result?
   (a) What happens when the learning rate is too small?
   (b) What happens when it is too large?
   (c) Why might neither extreme be desirable?

5 Generalization

Learning Objectives

- Define overfitting, underfitting, and the bias-variance tradeoff.
- Explain how model complexity affects test error.
- Apply L1/L2 regularization, dropout, and early stopping.

5.1 Introduction

Consider two students preparing for a final exam using past papers. **Memorizing Ellie** memorizes every answer from previous exams and scores perfectly when the same questions appear. However, she struggles when faced with new questions. In contrast, **Inductive Irene** focuses on understanding concepts and identifying patterns. Although she may not score perfectly on familiar questions, she performs consistently well on new ones. Similarly, a machine learning model that memorizes the training data may achieve excellent training performance but fail on unseen data, whereas a model that learns underlying patterns is more likely to **generalize** effectively.

This challenge of learning patterns that generalize effectively to unseen data is one of the central problems in machine learning and statistical learning theory. It lies at the heart of concepts such as **overfitting**, **underfitting**, **model complexity**, and generalization error. More broadly, this issue reflects a fundamental question encountered throughout science and statistics: under what conditions can conclusions drawn from specific observations be reliably extended to broader situations or future events?

Whenever machine learning models are trained on finite samples, there is always a risk that the model may simply memorize the training data rather than learn underlying patterns that generalize to unseen examples. A phenomenon, where a model fits the training data too closely while failing to generalize effectively to new data, is known as *overfitting*. To address this problem, machine learning practitioners employ a variety of techniques collectively referred to as *regularization* methods. These methods are designed to improve a model's ability to generalize by reducing excessive sensitivity to the training data.

Whenever a machine learning model is too simple to capture the underlying relationships present in the data, it may fail to learn important patterns during training. A phenomenon in which a model performs poorly on both the training data and unseen data because it is unable to adequately represent the complexity of the problem is known as *underfitting*. Underfitting typically occurs when the model has insufficient capacity, the training process is inadequate, or overly restrictive assumptions are imposed on the learning algorithm. To address this issue, practitioners may increase model complexity, incorporate more informative features, extend the training process, or reduce excessive regularization, thereby enabling the model to better capture the underlying structure of the data.

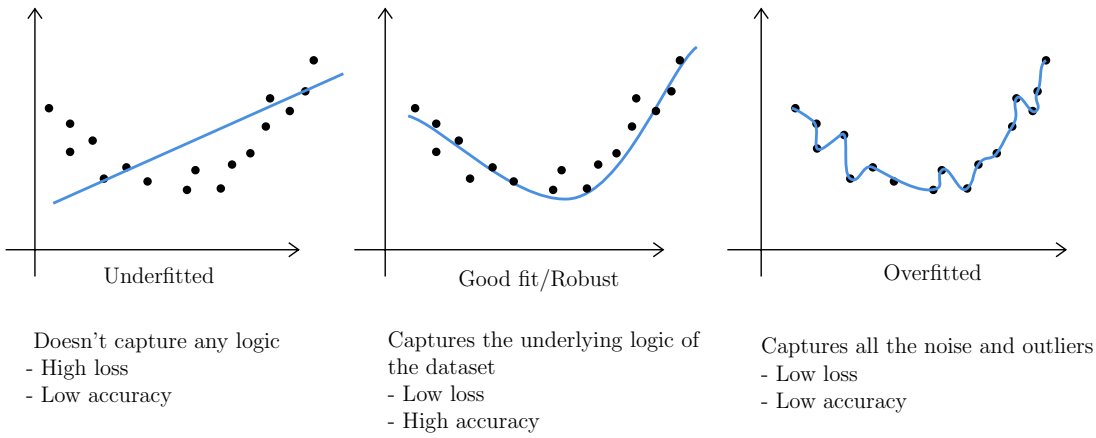


Figure 6: Underfitting and overfitting illustrated.

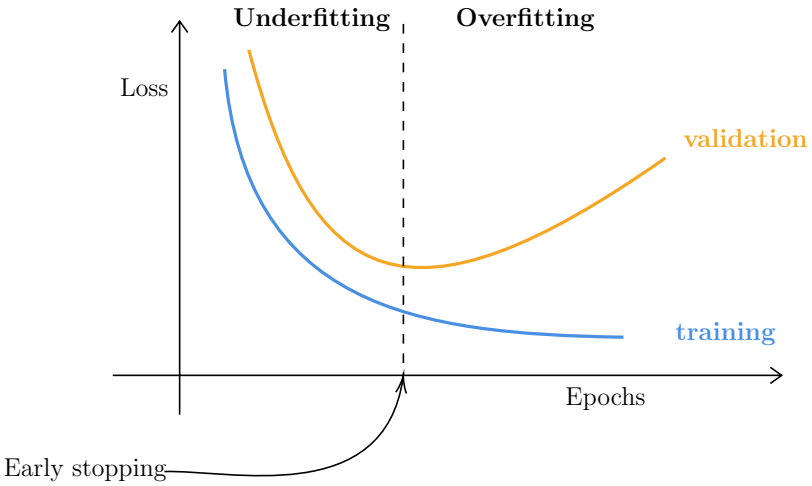


Figure 7: Underfitting and overfitting illustrated.

Aspect	Underfitting	Overfitting
Definition	The model is too simple to capture the underlying patterns in the data.	The model learns the training data too closely, including noise and random fluctuations.
Training Performance	Poor.	Excellent
Testing Performance	Poor	Poor
Training Error	High	Very low
Validation/Test Error	High	High
Model Complexity	Too simple	Too complex
Bias	High	Low
Variance	Low	High
Generalization Ability	Poor because the model cannot learn the true patterns.	Poor because the model memorizes training data and fails on unseen data.
Example	Using a linear model to fit a	Using a very deep neural network

5.2 Regularization Techniques

Regularization refers to a set of techniques used to improve a model's ability to generalize by reducing overfitting and preventing it from becoming overly sensitive to the training data. The goal is to reduce overfitting usually achieved by reducing model capacity and/or reduction of the variance of the predictions.

5.2.1 L1/L2 regularization

L1 and L2 regularization are techniques used to reduce overfitting by adding a penalty term to the loss function, discouraging overly complex models.

- **L1 Regularization (Lasso)** adds the absolute values of model weights to the loss function. It can drive some weights to exactly zero, effectively performing feature selection.
- **L2 Regularization (Ridge)** adds the squared values of model weights to the loss function. It reduces the magnitude of weights without eliminating them, leading to smoother and more stable models.

5.2.2 Dropout

Randomly drops (sets to zero) a fraction of neurons during each training pass in neural networks.

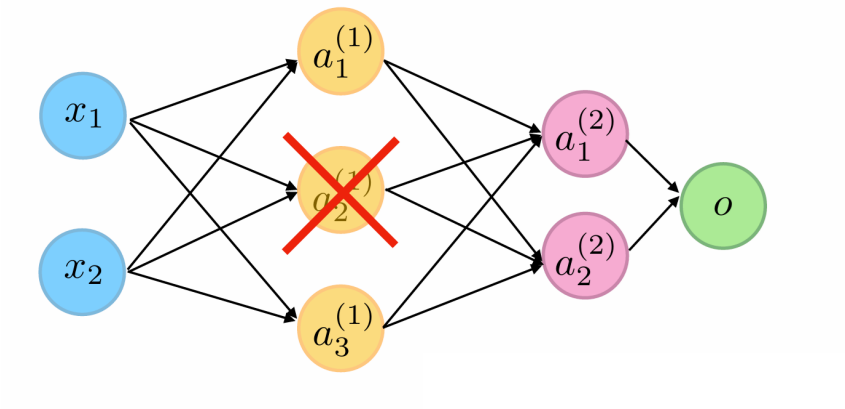


Figure 8: Dropping nodes

Dropout reduces reliance on any one neuron, making networks more robust, simple, and effective. However, this can slow down training and this technique requires careful tuning — dropping many nodes leads to underfitting.

5.2.3 Early stopping

Illustrated in Figure 7, early stopping monitors model performance on a validation set and stops training when performance stops improving.

Five-minute activity: Generalization

Context
You are building a machine learning model to classify patient reports into one of several rare disease categories. The dataset contains:

- **10,000** labelled patient reports (free text)
- **50,000** TF-IDF features (high-dimensional, sparse)
- **Class prevalence:** Rare diseases – only about 1% of samples are positive for any given disease (highly imbalanced)
- **Model:** Logistic regression (trained with SGD)
- **Current performance:**
  - Training accuracy: 99%
  - Test accuracy: 65%
  - You suspect that many of the 50,000 features are noise (irrelevant words or phrases)

Problem
You need to improve generalization /test accuracy without significantly increasing computational cost.

Your task
Answer the following questions with your neighbor.

1. Is the model currently overfitting, underfitting, or both? Justify using the given numbers.
2. Which of the following regularization method(s) would you apply first, and why? *(You may choose more than one, but prioritise the most suitable.)*
   (a) L1 regularization (Lasso)
   (b) L2 regularization (Ridge / weight decay)
   (c) Dropout
   (d) Early stopping
3. Which method(s) would you not use for this problem?

Glossary of Key Terms

Activation Function A mathematical function applied to a neuron's output to introduce nonlinearity into a neural network.

Accuracy The proportion of correct predictions made by a classification model.

Algorithm A step-by-step computational procedure used to solve a problem or perform a task.

Artificial Neural Network (ANN) A computational model inspired by the human brain, composed of interconnected neurons.

Backpropagation A learning algorithm used to update neural network weights by propagating errors backward through the network.

Batch Gradient Descent An optimisation algorithm that computes gradients using the entire training dataset before updating parameters.

Bias Systematic error from incorrect assumptions in the learning algorithm.

Classification A supervised learning task where the output variable belongs to a predefined category or class.

Clustering An unsupervised learning technique used to group similar data points together.

Confusion Matrix A table used to evaluate classification models by comparing predicted and actual values.

Cost Function A mathematical function that measures the error between predicted and actual values.

Cross-validation A resampling method used to estimate model generalisation performance.

Data Exploration The process of analysing and visualising datasets to identify patterns, trends, and anomalies.

Data Preprocessing The preparation and cleaning of raw data before model training.

Dataset A structured collection of data used for machine learning analysis and training.

Deep Learning A subfield of machine learning based on neural networks with multiple hidden layers.

Epoch One complete pass of the entire training dataset through the learning algorithm.

Feature An individual measurable property or characteristic used as input to a model.

Feature Engineering The process of creating, selecting, or transforming features to improve model performance.

F1-Score The harmonic mean of precision and recall.

Generalisation The ability of a machine learning model to perform well on unseen data.

Gradient Descent An iterative optimisation algorithm that minimises a loss function using gradients.

Hyperparameter A configuration setting chosen before training that controls the learning process.

Inference The process of using a trained model to make predictions on new data.

K-Fold Cross-Validation A validation method where data is split into k subsets and each subset is used once for validation.

Learning Rate The step size used during parameter updates in gradient descent.

Local Minimum A point where the loss function is lower than nearby points but not necessarily the lowest possible value.

Loss Function A mathematical representation of prediction error used during optimisation.

Machine Learning Pipeline A sequence of processes including data preparation, training, evaluation, and deployment.

Mean Squared Error (MSE) A regression loss function that measures the average squared prediction error.

Mini-Batch Gradient Descent A variant of gradient descent that updates parameters using small subsets of data.

Model A mathematical representation learned from data to make predictions or decisions.

Normalization Scaling data to a fixed range, often between 0 and 1.

Optimisation The process of adjusting model parameters to minimise prediction error.

Overfitting Fitting training data too closely, resulting in poor performance on unseen data.

Parameter Internal variables of a model learned during training.

Precision The proportion of predicted positive cases that are actually positive.

Prediction The output produced by a trained machine learning model.

Recall The proportion of actual positive cases correctly identified by a model.

Regression A supervised learning task where the output variable is continuous or numeric.

Regularisation Techniques used to reduce overfitting and improve generalisation.

ReLU A commonly used activation function defined as $f(x) = \max(0, x)$.

Stochastic Gradient Descent (SGD) An optimisation method that updates parameters using one training example at a time.

Supervised Learning A machine learning approach using labelled training data.

Test Set A portion of data used to evaluate the final performance of a trained model.

Training Set The subset of data used to train a machine learning model.

Underfitting A condition where a model is too simple to capture underlying patterns in the data.

Unsupervised Learning A machine learning approach that identifies patterns in unlabeled data.

Validation Set A subset of data used to tune model hyperparameters and assess generalisation during training.

Variance Sensitivity of a model to small changes in the training data.

Weight A trainable parameter in a neural network that determines the strength of connections between neurons.

Appendix

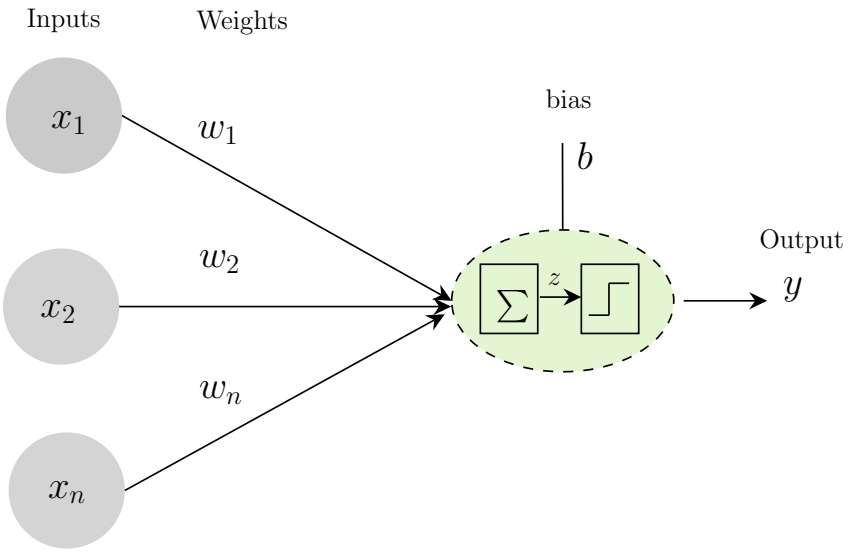


Figure 9: A single neuron comprised of inputs and their respective weights, the bias, the arithmetic operation, the activation function and the output.

A simple neuron computes:

$$y = z(w_1x_1 + w_2x_2 + \cdots + w_nx_n + b)$$

or in matrix form:

$$y = z(Wx + b)$$

where:

- x represents inputs
- w represents weights
- b represents bias
- z is the weighted sum
- y is the output of the neuron

Important terms

- Inputs
- Weights
- Biases
- Activation functions

- Feedforward computation
- Error calculation
- Backpropagation
- Gradient descent