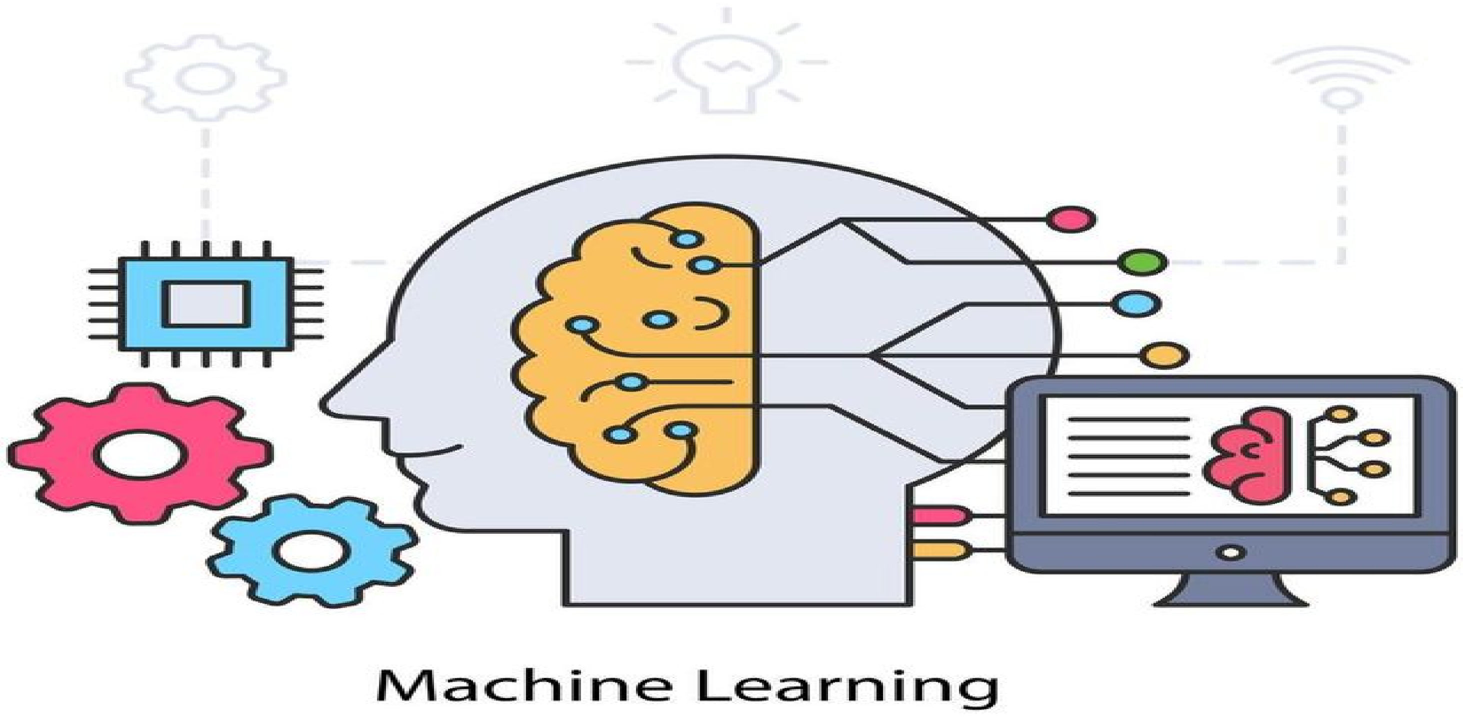


Traditional ML
Tutorial

2025

Traditional Machine Learning

Forough Shirin Abkenar

Contents

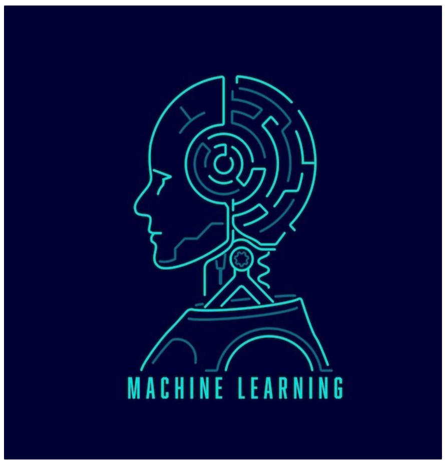
List of Figures

List of Tables

Overview

The current document studies fundamental topics in traditional machine learning (ML), such as the ML lifecycle, supervised learning, unsupervised learning, and evaluation metrics. The corresponding codes are implemented in Python and PyTorch.

It should be noted that the cliparts used in this document were downloaded from Pinterest [1].



Data Preparation

Machine learning (ML) models rely on data as input. In ML, data are divided into two major categories, namely *Structured* and *Unstructured*. The former includes *Numerical* and *Categorical* data, and the latter comprises *Images*, *Audios*, *Videos*, and *Texts*. Categorical data, a type of structured data, represent qualitative information that is divided into distinct groups or labels. They can be further classified into two subtypes: *Nominal* data (e.g., gender) and *Ordinal* data (e.g., rating scales) [2].

2.1 Feature Engineering

Each record in a dataset is called a data point (or sample). Each data point is composed of fundamental components, called *features*. The number of features and the relation between them play an important role in the performance of an ML model. Therefore, *feature engineering* is a crucial step in preparing the data before they are fed to the model.

The main focus of the current document is on structured data and image data. Therefore, we review feature engineering techniques relevant to these data types. Some techniques apply to both structured and image data; however, others are specific to images. Therefore, we explicitly highlight the image-specific methods in the corresponding subsections.

2.1.1 Handling Missing Values

Missing values are common challenges in ML datasets, where the feature values corresponding to some data points are missing. There are different methods to cope with the missing values issue [3].

Removing Rows with Missing Values

In this method, we remove all data points with missing values. Although this method is simple to implement and removes potentially problematic data points, it reduces the size of sample data and is vulnerable to introduce bias in the dataset provided certain groups are more likely to have missing values [3].

Imputation

Using the imputation method, the missing values are replaced by estimated values. There are two methods to impute the missing values [3]:

- **Mean, Median and Mode Imputation:** The missing values are replaced with the *mean*, *median*, or *mode* of the corresponding feature. This method is simple to implement. However, it might reduce the accuracy of predictions [3].
- **Forward and Backward Fill:** The missing values are filled with the nearest non-missing values from the same feature, where the forward fill method relies on replacing the missing value with the last observed non-missing value, and the backward fill approach replaces them with the next observed non-missing value [3].

Interpolation

Rather than relying on measures such as the mean, median, or mode (as in simple imputation), interpolation estimates missing values by leveraging the relationships between neighboring data points. This method is more complex to implement and depends on assumptions, such as the existence of linear or quadratic relationships within the data. However, it often yields more accurate results than imputation and better preserves data integrity by capturing underlying patterns or trends. Two common interpolation techniques are [3]:

- **Linear:** linear method uses a linear interpolation to estimate the missing values [3].
- **Quadratic:** Quadratic interpolation method assumes a quadratic relationship between a missing value and its surrounding known values, and estimates the missing value accordingly [3].

2.1.2 Resizing

To standardize the shape of input images, they must be resized to a fixed size, such as 28×28.

2.1.3 Scaling

Data can have different value scales, where larger values may unintentionally dominate certain features. To prevent such issues during model training and evaluation, data are scaled to a specific range. Two common scaling methods are *normalization*, *standardization*, and *log scaling* [4].

Normalization

Normalization, also known as min-max scaling, wherein data are scaled into the range $[0, 1]$ [4]:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{2.1}$$

where x_{min} and x_{max} are the minimum and the maximum values of the feature, respectively. It is worth mentioning that normalization does not alter the data distribution [4].

Standardization

Unlike normalization, which preserves the original distribution of the data, standardization, also known as Z-score normalization, transforms a feature so that it has a mean of 0 and a standard deviation of 1 [4]:

$$x_{scaled} = \frac{x - \mu}{\sigma}, \tag{2.2}$$

where μ and σ are the mean and standard deviation of the feature, respectively [4].

Log Scaling

Features can exhibit a power law distribution, where low values of x correspond to high values of y, and y decreases rapidly as x increases. An example of this is movie ratings, where a few movies receive many ratings while most receive very few. Logarithmic scaling can help mitigate the effects of a power law distribution by transforming the data into a more balanced scale [4].

$$x_{scaled} = \log(x) \tag{2.3}$$

2.1.4 Binning

When the overall linear relationship between a feature and the label is weak or nonexistent, or when feature values are clustered, traditional scaling methods may fail. Binning, also known as bucketing, provides an effective alternative by converting numerical data into categorical data. This method groups numerical subranges into bins or buckets, which can better represent features that exhibit clustered or “clumpy” distributions rather than linear patterns [4].

2.1.5 Encoding

As mentioned earlier, categorical data represent qualitative information. Since ML models operate on numerical values, categorical data must be transformed into a numeric format through *Encoding*. Encoding converts categorical values into numerical representations and can be performed using methods such as *One-Hot Encoding* and *Embedding Learning* [4].

One-Hot Encoding

One-hot encoding assigns a binary vector to each category [4]. For example, if the feature is weekdays, each day can be encoded as:

Table 2.1: One-Hot Encoding Example

Weekdays	Value						
Monday	1	0	0	0	0	0	0
Tuesday	0	1	0	0	0	0	0
Wednesday	0	0	1	0	0	0	0
Thursday	0	0	0	1	0	0	0
Friday	0	0	0	0	1	0	0
Saturday	0	0	0	0	0	1	0
Sunday	0	0	0	0	0	0	1

Embedding Learning

Pre-trained models can provide embeddings, i.e., numerical vector representations, of input data. These models are particularly useful when capturing the semantic relationships between inputs is important.

2.1.6 Consisting Color Mode

In image processing, it is important for all images to have a consistent color mode. Common color modes include Grayscale, RGB (Red, Green, Blue), and CMYK (Cyan, Magenta, Yellow, Key/Black).

2.2 Data Augmentation

Data augmentation is a technique to increase the size and diversity of data samples for training purposes. To this end, the corresponding augmentation method generates new data from the existing data [5].

- **Tabular Data:** For tabular data, new samples can be generated through techniques such as adding random noise to existing values, performing feature permutation (swapping values within the same column), or creating synthetic data based on the mean and standard deviation of the original data [5].

- **Images:** For image data, new samples can be generated through augmentation techniques such as cropping, adjusting saturation, flipping (horizontal or vertical), and rotating the original images [5].

2.3 Data Balancing

If a labeled dataset is imbalanced, i.e., the number of data points varies significantly across categories, the model tends to learn patterns from the majority classes while underrepresenting the minority classes. Data balancing techniques, namely *Upsampling* and *Downsampling*, are used to address this issue and ensure fair learning across all categories [6].

2.3.1 Upsampling

Upsampling methods increase the number of samples in the minority class. Although upsampling increases the dataset size, it is vulnerable to data leak, which leads to model overfitting. Common upsampling techniques are listed as, random oversampling, synthetic minority oversampling technique (SMOTE), adaptive synthetic sampling approach (ADASYN), and data augmentation [7].

Random Oversampling

Random oversampling chooses data points randomly from the minority class and duplicates them. Random oversampling is vulnerable to model overfitting [7].

Synthetic Minority Oversampling Technique

The SMOTE generates new samples for the minority class by interpolation. First, for each minority class data point, the algorithm identifies its K nearest neighbors (with K commonly set to 5). Then, one of these neighbors is randomly selected, and a new synthetic sample is created at a random point along the line segment connecting the original data point and the chosen neighbor in the feature space. This process is repeated with different neighbors as needed until the desired level of upsampling is achieved [7].

Adaptive Synthetic Sampling Approach

The ADASYN technique extends the idea of SMOTE by focusing on regions where the minority class is underrepresented. A K-nearest neighbor (KNN) model is first built on the entire dataset, and each minority class point is assigned a "hardness factor" (r), defined as the ratio of majority class neighbors to the total number of neighbors in KNN. Similar to SMOTE, new synthetic samples are generated through linear interpolation between a minority data point and its neighbors. However, the number of samples generated is scaled by the hardness factor so that more synthetic points are created in regions where minority data are sparse, and fewer points are added in regions where they are already dense [7].

Data Augmentation

Data augmentation (see Section 2.2) can also be applied as a strategy for balancing datasets [7].

2.3.2 Downsampling

Downsampling methods reduce the number of samples in the majority class to match the size of the minority class. While this approach can lower the risk of model overfitting, it also increases the likelihood of underfitting and may introduce bias by discarding potentially useful data. Common downsampling techniques are random downsampling and near miss downsampling [8].

Random Downsampling

Similar to random oversampling in upsampling, random downsampling selects data points at random; however, in this case, the selected points come from the majority class and are removed [8].

Near Miss Downsampling

Near Miss Downsampling involves distance-based instance selection. In this method, the pairwise distance between all majority and minority class instances is first calculated. Based on these distances, majority class instances that are farther away from minority points are removed. This ensures that the remaining majority samples are closer to the minority class distribution, helping the model better capture decision boundaries [8].

Evaluation Metrics

After training (or tuning/fitting) a classification model, it is important to evaluate its performance and assess how well it can make predictions. To do this, the model is provided with test data, i.e., unseen data that were not used during training, and its performance is measured using various evaluation metrics. Depending on the type of the ML model, various evaluation metrics are employed.

3.1 Classification

Several metrics can be used to evaluate the performance of a tuned model for the classification purposes (binary classification). In this chapter, we focus on three key metrics: *Accuracy*, *Recall* (or true positive rate), *false positive rate*, *Precision*, *F1-score*, and *ROC-AUC*.

3.1.1 Confusion Matrix

Before discussing evaluation metrics, it is important to understand the confusion matrix and its components, which play a crucial role in model evaluation [4].

A confusion matrix consists of four elements: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). Figure 3.1 illustrates the confusion matrix, where columns represent the actual values and rows represent the predicted values. TP and TN occur when the model's predictions match the actual values. Conversely, if the model predicts a negative (positive) instance as positive (negative), it corresponds to FP (FN) [4].

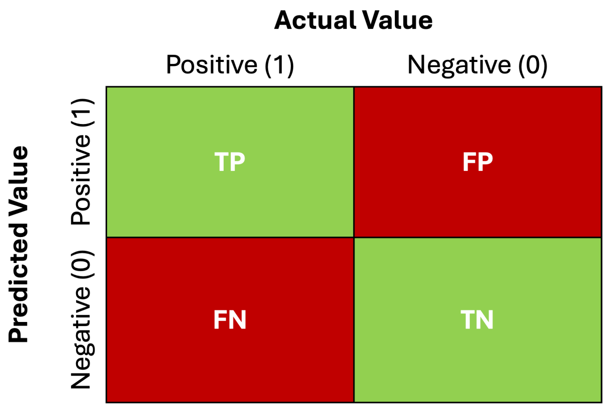


Figure 3.1: Confusion Matrix [4].

3.1.2 Accuracy

Accuracy indicates how accurate a model is to predict both positive and negative classes correctly [4].

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \tag{3.1}$$

3.1.3 Recall

Recall, also known as true positive rate (TPR), measures how effectively a model identifies positive instances. Specifically, it is the proportion of true positive instances relative to all actual positive samples, including both correctly predicted positives and positive instances that were incorrectly predicted as negative [4].

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3.2}$$

3.1.4 False Positive Rate

The false positive rate (FPR) measures how often a model incorrectly classifies negative instances as positive. Specifically, it is the proportion of negative samples that are misclassified as positive relative to all actual negative samples [4].

$$\text{FPR} = \frac{\text{FP}}{\text{TN} + \text{FP}} \tag{3.3}$$

3.1.5 Precision

Precision describes how precise a model is to classify true positive instances. Precision measures the proportion of true positive instances among all predicted positive instances [4].

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{3.4}$$

3.1.6 F1-score

In imbalanced datasets, the arithmetic mean can be skewed by high values. In contrast, the harmonic mean gives more weight to lower values. The F1-score (that is F$_\beta$-score with $\beta = 1$) is the harmonic mean of precision and recall, that guarantees a low value in either metric results in a correspondingly low F1-score [4].

$$\text{F1-score} = \frac{(1+\beta^2) \times \text{Precision} \times \text{Recall}}{\beta^2 \times \text{Precision} + \text{Recall}} \overset{\beta=1}{=} 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3.5}$$

3.1.7 ROC-AUC

Accuracy (Sec. 3.1.2) depends on a specific decision threshold. In contrast, the Receiver Operating Characteristic (ROC) curve provides a visual representation of model performance across all thresholds, making it threshold-independent. The ROC plots the True Positive Rate (TPR) against the False Positive Rate (FPR). The Area Under the Curve (AUC) quantifies the probability that the model will assign a higher score to a randomly chosen positive instance than to a randomly chosen negative one [4].

Figure 3.2 illustrates the ROC curve for a binary classification scenario in which the model makes predictions completely at random, like a coin flip. In this case, the AUC equals 0.5 [4].

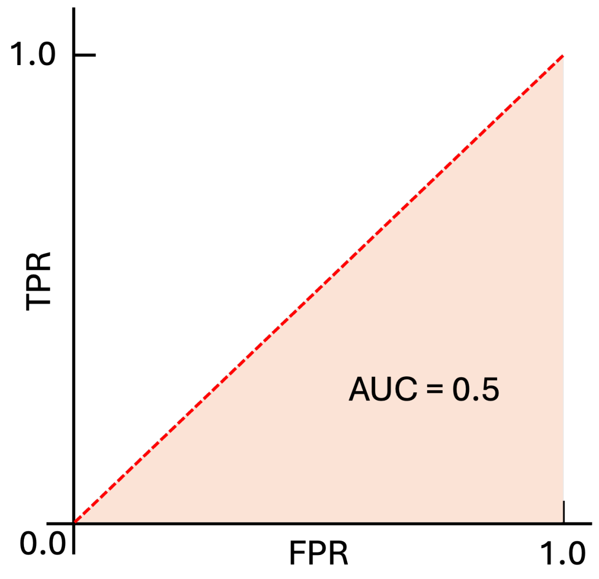


Figure 3.2: ROC-AUC of completely random guesses [4].

The ROC-AUC concept can be extended to Precision-Recall (PR) curves to more effectively evaluate model performance. Figure 3.3 shows the PR curve and its corresponding AUC for an example scenario, providing insight into how the model balances precision and recall. PR-AUC is particularly useful for imbalanced datasets, where the positive class is rare, because it focuses on the model's performance with respect to the minority class rather than being dominated by the majority class, as can happen with ROC-AUC. In general, higher PR-AUC values indicate better model performance [4].

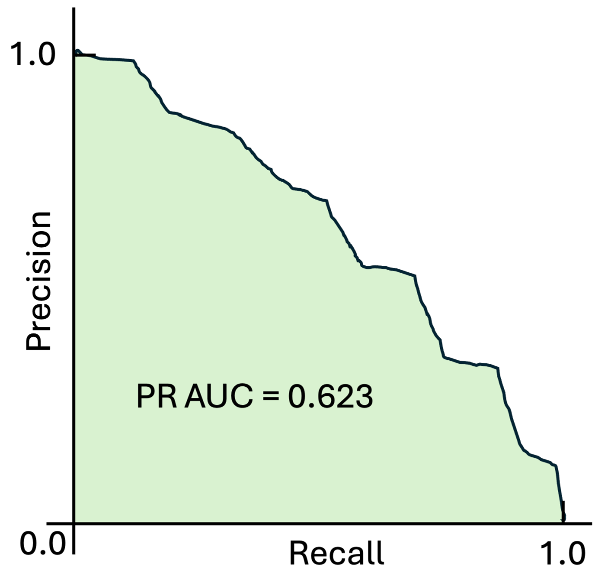


Figure 3.3: PR-AUC [4].

3.2 Regression

Unlike classification models, which predict discrete classes or labels, regression models produce continuous numerical values as output. Therefore, specific evaluation metrics are needed to assess the performance of regression models.

In regression models, the key concept for evaluating performance is *error* (or *residual*). The error is defined as the difference between the actual value and the predicted value. Regression models are developed to minimize the error.

3.2.1 Mean Absolute Error

Mean absolute error (MAE) measures the arithmetic mean of absolute errors for predictions [9].

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|\text{Actual}_i - \text{Predicted}_i| \tag{3.6}$$

MAE treats all errors equally, assigning the same weight to small and large errors. As a result, large errors are not penalized more heavily than small ones. This property also makes MAE relatively robust to outliers, i.e., extreme values that lie far from the typical range of the data. Overall, MAE is appropriate for balanced data [9].

3.2.2 Mean Squared Error

Mean squared error (MSE) calculates the arithmetic mean of squared differences between the actual data and the predicted values [9].

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\text{Actual}_i - \text{Predicted}_i)^2 \tag{3.7}$$

Squaring the errors makes the MSE enable to penalize larger errors more heavily than smaller errors, making it sensitive to outliers in the data [9].

3.2.3 Root Mean Squared Error

Root mean squared error (RMSE) measures to root of MSE, i.e., the arithmetic mean of squared differences between the actual data and the predicted values [9].

$$\text{RMSE} = \sqrt{\text{MSE}} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{Actual}_i - \text{Predicted}_i)^2} \tag{3.8}$$

3.2.4 R-squared Score

R-squared Score (R^2-score) is a measure of goodness of fit, i.e., it measures how close the data points are to the fitted line [9].

$$\text{R}^2\text{-score} = 1 - \frac{\text{SS}_{\text{RES}}}{\text{SS}_{\text{TOT}}} = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2}, \tag{3.9}$$

where SS_{RES} and SS_{TOT} show the sum of squares residuals (errors) and total, respectively. The former, measures the sum of squared difference between predicted values and actual values. The latter calculated the sum of squared difference between the actual values and their arithmetic mean [9].

3.3 Clustering

Unlike classification and regression problems, where data are labeled, clustering deals with unlabeled data. Since no ground-truth labels are available, clustering models group data based on similarities such as distance or distribution. As a result, traditional evaluation metrics used in classification or regression are not directly applicable to clustering.

3.3.1 Silhouette Score

The Silhouette Score evaluates the quality of clustering by measuring how well each data point fits within its assigned cluster compared to neighboring clusters. A score close to +1 indicates that a data point is well-matched to its own cluster and far from other clusters, while a score near 0 suggests that the point lies on or near a cluster boundary. Negative scores (close to −1) indicate that a data point may have been misassigned to the wrong cluster. The overall clustering quality is typically assessed by the average silhouette score across all data points, with **higher values** reflecting better-defined and more clearly separated clusters [10].

$$s(i) = \frac{b(i) - a(i)}{\max\{a(i), b(i)\}}, \tag{3.10}$$

where $a(i)$ is the inter-cluster distance, i.e., average distance from point i to all other points in the same cluster; and $b(i)$ represents the nearest-cluster distance, i.e., minimum average distance from point i to points in the nearest neighboring cluster [10].

3.3.2 Davies-Bouldin Index

The Davies-Bouldin Index (DBI) is an internal evaluation metric that measures the quality of clustering based on the trade-off between cluster compactness and separation. Compactness is quantified by the average distance of data points within a cluster to their centroid, while separation is measured by the distance between cluster centroids. A **lower DBI** value indicates better clustering, as it reflects more compact clusters that are well separated from each other. Since DBI relies only on the data and cluster assignments, it does not require external ground-truth labels [10].

$$DBI = \frac{1}{k}\sum_{i=1}^{k} \max_{i \neq j}\left(\frac{S_i + S_j}{M_{ij}}\right), \tag{3.11}$$

where k is the number of clusters; S_i represents the compactness of the cluster i, that is, the average distance of all points of the cluster i to its centroid; and M_{ij} indicates the separation between the clusters, that is, the distance between the centroids of the clusters i and j. Notably, $(S_i + S_j)/M_{ij}$ compares the similarity between clusters i and j [10].

3.4 Ranking and Recommendation

In ranking and recommendation tasks, the developed machine learning system generates a list of outputs ranked from highest to lowest relevance. However, according to the ground-truth data, some truly relevant items may be incorrectly classified as irrelevant. Evaluation metrics such as **Recall@k**, **Precision@k**, and **Mean Average Precision (mAP)** are commonly used to assess the model's performance in capturing and ranking the relevant results accurately.

3.4.1 Precision@K

In ranking and recommendation tasks, the evaluation is typically limited to the top-K retrieved items. In this context, Precision@K measures how many of the items within the top-K positions are relevant. Formally, Precision@K is defined as the ratio of relevant items retrieved in the top-K results to K. Therefore, we have [11]

$$\text{Precision@K} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{Number of relevant items in K}}{\text{K}} \tag{3.12}$$

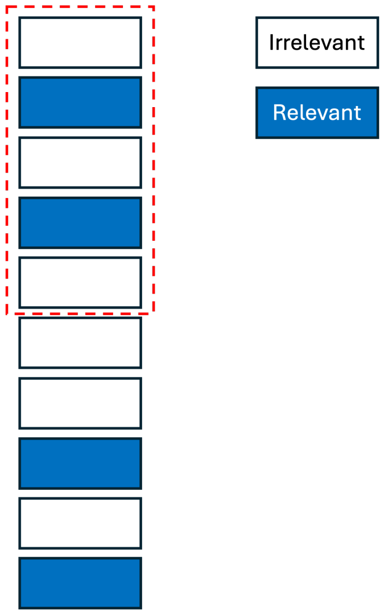


Figure 3.4: An example for a single-list ranking system.

Figure 3.4 illustrates an example of a ranking system with $K = 5$. As shown, the model successfully identifies two relevant items out of four total relevant items within the top-K results. Therefore, the Precision@K value is computed as $2/5 = 0.4$.

It is worth noting that Precision@K is highly sensitive to the choice of K. Consider a case where only two relevant items exist in the ground-truth data, and the model successfully retrieves both within the top-K results. If K is set to a large value, the precision will decrease substantially, since the number of retrieved items increases while the number of relevant items remains constant.

3.4.2 Recall@K

Recall is defined as the model's ability to identify true positive instances in its predictions (see Eq. 3.2 in Sec. 3.1.3). In ranking and recommendation tasks, Recall@K measures how many of the relevant items are correctly retrieved by the model within the top-K positions. Formally, Recall@K is defined as the ratio of relevant items retrieved in the top-K results to the total number of relevant items in the ground-truth data. Therefore, we have [11]

$$\text{Recall@K} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{Number of relevant items in K}}{\text{Total number of relevant items}} \tag{3.13}$$

Referring to the example provided in Fig. 3.4, the Recall@K value is computed as $2/4 = 0.5$. Therefore, Recall@K reduces the sensitivity to K observed in Precision@K. However, although Recall@K is less sensitive to the value of K compared to Precision@K, it is insensitive to the relative ranking of relevant items within the top-K positions. For example, consider a case where only two relevant items exist in the ground-truth data, and both are retrieved within the top-K results but appear at lower ranks (e.g., positions 9 and 10 when $K = 10$). In this case, Recall@K equals 1.0, even though the ranking quality is poor.

3.4.3 Mean Reciprocal Rank@K (MRR@K)

To understand Mean Reciprocal Rank@K (MRR@K), it is necessary to introduce Reciprocal Rank@K (RR@K) first. RR@K is a metric that focuses on the position of the first relevant item within the top-

K results. It is defined as [11]

$$\text{RR} = \frac{1}{\text{Rank of the first relevant item}} \tag{3.14}$$

For example, considering the example in Fig. 3.4, the RR@K is given as $1/2 = 0.5$. Considering U ranked lists in the results, each corresponding to a query in an information retrieval task or a user in a recommendation system, MRR@K is defined as the average of RR@K across all U lists, i.e.,

$$\text{MRR@K} = \frac{1}{U}\sum_{u=1}^{U} RR@K_u \tag{3.15}$$

For instance, Fig. 3.5 illustrates a ranking system with three lists in the outputs. As shown, the positions of the first relevant items are 2, 3, and 2 in the first, the second, and the third lists, respectively. Therefore, the corresponding RR@K values are computed as $1/2 = 0.5$, $1/3 = 0.33$, and $1/2 = 0.5$. Accordingly, the MRR@K is equal to $(0.5 + 0.33 + 0.5)/3 = 0.44$.

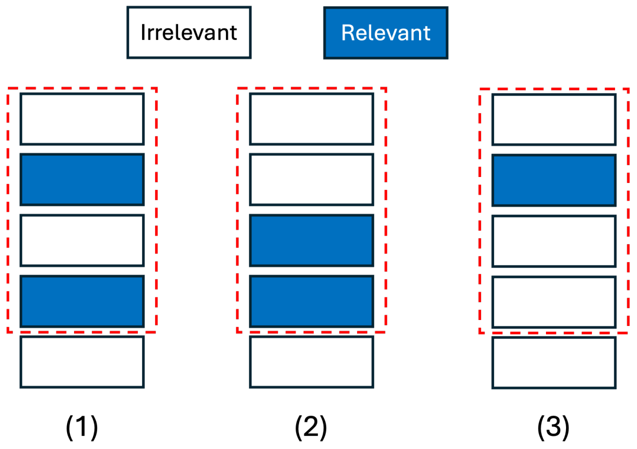


Figure 3.5: An example for a multi-list ranking system.

MRR@K addresses the limitation of Precision@K and Recall@K, both of which ignore the ranks of relevant items within the top-K results. However, MRR@K only accounts for the rank of the first relevant item and disregards the positions of other relevant items that may also appear within the top-K list.

3.4.4 Mean Average Precision@K (mAP@K)

Unlike MMR@K that measures the rank of the first relevant item in the output lists, Mean Average Precision@K (mAP@K) considers all relevant elements in the top-K positions. mAP@K is the mean of AP@K values. Taking into account N_u relevant items in the list u, we have [11]

$$\text{AP@K}_u = \frac{1}{N_u}\sum_{k=1}^{K} \text{Precision}(k) \times \text{rel}(k), \tag{3.16}$$

where Precision(k) is the precision of predictions at top-k positions, where $k = 1, \dots, K$; also, rel(k) is the relevance score of the item, that can be a binary value (relevant/not relevant) or a graded score (e.g., 0, 1, 2, 3). In the case of AP@K, we defined is as a binary variable where is equal to 1 if the item is relevant, and 0 otherwise. For example, referring to Fig. 3.5, for $u = 1$, we have:

- $k = 1$: $\text{Precision}(k) = 0/1 = 0$, $\text{rel}(k) = 0$
- $k = 2$: $\text{Precision}(k) = 1/2 = 0.5$, $\text{rel}(k) = 1$
- $k = 3$: $\text{Precision}(k) = 1/3 = 0.33$, $\text{rel}(k) = 0$

- $k = 4$: Precision$(k) = 2/4 = 0.5$, rel$(k) = 1$

Therefore,

$$\text{AP@K}_1 = \frac{0 \times 0 + 0.5 \times 1 + 0.33 \times 0 + 0.5 \times 1}{2} = \frac{0 + 0.5 + 0 + 0.5}{2} = \frac{1.0}{2} = 0.5$$

Accordingly, for $u = 2$ and $u = 3$, we have

$$\text{AP@K}_2 = \frac{0 \times 0 + 0 \times 0 + 0.33 \times 1 + 0.0.5 \times 1}{2} = \frac{0 + 0 + 0.33 + 0.5}{2} = \frac{0.83}{2} = 0.42$$

$$\text{AP@K}_3 = \frac{0 \times 0 + 0.5 \times 1 + 0 \times 0 + 0 \times 0}{1} = \frac{0 + 0.5 + 0 + 0}{1} = \frac{0.5}{1} = 0.5$$

mAP@K measure the mean of all AP@K values as

$$\text{mAP@K} = \frac{1}{U} \sum_{u=1}^{U} \text{AP@K}_u \tag{3.17}$$

Therefore, mAP@K value for the above-mentioned example in Fig. 3.5 is computed as

$$\text{mAP@K} = \frac{0.5 + 0.42 + 0.5}{3} = \frac{1.42}{3} = 0.47$$

3.4.5 Normalized Discounted Cumulative Gain@K (nDCG@K)

It is important not only to determine whether an item is relevant within an output list but also to evaluate how relevant it is. None of the aforementioned metrics account for the degree of relevance of items. In contrast, Normalized Discounted Cumulative Gain@K (nDCG@K) captures this aspect by assigning graded relevance scores to items. To compute nDCG@K, the Discounted Cumulative Gain@K (DCG@K) is first calculated, which applies a logarithmic discount factor based on the position of each item in the ranked list, i.e., $\log_2(i + 1)$. Formally, DCG@K is defined as [11]

$$\text{DCG@K} = \sum_{i=1}^{K} \frac{\text{rel}_i}{\log_2(i + 1)}, \tag{3.18}$$

In addition to the predicted ranks, the Ideal DCG (IDCG) metric represents the maximum possible DCG for a given set of results. nDCG@K normalizes the DCG with respect to the IDCG, yielding a score between 0 and 1. Formally [11],

$$\text{nDCG@K} = \frac{\text{DCG@K}}{\text{IDCG@K}} \tag{3.19}$$

For instance, consider the results in Fig. 3.19. There are three output lists, labeled 1, 2, and 3, and an ideal list with the ideal ranking of the items.

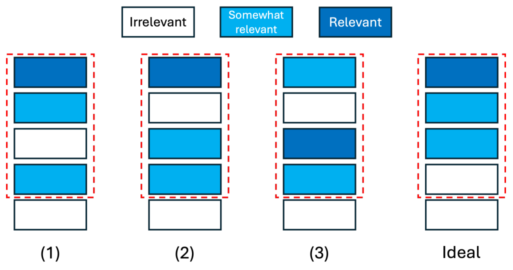


Figure 3.6: An example for nDCG.

Since the items are categorized as irrelevant, somewhat relevant, and relevant, we define relevance scores 0, 1, and 2, respectively. Therefore, for each list u, the corresponding DCG@K$_u$ is computed as

$$\text{DCG@K}_1 = \frac{2}{\log_2(1+1)} + \frac{1}{\log_2(2+1)} + \frac{0}{\log_2(3+1)} + \frac{1}{\log_2(4+1)} = \frac{2}{1} + \frac{1}{1.59} + 0 + \frac{1}{2.32} = 2 + 0.63 + 0.43 = 3.06$$

$$\text{DCG@K}_2 = \frac{2}{\log_2(1+1)} + \frac{0}{\log_2(2+1)} + \frac{1}{\log_2(3+1)} + \frac{1}{\log_2(4+1)} = \frac{2}{1} + 0 + \frac{1}{2} + \frac{1}{2.32} = 2 + 0.5 + 0.43 = 2.93$$

$$\text{DCG@K}_2 = \frac{1}{\log_2(1+1)} + \frac{0}{\log_2(2+1)} + \frac{2}{\log_2(3+1)} + \frac{1}{\log_2(4+1)} = \frac{1}{1} + 0 + \frac{2}{2} + \frac{1}{2.32} = 1 + 1 + 0.43 = 2.43$$

Also, IDCG@K value is given by

$$\text{IDCG@K} = \frac{2}{\log_2(1+1)} + \frac{1}{\log_2(2+1)} + \frac{1}{\log_2(3+1)} + \frac{0}{\log_2(4+1)} = \frac{2}{1} + \frac{1}{1.59} + \frac{1}{2} + 0 = 2 + 0.63 + 0.5 = 3.13$$

Accordingly, the nDCG@K for each list is calculated as

$$\text{nDCG@K}_1 = \frac{\text{DCG@K}_1}{\text{IDCG@K}} = \frac{3.06}{3.13} = 0.98$$

$$\text{nDCG@K}_2 = \frac{\text{DCG@K}_2}{\text{IDCG@K}} = \frac{2.93}{3.13} = 0.94$$

$$\text{nDCG@K}_3 = \frac{\text{DCG@K}_3}{\text{IDCG@K}} = \frac{2.43}{3.13} = 0.78$$

Model Performance

To develop a machine learning model, a dataset is used to train the model, referred to as the **training set**. The model is then evaluated on previously unseen data, known as the **test set**. A robust model is one that not only performs well on the training data but also generalizes effectively to the test data. To assess the quality of a model, two fundamental concepts are often considered: bias and variance.

4.1 Bias

In simple terms, bias refers to the error incurred by a model when predicting values. A high bias indicates that the model is not trained well, meaning it is too simple to capture the underlying patterns in the data. On the other hand, a low bias suggests that the model is more flexible and better able to learn the relationship between the input features and the corresponding labels or output values. Given the actual values Y, and the predicted values $\hat{Y}$, the bias is defined as [12]

$$\text{Bias} = \mathbb{E}[\hat{Y}] - Y, \tag{4.1}$$

where $\mathbb{E}[\hat{Y}]$ is the expected value of the predictions. High bias leads to poor model performance, a phenomenon known as **underfitting**, where the model fails to capture the underlying patterns in the training data. As a result, it performs poorly even during the training phase [12].

4.2 Variance

Mathematically, variance measures the spread of data around its mean. In machine learning, variance quantifies the sensitivity of a model's predictions to different subsets of the training data. In simple terms, it indicates how much a model's predictions change when trained on different datasets. Formally, variance measures the expected squared deviation of the model's predictions from their mean, expressed as [12]

$$\text{Variance} = \mathbb{E}\left[(\hat{Y} - \mathbb{E}[\hat{Y}])^2\right] \tag{4.2}$$

A model exhibiting high variance along with low bias is said to suffer from **overfitting**, that is a common problem in ML. Overfitting occurs when the model memorizes the training data, including noise, rather than learning the underlying patterns. As a result, when presented with unseen data, the model fails to generalize, leading to a significant drop in performance [12].

4.3 Overfitting and Underfitting

As mentioned earlier, overfitting occurs when a model has low bias but high variance. In contrast, underfitting refers to the situation wherein the model exhibits high bias and low variance. Figure 4.1 shows the bias-variance trade-off diagram [12].

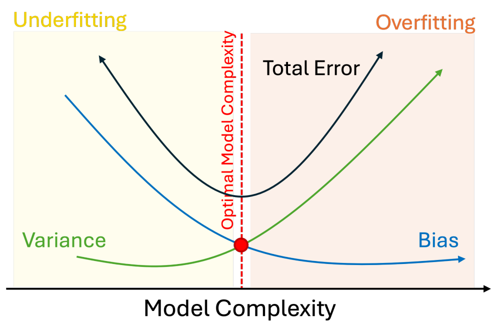


Figure 4.1: Bias-variance trade-off [12].

4.3.1 Methods to Address

There are different methods and techniques to overcome both overfitting and underfitting. In the rest, we review some of them. Before reviewing the corresponding methods, we discuss two important techniques, namely regularization and cross validation, in details [12, 13].

Regularization

Regularization methods, such as L1-norm and L2-norm, mitigate the overfitting by adding a penalty term to the weights during updating [13].

The L1-norm, also known as Least Absolute Shrinkage and Selection Operator (Lasso) regression, introduces the absolute value of the coefficient magnitudes as a penalty term to the loss function $\mathcal{L}$. Formally, it is defined as [13]

$$\mathcal{L}_{L1} = \mathcal{L} + \lambda \sum_{i}^{M} |w_i| = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{i}^{M} |w_i|, \tag{4.3}$$

where N and M are total number of samples and features, respectively; λ is the regularization parameter; w shows the weight (or coefficient); and y_i and $\hat{y}_i$ represent the actual and predicted values, respectively. The L1 penalty encourages sparsity by shrinking less important weights toward zero that allows only the most significant features to contribute to the model during training [13].

The L2-norm, also known as Ridge regression or Euclidean norm, adds the square of the coefficient magnitudes as a penalty term to the loss function $\mathcal{L}$. Formally, it is defined as [13]

$$\mathcal{L}_{L1} = \mathcal{L} + \lambda \sum_{i}^{M} w_i^2 = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 + \lambda \sum_{i}^{M} w_i^2 \tag{4.4}$$

Unlike L1 regularization, which enforces sparsity by driving some coefficients exactly to zero, L2 regularization uniformly penalizes large weights, leading to smaller but nonzero coefficients. This helps prevent overfitting by reducing model complexity while maintaining all features' contributions. However, it is highly sensitive to outliers in the data. Any outlier in the data can increase the error. To minimize this large penalty, the model will shift its parameters toward the outlier. Thus, the model is vulnerable to an improper fitting [13].

In summary, L1 and L2 regularization each offer distinct advantages depending on the characteristics of the data and the model. L1 regularization encourages sparsity by driving less important coefficients exactly to zero that makes it useful for feature selection in high-dimensional datasets. L2 regularization, on the other hand, penalizes large weights more smoothly that yields smaller but nonzero coefficients and promoting weight stability across correlated features. To leverage the benefits of both methods, Elastic Net regularization combines the L1 and L2 penalties as follows [13]:

$$\mathcal{L}_{ElasticNet} = \mathcal{L} + \lambda_1 \sum_{i}^{M} |w_i| + \lambda_2 \sum_{i}^{M} w_i^2, \tag{4.5}$$

where λ_1 and λ_2 control the contribution of the L1 and L2 terms, respectively. Elastic Net is particularly effective when dealing with datasets that contain highly correlated features or when both feature selection and regularization are desired simultaneously [13].

Cross-Validation

Cross-validation is a widely used technique to mitigate overfitting by providing a more reliable estimate of a model's generalization performance. The most common type is **k-fold cross-validation**, where the training dataset is divided into k equal-sized subsets, known as *folds* [14].

The training procedure is repeated k times. In each iteration, one fold is used as the validation set, while the remaining $(k-1)$ folds are combined to form the training set. This ensures that every data point is used for both training and validation exactly once. Importantly, each iteration is independent, and the model learns separate weights corresponding to that iteration [14].

After completing all k iterations, the validation results from each fold are averaged to obtain the final performance estimate. This approach effectively reduces both **bias** and **variance**, thereby improving the model's ability to generalize and preventing overfitting [14].

Alleviate Underfitting and Overfitting

- **Underfitting:** To cope with the underfitting problem [12]:
  - Increase the model complexity: More complex models are able to learn underlying pattern of data. One efficient way is to use complex model (such as deep neural network models) rather than simpler model (such as linear/logistic regression) [12].
  - Add more features: Increasing the dimensionality of data, in terms of features, can help models to learn more complex patterns. Add more relevant features using feature engineering process [12].
  - Reduce regularization: Regularization techniques, such as the L1-norm (Lasso) and L2-norm (Ridge or Euclidean norm), play a crucial role in preventing overfitting by penalizing overly complex models. However, excessive regularization can lead to underfitting, as the model becomes too constrained to capture important patterns in the data. To mitigate this issue, the regularization strength can be reduced by using a smaller penalty coefficient [12].
  - Increase training duration: Increasing the number of epochs for training the model, let the model to learn more effectively [12].
- **Overfitting:** To prevent the model from overfitting [12]:
  - Increase training data: Increasing the dimensionality of training data, in terms of the number of samples, can help mitigate the overfitting problem. With more data, the model has greater opportunity to generalize and learn the underlying patterns, rather than memorizing the training examples [12].
  - Reduce model's complexity: Sometimes, the underlying data patterns are simple enough to be effectively learned by simpler models rather than highly complex ones. Complex models are not always the optimal choice; they tend to perform well only when the data are intricate and involve a large number of features, making pattern learning more challenging. To reduce the risk of overfitting in such cases, it is advisable to use simpler algorithms or architectures, decrease the number of layers, or reduce the number of neurons in the model [12].

  - Use regularization: Regularization techniques, such as the L1-norm (Lasso) and L2-norm (Ridge or Euclidean norm), play a crucial role in preventing overfitting by penalizing overly complex models [12].
  - Use dropout: Dropout is a regularization technique in neural networks that involves randomly deactivating a subset of neurons during training. This prevents the model from becoming overly reliant on specific neurons or pathways, thereby improving its ability to generalize to unseen data [12].
  - Implement early stopping: Early stopping is a regularization technique that halts training when the model's performance on a separate validation set begins to deteriorate, even if its performance on the training set continues to improve. This approach helps prevent overfitting by stopping the learning process before the model starts to memorize noise in the training data [12].
  - Perform feature selection: Irrelevant or redundant features can cause the model to overfit to noise. Removing those features improves generalization and the model's performance [12].
  - Deploy data augmentation: When the diversity of a dataset is low, a model is more likely to memorize the data patterns rather than learning them. Data augmentation is a technique that addresses this issue by increasing dataset diversity. To this end, it generates new data points through transformations of existing samples [12].
  - Apply cross-validation: Cross-validation guarantees that every data point in the training set is used for both training and validation exactly once. Therefore, it avoids the model to be trained over fixed sets. As a result, both bias and variance are decreased [12].

4.4 Vanishing and Exploding Gradient

Gradients are the key parameters in model training. During backpropagation in neural network (NN) models, the gradient of the loss is computed w.r.t. the weights and biases separately. Then, based on the optimization algorithm, such as Stochastic Gradient Descent (SGD) or Adaptive Moment Estimation (Adam)-both driven by the gradient descent mechanism, the weights and biases are updated layer by layer. This implies that in each layer, the weights are multiplied recursively by gradient-based updates so that the overall loss is minimized across the network. However, if these multipliers become excessively small or large, the learning process encounters major difficulties. The former issue is referred to as **gradient vanishing**, while the latter is known as **gradient exploding** [15].

4.4.1 Gradient Vanishing

One common challenge in neural networks, particularly recurrent neural networks (RNNs), that often leads to underfitting is **gradient vanishing**. During backpropagation, if the gradients are repeatedly multiplied by values smaller than one, they shrink exponentially as they propagate backward through the layers. Consequently, the weights in the early layers approach zero, preventing the model from learning useful patterns. This phenomenon results in slow or completely stalled learning [15].

4.4.2 Gradient Exploding

The opposite of gradient vanishing is the **gradient exploding** problem. It occurs when the gradients grow exponentially during backpropagation—often in deep or recurrent networks. This causes the model parameters to oscillate or diverge. When gradient exploding happens, the loss fails to converge, and the model parameters may take extremely large values that results in unstable training or numerical overflow. Gradient exploding typically arises when [15]:

- The network is very deep, and the product of large gradients across multiple layers accumulates exponentially [15].
- Improper weight initialization leads to excessively large parameter updates [15].
- The learning rate is too high that amplifies each weight update [15].

4.4.3 Techniques to Cope with Gradient Vanishing and Exploding

Before diving into the gradient vanishing and exploding problems, it is essential to review several fundamental techniques that can effectively alleviate both issues.

Weight Initialization

Avoiding random initialization of weights and adopting principled initialization strategies can significantly reduce the probability of both vanishing and exploding gradients. Techniques such as **Kaiming initialization** and **Xavier initialization** set the initial weights at appropriate scales, thereby maintaining stable gradient propagation during early training [16, 17].

- **Kaiming Initialization:** Kaiming (or He) initialization is suitable for layers that use the *ReLU* activation function. It prevents vanishing gradients by assigning a larger variance to the initial weights. There are two common variants [16]:
  - **Kaiming Normal Initialization:** Weights are drawn from a normal distribution with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{n_{in}}}$:
  $$w_i \sim \mathcal{N}(0, \sqrt{\tfrac{2}{n_{in}}}),$$
  where n_{in} is the number of input units to the layer.
  - **Kaiming Uniform Initialization:** Weights are drawn from a uniform distribution in the range:
  $$w_i \sim U\left(-\sqrt{\tfrac{6}{n_{in}}}, \sqrt{\tfrac{6}{n_{in}}}\right),$$
  that ensures variance preservation across layers.

- **Xavier Initialization:** Xavier (or Glorot) initialization is more suitable for layers using *Sigmoid* or *Tanh* activation functions. It aims to keep the variance of activations and gradients consistent across layers. Two variants exist [17]:
  - **Normal Xavier Initialization:** Weights are drawn from a normal distribution with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{n_{in}+n_{out}}}$:
  $$w_i \sim \mathcal{N}\left(0, \sqrt{\tfrac{2}{n_{in}+n_{out}}}\right),$$
  where n_{in} and n_{out} denote the number of input and output units, respectively.
  - **Uniform Xavier Initialization:** Weights are drawn from a uniform distribution within the range:
  $$w_i \sim U\left(-\sqrt{\tfrac{6}{n_{in}+n_{out}}}, \sqrt{\tfrac{6}{n_{in}+n_{out}}}\right)$$

Normalization

Normalizing inputs to each layer stabilizes gradient flow, accelerates convergence, and mitigates both vanishing and exploding gradients. Two widely used normalization methods are described below.

- **Batch Normalization:** This technique normalizes the input activations within each mini-batch, ensuring zero mean and unit variance. Consequently, the input to each layer becomes dependent on the batch statistics. Although highly effective for large batch sizes, it can be less stable for small batches or recurrent neural networks (RNNs), where batch statistics fluctuate significantly [18].
- **Layer Normalization:** This approach normalizes activations across the features of each individual sample, independently of other samples in the batch. This makes it particularly suitable for small or variable batch sizes, as in RNNs and natural language processing tasks [19].

Additional Strategies for Gradient Stabilization

Several other strategies can further mitigate the vanishing and exploding gradient problems [15]:

- **Alternative Activation Functions:** Saturating activation functions such as *Sigmoid* and *Tanh* are prone to vanishing gradients because their derivatives approach zero for large input magnitudes. In contrast, *ReLU* maintains gradients close to 1 for positive inputs, preventing exponential shrinkage [15].
- **Residual and Skip Connections:** Architectures such as ResNets incorporate shortcut connections that allow gradients to bypass intermediate layers, ensuring more effective propagation to earlier layers [15].
- **Gradient Clipping:** This widely used technique constrains the gradient norm to a predefined threshold (e.g., by rescaling gradients when their magnitude exceeds a limit), thereby preventing instability due to gradient explosion [15].
- **Learning Rate Adjustment:** A smaller learning rate ensures smoother parameter updates, helping avoid overshooting and gradient amplification during optimization [15].
- **Gated Architectures (for RNNs):** In sequential models, gating mechanisms such as those in Long Short-Term Memory (LSTM) regulate information and gradient flow over time, effectively mitigating both vanishing and exploding gradients in long sequences [15].

In practice, both vanishing and exploding gradients can coexist in different parts of a deep network. Therefore, combining techniques such as careful initialization, normalization, skip connections, and adaptive optimizers (e.g., Adam) is essential to achieve stable and efficient training.

Supervised Learning

Supervised learning is referred to as the methods wherein the training data are labeled. Thus, models are developed for the supervised learning scenarios are categorized into *classification* and *regression*.

The classification is divided into **binary classification** and **multi-class** datasets. The former contains data points with only two possible labels, commonly represented as 0 and 1, such as the **Breast Cancer** dataset in *scikit-learn*, where each instance indicates whether a patient has cancer or not. The latter includes datasets with more than two classes, such as the **Iris** dataset in *scikit-learn*, which comprises three distinct classes. The goal of a classifier is to correctly categorize data samples to achieve high predictive performance, measured by **accuracy**, or by **precision** and **recall** in the case of imbalanced datasets.

On the other hand, regression models focus on predicting **continuous numerical values** for the input data. The goal of regression is to **minimize prediction loss (or error)**, i.e., the difference between the true (ground-truth) values and the predicted values (see Sec. 3.2 for the definition of error).

The most common algorithms used in supervised learning include *Logistic/Linear Regression*, **Decision Tree**, **Random Forest**, **Gradient Boosting**, **Support Vector Machines (SVM)**, **K-Nearest Neighbors (KNN)**, and **Bayesian algorithms**, all of which can be applied to both classification and regression tasks. In the remainder of this section, we provide a detailed review of these algorithms[1].

[1] **Note:** The main reference for the current section is the ***scikit-learn*** [20].

5.1 Logistic Regression

Despite its name, logistic regression is actually a classification method. It can be applied to both binary and multi-class classification tasks.

5.1.1 Binary Classification

Logistic regression is a linear-based model that follows the linear equation in Eq. 5.1 to predict values.

$$z = \omega x + b, \tag{5.1}$$

where ω is the weight of the model, x represents the independent data, and b shows the bias. For the classification purposes, the output z is fed into the *sigmoid* function as

$$\hat{z} = \frac{1}{1 + e^{-z}} \tag{5.2}$$

Finally, $\hat{y}$ represents the corresponding label so that

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{z} < 0.5 \\ 1 & \text{otherwise} \end{cases} \tag{5.3}$$

5.1.2 Multi-class Classification

In multi-class task, it is assumed that there are C classes. Thus, for each data point $x_i \in X$, where X is the set of data points, the output z_i is fed into the softmax function to predict the corresponding probability, i.e., the probability that z_i belongs to class $c \in C$. Therefore, we have

$$\hat{z_i} = \frac{exp(z_i)}{\sum_{c=1}^{C} exp(z_c)} \tag{5.4}$$

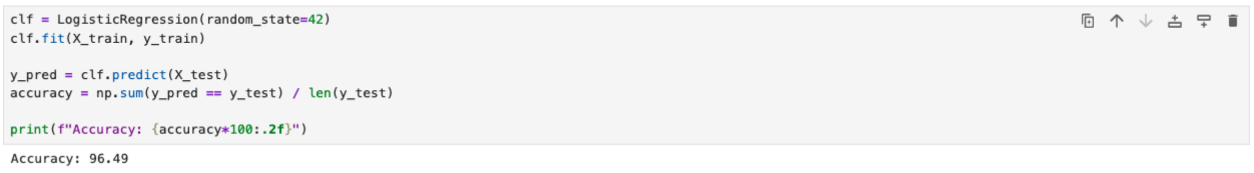
Assuming that our data is Independent and Identically Distributed (IID), the loss for a single data point is given by

$$\mathcal{L} = -\sum_{c=1}^{C} y_c \log(z_c) \tag{5.5}$$

5.1.3 Implementation

The machine learning logistic regression model for binary classification in Python was developed from scratch following the guidelines provided in [21]. The complete implementation script is available on Logistic Regression (Binary) from Scratch. Also, the corresponding model for the multi-class classification was developed from scratch, that is available on Logistic Regression (Multi-Class) from Scratch.

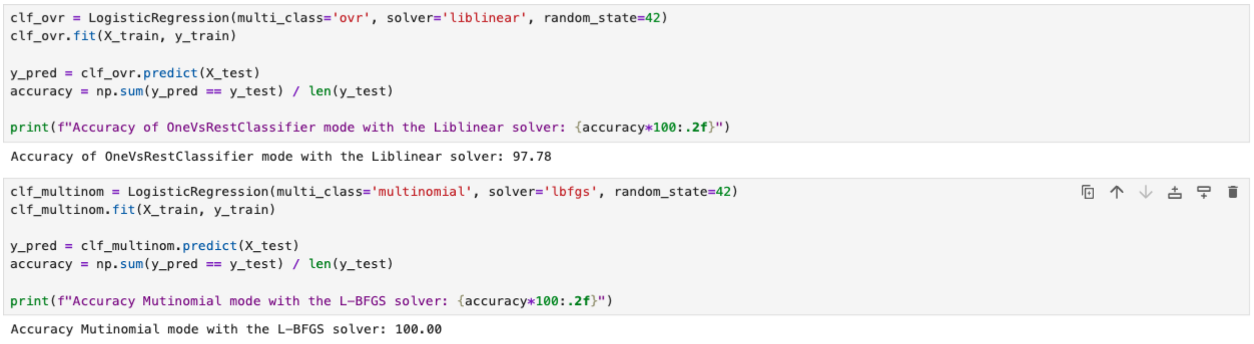
We developed a logistic regression classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process and the evaluation results of the model. Notably, the model is imported from ***sklearn**.linear_model*. The complete implementation script is available on the GitHub page Logistic Regression for Binary Classification.



```
clf = LogisticRegression(random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 96.49
```

We also conducted the experiment using a multi-class logistic regression model. For this purpose, we utilized the **Iris** dataset and trained the model under two different configurations. First, we fitted the model using the **One-vs-Rest (OvR)** classification strategy with the **Liblinear** solver, which supports both L1 and L2 regularization. In the second configuration, we developed the model using the **Multinomial** classification approach with the **Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS)** solver. The corresponding results are presented in the following screenshot. The complete implementation script is available on the GitHub page Logistic Regression for Multi-Class Classification.



```
clf_ovr = LogisticRegression(multi_class='ovr', solver='liblinear', random_state=42)
clf_ovr.fit(X_train, y_train)

y_pred = clf_ovr.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy of OneVsRestClassifier mode with the Liblinear solver: {accuracy*100:.2f}")
```

```
Accuracy of OneVsRestClassifier mode with the Liblinear solver: 97.78
```

```
clf_multinom = LogisticRegression(multi_class='multinomial', solver='lbfgs', random_state=42)
clf_multinom.fit(X_train, y_train)

y_pred = clf_multinom.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy Mutinomial mode with the L-BFGS solver: {accuracy*100:.2f}")
```

```
Accuracy Mutinomial mode with the L-BFGS solver: 100.00
```

5.2 Linear Regression

Linear regression is a linear-based model that follows the linear equation in Eq. 5.6 to predict values.

$$\hat{y} = \omega x + b, \tag{5.6}$$

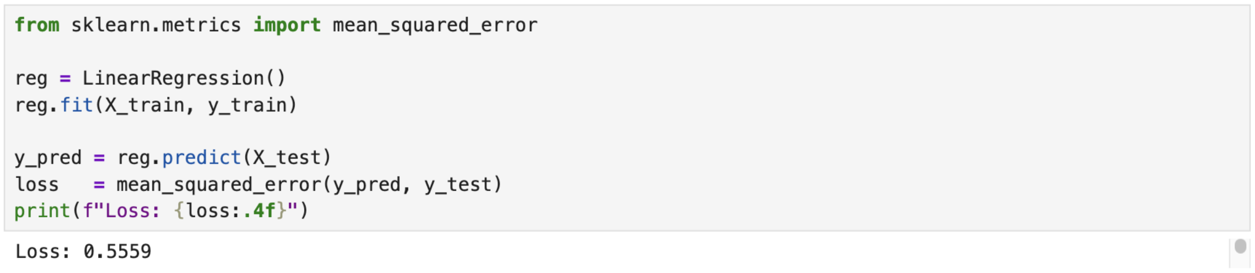
where ω is the weight of the model, x represents the independent data, and b shows the bias. The mean squared error (MSE) or mean absolute error (MAE) are used to evaluate the predictions. For example, relying on MSE, we have

$$\mathcal{L} = \frac{\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}{N} \tag{5.7}$$

5.2.1 Implementation

The machine learning linear regression model in Python was developed from scratch following the guidelines provided in [22]. The complete implementation script is available on Linear Regression from Scratch.

We developed a linear regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process the regressor, following by the evaluation results. Notably, the model is imported from ***sklearn***.*linear_model*. The complete implementation script is available on the GitHub page Linear Regression.



```
from sklearn.metrics import mean_squared_error

reg = LinearRegression()
reg.fit(X_train, y_train)

y_pred = reg.predict(X_test)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.5559
```

5.3 Decision Tree

A decision tree is an algorithm that recursively splits a dataset into subsets based on its features. The splitting criterion is typically determined using one of two metrics, i.e., *Information Gain (IG)* or *Gini Index*.

- **Information Gain:** IG is based on *entropy*, which quantifies the level of uncertainty or randomness in the information being processed. IG measures the reduction in entropy that results from partitioning the dataset according to a given feature. To calculate the IG, we have

$$\mathrm{IG(D,\ A) = H(D) - H(D|A)}, \tag{5.8}$$

where D and A are the dataset and the feature, respectively; H(D) represents the entropy of the dataset; and H(D|A) shows the entropy of the dataset w.r.t. the feature A (conditional entropy). The entropy of a set S, where {A, D ∈ S} is given by

$$\mathrm{H(S)} = -\sum_{x\in S} p_i(x)\log(p_i(x)), \tag{5.9}$$

where p_i is proportion of the i-th class in the set.

- **Gini Index:** Gini Index measures the probability that a randomly chosen instance would be misclassified if it were labeled according to the class distribution in the dataset. Therefore, we have

$$\mathrm{H(S)} = 1 - \sum_{x\in S} p_i^2(x), \tag{5.10}$$

where p_i is the probability of an instance belonging to class i.

Overall, an IG–based model aims to maximize IG, while a Gini Index–based model aims to minimize the Gini value. IG is generally more suitable for imbalanced datasets, whereas the Gini Index is computationally simpler to implement.

5.3.1 Numerical Example

Given the data in Table 5.1, we perform the decision tree algorithm based on IG.

Table 5.1: Example Data for Decision Tree [23].

Neighborhood	**No. of Rooms**	**Affordable**
West	3	Yes
West	5	Yes
West	2	Yes
East	3	Yes
East	4	Yes
East	6	No
East	5	No
East	2	Yes

First, we calculate the entropy of the data w.r.t. the labels (i.e., “Affordable” feature). Considering 6 data points labeled as “Yes” and 2 data points labeled as “No” (see Fig. 5.1), the probabilities of “Yes” (p_{yes}) and “No” (p_{no}) are equal to $p_{yes} = 6/8$ and $p_{no} = 2/8$. Therefore, we have

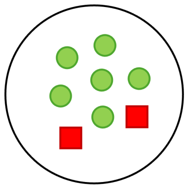


Figure 5.1: Root node for decision tree algorithm.

$$\begin{aligned}\mathrm{H(D)} = & -p(Y = yes)\log(p(Y = yes)) - \\ & p(Y = no)\log(p(Y = no)) \\ = & -\frac{6}{8}\log(\frac{6}{8}) - \frac{2}{8}\log(\frac{2}{8}) = 0.81\end{aligned}$$

In the next step, the data samples must be split based on the features, i.e., “Neighborhood” and “No. of Rooms”, separately. The corresponding IG needs to be calculated for each feature, and the feature with the highest IG is chosen as the splitting feature. Thus, we have

$$\begin{aligned}\mathrm{H(D|Neighborhood)} &= \mathrm{H(D|Neighborhood{=}West)} + \mathrm{H(D|Neighborhood{=}East)} \\ &= \mathrm{H(Y{=}yes|Neighborhood{=}West)} + \mathrm{H(Y{=}no|Neighborhood{=}West)} + \\ &\quad \mathrm{H(Y{=}yes|Neighborhood{=}East)} + \mathrm{H(Y{=}no|Neighborhood{=}East)} \\ &= -\frac{3}{8}\left(\frac{3}{3}\log(1) + \frac{0}{3}\log(0)\right) - \frac{5}{8}\left(\frac{3}{5}\log(\frac{3}{5}) + \frac{2}{5}\log(\frac{2}{5})\right) = 0.61\end{aligned}$$

$$\begin{aligned}\mathrm{H(D|No.\ of\ Rooms)} &= \mathrm{H(D|No.\ of\ Rooms{<}5)} + \mathrm{H(D|No.\ of\ Rooms} \geq 5) \\ &= \mathrm{H(Y{=}yes|No.\ of\ Rooms{<}5)} + \mathrm{H(Y{=}no|No.\ of\ Rooms{<}5)} + \\ &\quad \mathrm{H(Y{=}yes|No.\ of\ Rooms} \geq 5) + \mathrm{H(Y{=}no|No.\ of\ Rooms} \geq 5) \\ &= -\frac{5}{8}\left(\frac{5}{5}\log(1) + \frac{0}{5}\log(0)\right) - \frac{3}{8}\left(\frac{1}{2}\log(\frac{1}{2}) + \frac{2}{3}\log(\frac{2}{3})\right) = 0.35\end{aligned}$$

Therefore, IG for each group is calculated as

$$\begin{aligned}\mathrm{IG(D|Neighborhood)} &= 0.81 - 0.61 = 0.2 \\ \mathrm{IG(D|No.\ of\ Rooms)} &= 0.81 - 0.35 = 0.46\end{aligned}$$

Since IG(D|No. of Rooms) > IG(D|Neighborhood), we choose “No. of Rooms” as the splitting feature.

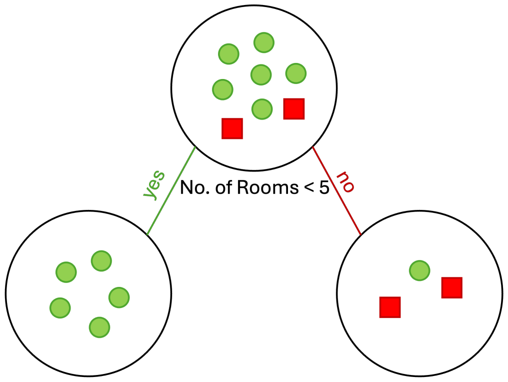


Figure 5.2: Decision tree after first splitting.

In the next round, since the entropy of the left child of the tree is zero (all samples belong to the same class, i.e., “Yes”), we split the right child of the tree w.r.t. the “Neighborhood”. Therefore, we have

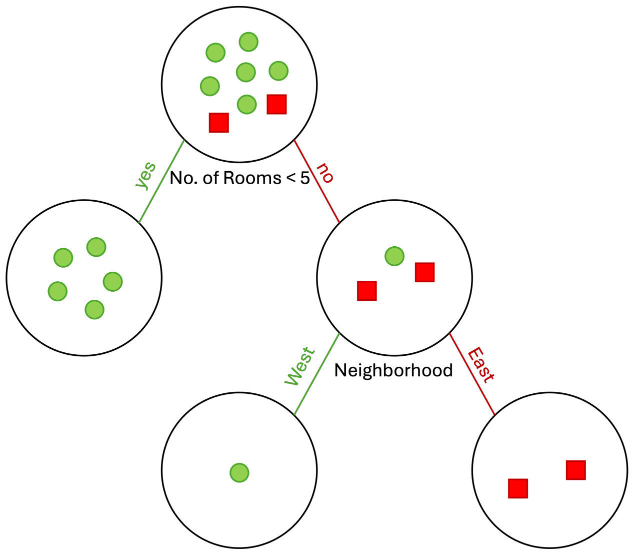


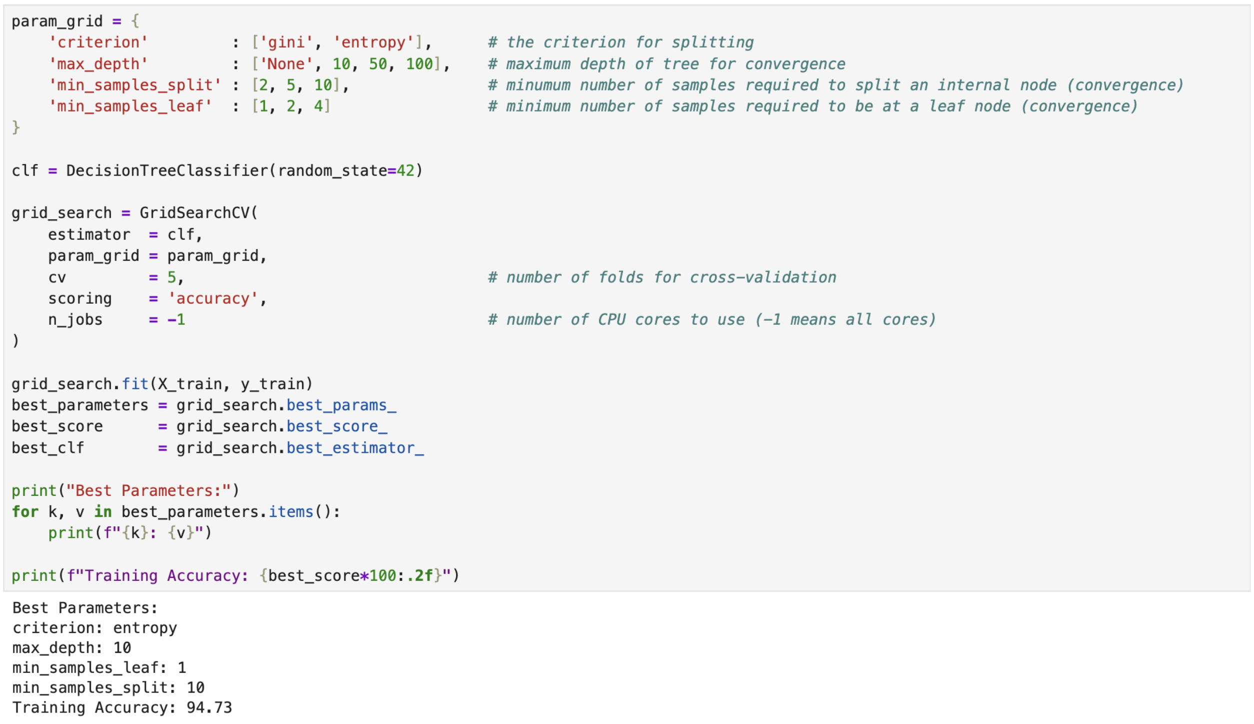
Figure 5.3: Decision tree after second splitting.

5.3.2 Implementation

The machine learning decision tree model in Python was developed from scratch following the guidelines provided in [23]. The complete implementation script is available on Decision Tree from Scratch.

Classifier

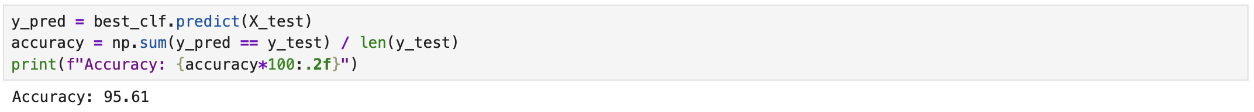
We developed a decision tree classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process of the classifier. The model was imported from ***sklearn**.tree*. Moreover, we applied **Grid Search Cross-Validation** to identify the optimal hyperparameters of the model. In this process, we defined ranges for various convergence criteria, such as *max_depth*, *min_samples_split*, *min_samples_leaf*, and the splitting criterion (i.e., *Gini Index* or *Information Gain*).



```
param_grid = {
    'criterion'         : ['gini', 'entropy'],       # the criterion for splitting
    'max_depth'         : ['None', 10, 50, 100],     # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],                # minumum number of samples required to split an internal node (convergence)
    'min_samples_leaf'  : [1, 2, 4]                  # minimum number of samples required to be at a leaf node (convergence)
}

clf = DecisionTreeClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator  = clf,
    param_grid = param_grid,
    cv         = 5,                                  # number of folds for cross-validation
    scoring    = 'accuracy',
    n_jobs     = -1                                  # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_clf        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Accuracy: {best_score*100:.2f}")
```

```
Best Parameters:
criterion: entropy
max_depth: 10
min_samples_leaf: 1
min_samples_split: 10
Training Accuracy: 94.73
```

After identifying the best classifier through Grid Search Cross-Validation, we evaluated its performance using the test dataset. The following screenshot presents the corresponding results. The complete implementation script is available on the GitHub page Decision Tree Classifier.

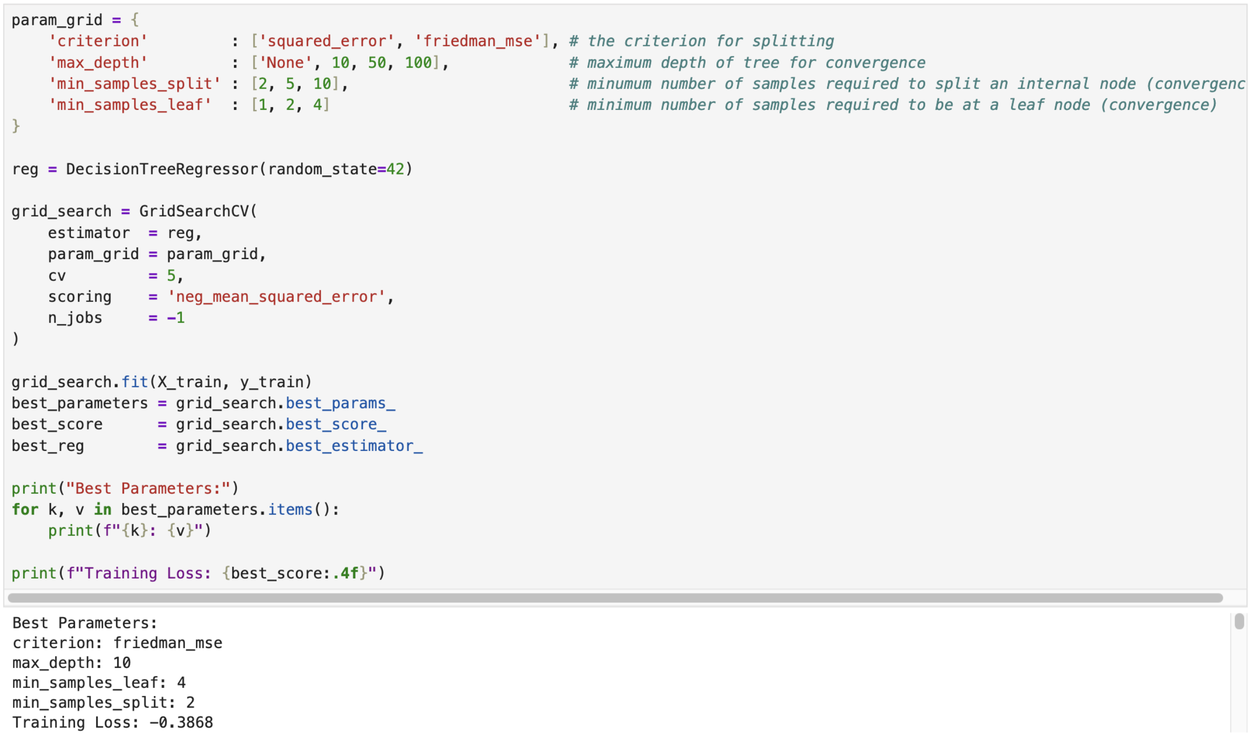


```
y_pred = best_clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 95.61
```

Regressor

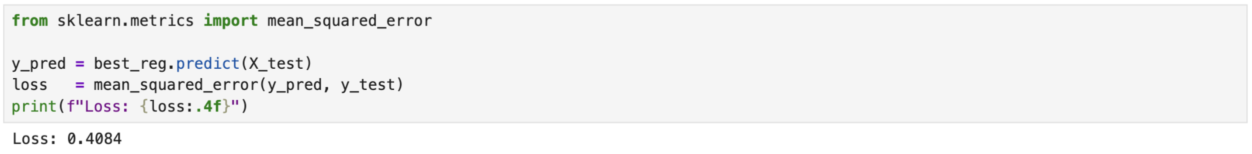
We developed a decision tree regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process of the regressor, where **Grid Search Cross-Validation** was employed to tune the optimal hyperparameters.



```
param_grid = {
    'criterion'        : ['squared_error', 'friedman_mse'], # the criterion for splitting
    'max_depth'        : ['None', 10, 50, 100],             # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],                       # minumum number of samples required to split an internal node (convergenc
    'min_samples_leaf'  : [1, 2, 4]                         # minimum number of samples required to be at a leaf node (convergence)
}

reg = DecisionTreeRegressor(random_state=42)

grid_search = GridSearchCV(
    estimator  = reg,
    param_grid = param_grid,
    cv         = 5,
    scoring    = 'neg_mean_squared_error',
    n_jobs     = -1
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_reg        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Loss: {best_score:.4f}")
```

```
Best Parameters:
criterion: friedman_mse
max_depth: 10
min_samples_leaf: 4
min_samples_split: 2
Training Loss: -0.3868
```

After training the regressor, we evaluated its performance on the test dataset. The following screenshot displays the corresponding results. The complete implementation script is available on the GitHub page Decision Tree Regressor.



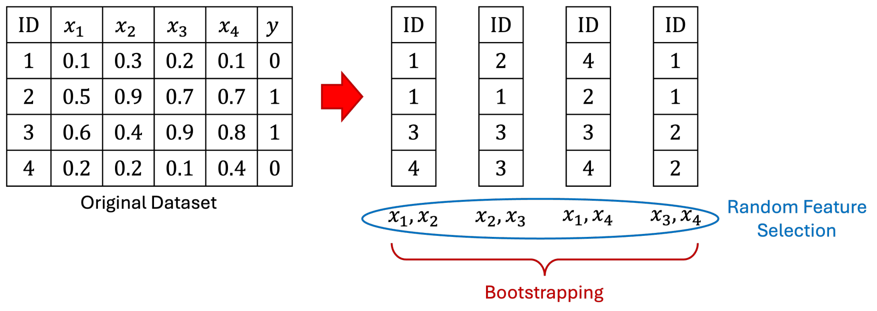
```
from sklearn.metrics import mean_squared_error

y_pred = best_reg.predict(X_test)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.4084
```

5.4 Random Forest

Random forest is an ensemble method that combines multiple decision trees, also known as a “forest”, to make predictions. It employs a **bootstrapping** mechanism, also known as **bagging**, in which multiple subsets of the original dataset are generated by sampling with replacement. Each subset contains the same number of samples as the original dataset, but individual data points may appear more than once.

For each subset, a random subset of features is also selected, typically with the number of features close to either $\log_2 M$ or $\sqrt{M}$, where M is the total number of features. A separate decision tree is then trained on each bootstrapped dataset. Figure 5.4 illustrates an example of bootstrapping and random feature selection for a dataset with four samples and four features.



ID	x_1	x_2	x_3	x_4	y
1	0.1	0.3	0.2	0.1	0
2	0.5	0.9	0.7	0.7	1
3	0.6	0.4	0.9	0.8	1
4	0.2	0.2	0.1	0.4	0

ID	ID	ID	ID
1	2	4	1
1	1	2	1
3	3	3	2
4	3	4	2
x_1, x_2	x_2, x_3	x_1, x_4	x_3, x_4

Figure 5.4: An example of bootstrapping and random feature selection in Random Forest.

After training, the results from all trees are aggregated to produce the final prediction. For **classification** tasks, the ensemble uses **majority voting**, while for **regression** tasks, it computes the **average** of the individual tree predictions. Key characteristics of Random Forest include:

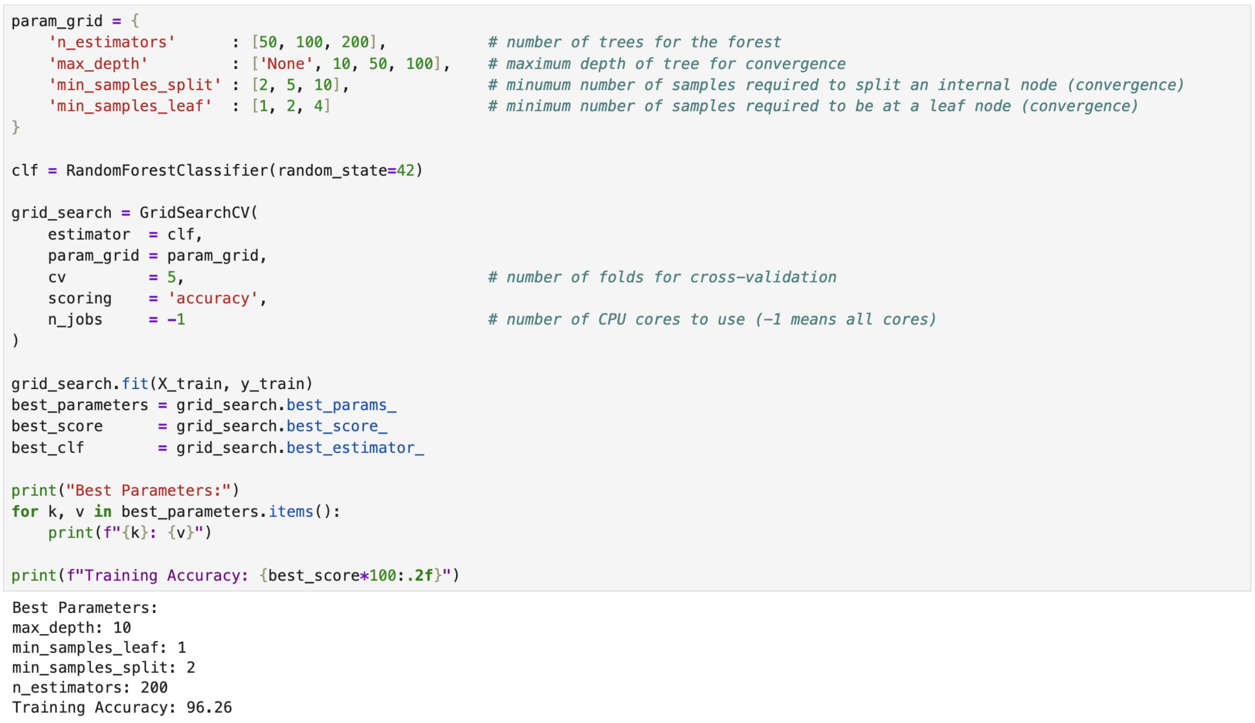
- Since Random Forest employs both **Bootstrapping** and **Aggregation**, it is commonly referred to as a **Bagging** method.
- The randomness in the algorithm originates from two sources: the random sampling of data points during bootstrapping and the random selection of features for each tree.
- Random Forest is more robust and less prone to overfitting:
  - Bootstrapping alleviates overfitting by ensuring that each tree is trained on a different subset of data.
  - Random feature selection ensures diversity among trees by producing different decision boundaries. This reduces variance and helps prevent overfitting.
- Compared to Gradient Boosting, Random Forest is faster to train and easier to tune, as its trees can be trained in parallel.

5.4.1 Implementation

The machine learning random forest model in Python was developed from scratch following the guidelines provided in [24]. The complete implementation script is available on Random Forest from Scratch.

Classifier

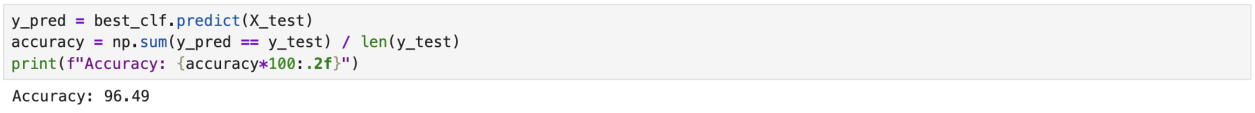
We developed a random forest classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process of the classifier. The model was imported from ***sklearn**.ensemble*. Moreover, we applied **Grid Search Cross-Validation** to identify the optimal hyperparameters of the model. In this process, we defined ranges for various parameters, such as *n_estimators* (no. of trees), *max_depth*, *min_samples_split*, and *min_samples_leaf*.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],          # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],   # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],              # minumum number of samples required to split an internal node (convergence)
    'min_samples_leaf'  : [1, 2, 4]                # minimum number of samples required to be at a leaf node (convergence)
}

clf = RandomForestClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator  = clf,
    param_grid = param_grid,
    cv         = 5,                                # number of folds for cross-validation
    scoring    = 'accuracy',
    n_jobs     = -1                                # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_clf        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Accuracy: {best_score*100:.2f}")
```

```
Best Parameters:
max_depth: 10
min_samples_leaf: 1
min_samples_split: 2
n_estimators: 200
Training Accuracy: 96.26
```

After identifying the best classifier through Grid Search Cross-Validation, we evaluated its performance using the test dataset. The following screenshot presents the corresponding results. The complete implementation script is available on the GitHub page Random Forest Classifier.

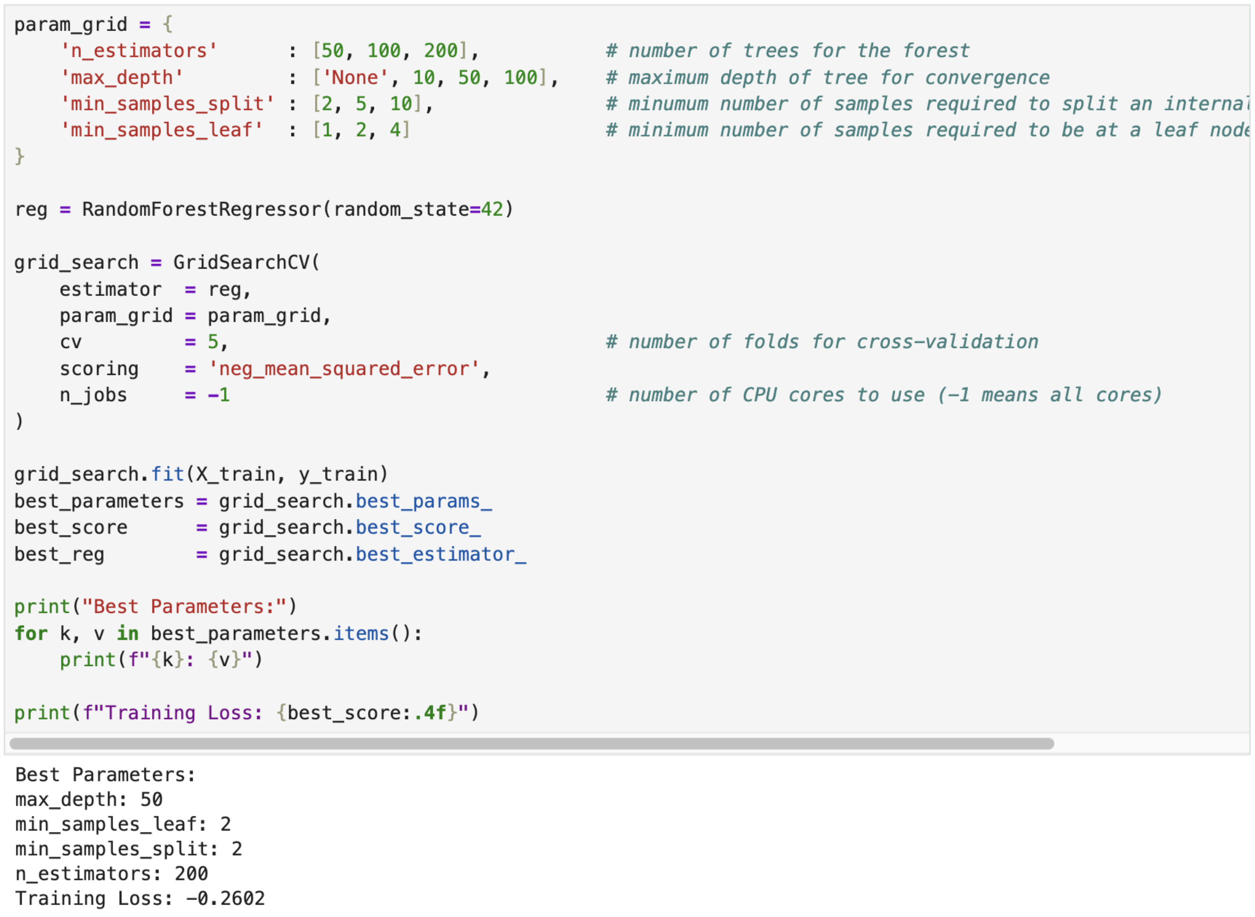


```
y_pred = best_clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 96.49
```

Regressor

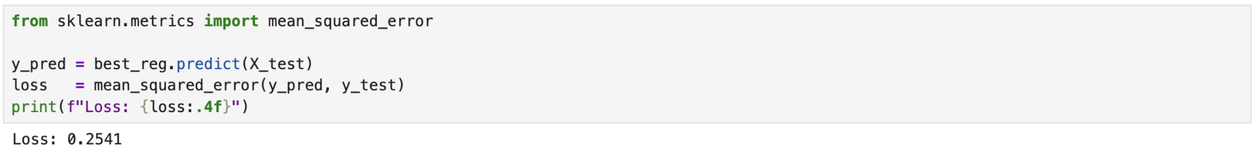
We developed a random forest regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process of the regressor, where **Grid Search Cross-Validation** was employed to tune the optimal hyperparameters.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],           # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],    # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],               # minumum number of samples required to split an internal
    'min_samples_leaf'  : [1, 2, 4]                 # minimum number of samples required to be at a leaf node
}

reg = RandomForestRegressor(random_state=42)

grid_search = GridSearchCV(
    estimator  = reg,
    param_grid = param_grid,
    cv         = 5,                                 # number of folds for cross-validation
    scoring    = 'neg_mean_squared_error',
    n_jobs     = -1                                 # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_reg        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Loss: {best_score:.4f}")
```

```
Best Parameters:
max_depth: 50
min_samples_leaf: 2
min_samples_split: 2
n_estimators: 200
Training Loss: -0.2602
```

After training the regressor, we evaluated its performance on the test dataset. The following screenshot displays the corresponding results. The complete implementation script is available on the GitHub page Random Forest Regressor.



```
from sklearn.metrics import mean_squared_error

y_pred = best_reg.predict(X_test)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.2541
```

5.5 Gradient Boosting

Unlike Random Forest, which constructs independent trees on bootstrap samples to reduce variance and improve generalization, **Gradient Boosting** builds trees sequentially, where each new tree aims to correct the errors made by the previous ones to achieve higher predictive accuracy.

Gradient Boosting enhances the final prediction function, $F(x)$, by combining M **weak learners**. In this approach, each subsequent weak learner is trained to minimize the residual errors (mistakes) of the preceding learners, and the aggregation of all weak learners produces a strong predictive model. Assuming $f_i(x)$ represents the i-th weak learner, the overall model is expressed as [25]

$$F(x) = \sum_{i=1}^{M} f_i(x)$$

To implement this, the algorithm first initializes a loss function and an initial weak learner. The loss function must be **differentiable**, as Gradient Boosting is a *gradient-based* optimization procedure. The algorithm begins with an extremely weak learner, $f_1(x)$ [25].

For each learner $f_j(x)$, where $j = 1, \ldots, M$, the algorithm computes the **negative gradient** of the loss function w.r.t. the predictions, i.e., [25]

$$\hat{r}_{ji} = -\frac{\partial \mathcal{L}(y_i, \hat{y}_i)}{\partial \hat{y}_i}\bigg|_{\hat{y}_i = F_j(x_i)},$$

where y_i and $\hat{y}_i$ are the actual and predicted values corresponding to the input x_i, respectively [25].

Next, the new weak learner, $f_{j+1}(x)$, is trained on the residuals $\hat{r}_j$ (i.e., the computed gradients) along with the original input data X. The contribution of this new weak learner, denoted as $\hat{\gamma}_{j+1}$, is obtained by solving [25]

$$\hat{\gamma}_{j+1} = \arg\min_{\gamma} \sum_{i=1}^{N} \mathcal{L}\left(y_i, F_j(x_i) + \gamma f_{j+1}(x_i)\right)$$

The model is then updated as [25]

$$F_{j+1}(x) = F_j(x) + \hat{\gamma}_{j+1} f_{j+1}(x)$$

This process continues iteratively until the algorithm converges to the optimal solution (i.e., the minimum loss) [25].

Comparison with Random Forest:

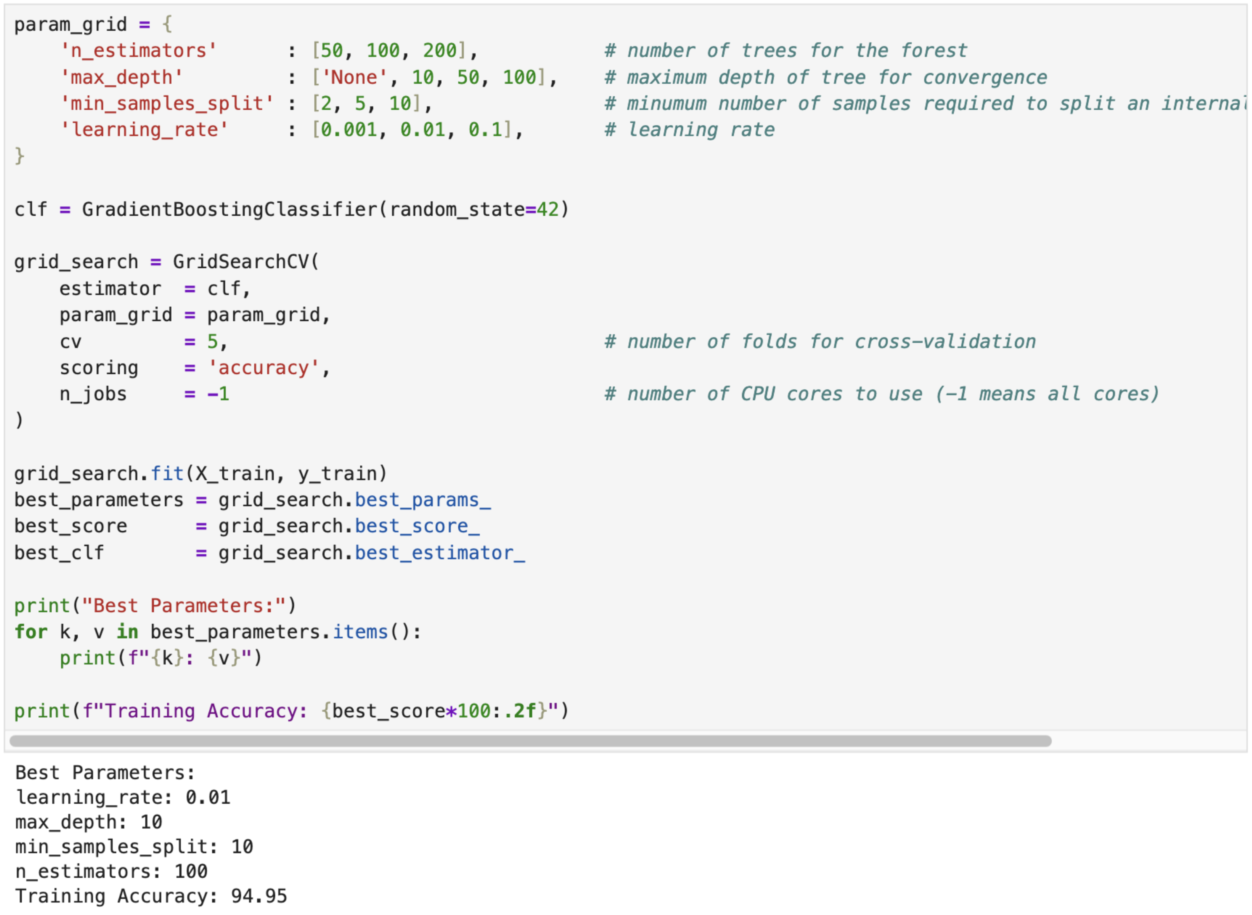
- Since each weak learner in Gradient Boosting is trained to correct the errors of its predecessors, it often achieves **higher predictive accuracy** than Random Forest.
- However, Gradient Boosting is **slower to train** and **more complex to tune**, as it involves more hyperparameters.
- It is also **more susceptible to overfitting** if not properly regularized (e.g., via learning rate, subsampling, or tree depth constraints).

5.5.1 Implementation

The machine learning random fores model in Python was developed from scratch. The complete implementation script is available on Gradient Boosting from Scratch.

Classifier

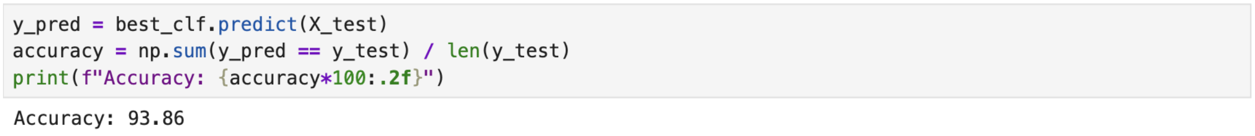
We developed a gradient boosting classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process of the classifier. The model was imported from ***sklearn***.*ensemble*. Moreover, we applied **Grid Search Cross-Validation** to identify the optimal hyperparameters of the model. In this process, we defined ranges for various parameters, such as *n_estimators* (no. of trees), *max_depth*, *min_samples_split*, and *learning_rate*.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],          # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],   # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],              # minumum number of samples required to split an interna
    'learning_rate'     : [0.001, 0.01, 0.1],      # learning rate
}

clf = GradientBoostingClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator  = clf,
    param_grid = param_grid,
    cv         = 5,                                # number of folds for cross-validation
    scoring    = 'accuracy',
    n_jobs     = -1                                # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_clf        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Accuracy: {best_score*100:.2f}")
```

```
Best Parameters:
learning_rate: 0.01
max_depth: 10
min_samples_split: 10
n_estimators: 100
Training Accuracy: 94.95
```

After identifying the best classifier through Grid Search Cross-Validation, we evaluated its performance using the test dataset. The following screenshot presents the corresponding results. The complete implementation script is available on the GitHub page Gradient Boosting Classifier.

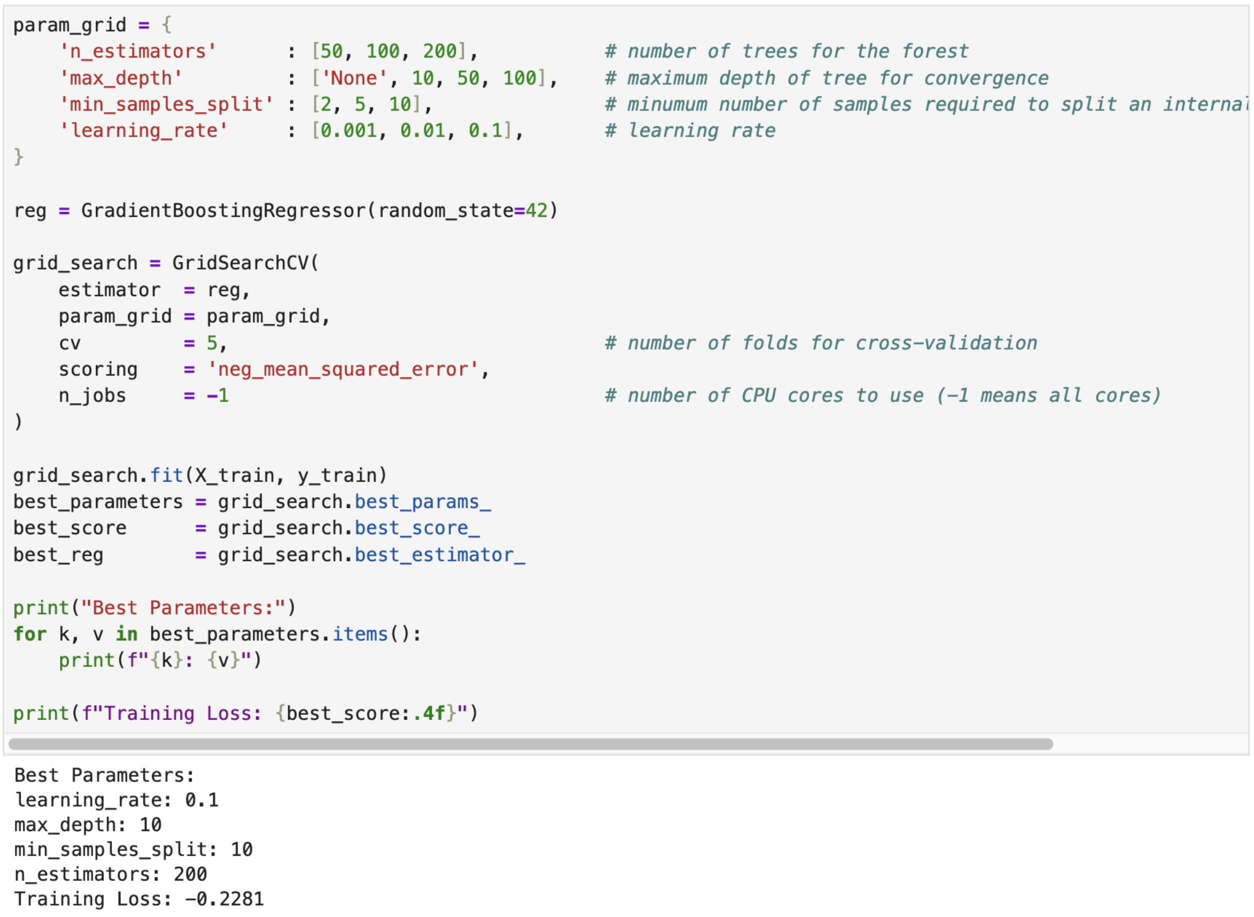


```
y_pred = best_clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 93.86
```

Regressor

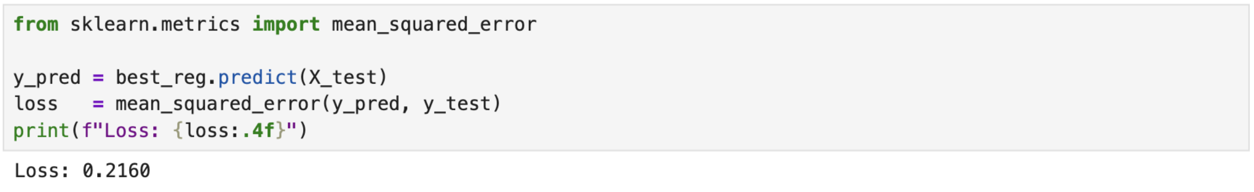
We developed a gradient boosting regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process of the regressor, where **Grid Search Cross-Validation** was employed to tune the optimal hyperparameters.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],          # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],    # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],              # minumum number of samples required to split an interna
    'learning_rate'     : [0.001, 0.01, 0.1],      # learning rate
}

reg = GradientBoostingRegressor(random_state=42)

grid_search = GridSearchCV(
    estimator  = reg,
    param_grid = param_grid,
    cv         = 5,                               # number of folds for cross-validation
    scoring    = 'neg_mean_squared_error',
    n_jobs     = -1                               # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_reg        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Loss: {best_score:.4f}")
```

```
Best Parameters:
learning_rate: 0.1
max_depth: 10
min_samples_split: 10
n_estimators: 200
Training Loss: -0.2281
```

After training the regressor, we evaluated its performance on the test dataset. The following screenshot displays the corresponding results. The complete implementation script is available on the GitHub page Gradient Boosting Regressor.



```
from sklearn.metrics import mean_squared_error

y_pred = best_reg.predict(X_test)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.2160
```

5.5.2 XGBoost

Extreme Gradient Boosting (XGBoost) is an enhanced variation of the traditional Gradient Boosting algorithm that provides significant improvements in speed, accuracy, and feature handling. These enhancements are achieved through advanced techniques such as regularization, parallel processing, and a more sophisticated loss function optimization using the second-order Taylor approximation instead of standard first-order gradient descent. Key characteristics of XGBoost include:

- XGBoost incorporates both L1-norm and L2-norm regularization techniques to prevent overfitting and improve model generalization.
- It includes built-in mechanisms for handling missing values efficiently, eliminating the need for external preprocessing.
- XGBoost inherently supports cross-validation, which helps in reducing overfitting and improving model robustness.
- It is computationally efficient and supports parallel processing via leveraging multi-core architectures for faster training and tuning.
- XGBoost employs a depth-first tree growth strategy and stops branching when further splits provide no performance gain. The model then performs backward pruning to optimize the final tree structure.
- It supports user-defined objective functions and evaluation metrics that enhances versatility and provides flexibility for a wide range of learning tasks.

Compared to traditional Gradient Boosting, XGBoost leverages a **Taylor series approximation** of the loss function to accelerate convergence. In XGBoost, the overall objective function is defined as

$$\mathcal{L}(\Phi) = \sum_i \mathcal{L}(y_i, \hat{y}_i) + \sum_k \Omega(f_k), \tag{5.11}$$

where the first term represents the residual loss w.r.t. the samples, aiming to minimize the bias, and the second term denotes the regularization component over all trees, aiming to control model complexity and reduce variance.

The residual loss at iteration t combines the predictions of all previous trees with the contribution from the current tree, such that

$$\mathcal{L}(y_i, \hat{y}_i) = \mathcal{L}(y_i, \hat{y}_{i(t-1)} + f_t(x_i)), \tag{5.12}$$

where $\hat{y}_{i(t-1)}$ represents the aggregated predictions from the previous $(t-1)$ trees, and $f_t(x_i)$ corresponds to the prediction of the current tree.

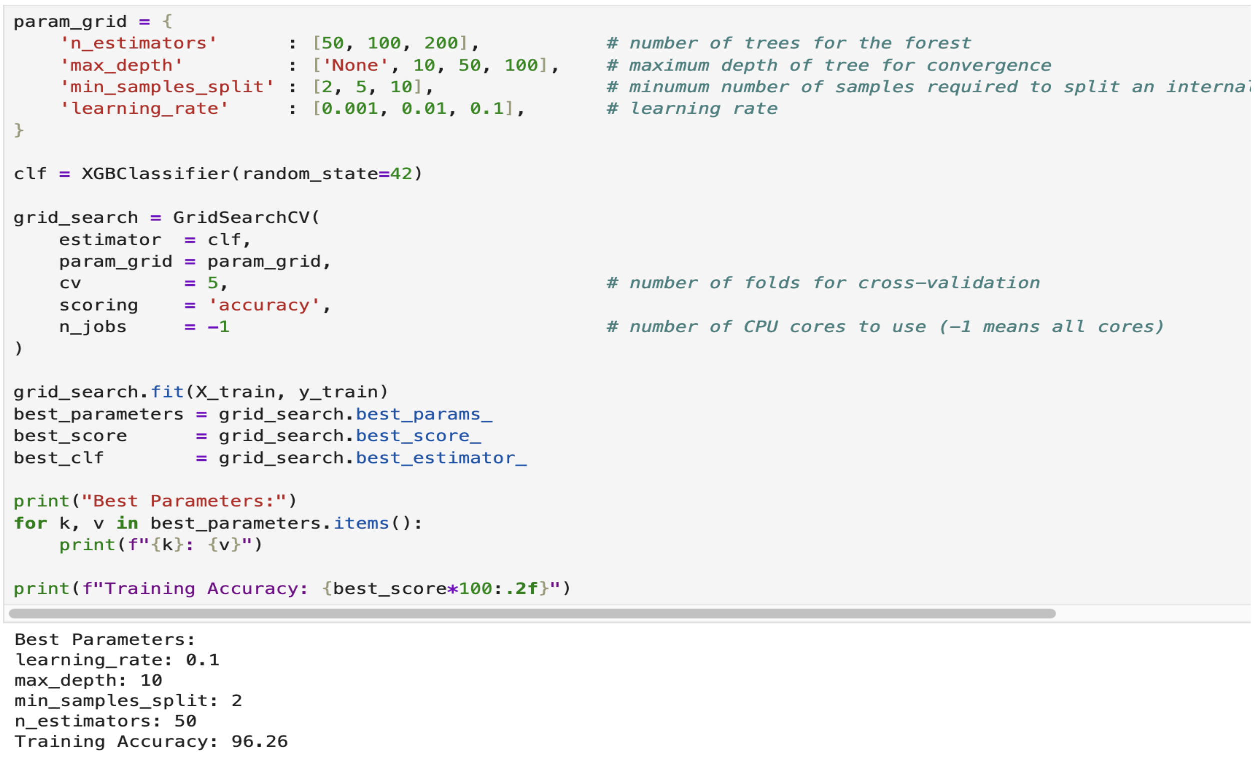
To efficiently optimize this objective, XGBoost applies a second-order Taylor series approximation to the loss function around $\hat{y}_{i(t-1)}$, that yields in

$$\mathcal{L}(y_i, \hat{y}_{i(t-1)} + f_t(x_i)) \approx \mathcal{L}(y_i, \hat{y}_i) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i), \tag{5.13}$$

where g_i and h_i denote the first- and second-order derivatives (gradient and Hessian) of the loss function w.r.t. the previous prediction, defined as $g_i = \partial_{\hat{y}_{i(t-1)}} \mathcal{L}(y_i, \hat{y}_{i(t-1)})$ and $h_i = \partial^2_{\hat{y}_{i(t-1)}} \mathcal{L}(y_i, \hat{y}_{i(t-1)})$.

Classifier

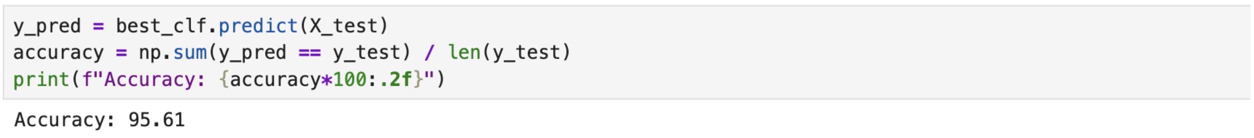
We developed a XGBoost classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process of the classifier. The model was imported from ***xgboost***. Moreover, we applied **Grid Search Cross-Validation** to identify the optimal hyperparameters of the model. In this process, we defined ranges for various parameters, such as *n_estimators* (no. of trees), *max_depth*, *min_samples_split*, and *learning_rate*.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],          # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],   # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],              # minumum number of samples required to split an interna
    'learning_rate'     : [0.001, 0.01, 0.1],      # learning rate
}

clf = XGBClassifier(random_state=42)

grid_search = GridSearchCV(
    estimator  = clf,
    param_grid = param_grid,
    cv         = 5,                                # number of folds for cross-validation
    scoring    = 'accuracy',
    n_jobs     = -1                                # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_clf        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Accuracy: {best_score*100:.2f}")
```

```
Best Parameters:
learning_rate: 0.1
max_depth: 10
min_samples_split: 2
n_estimators: 50
Training Accuracy: 96.26
```

After identifying the best classifier through Grid Search Cross-Validation, we evaluated its performance using the test dataset. The following screenshot presents the corresponding results. The complete implementation script is available on the GitHub page XGBoost Classifier.

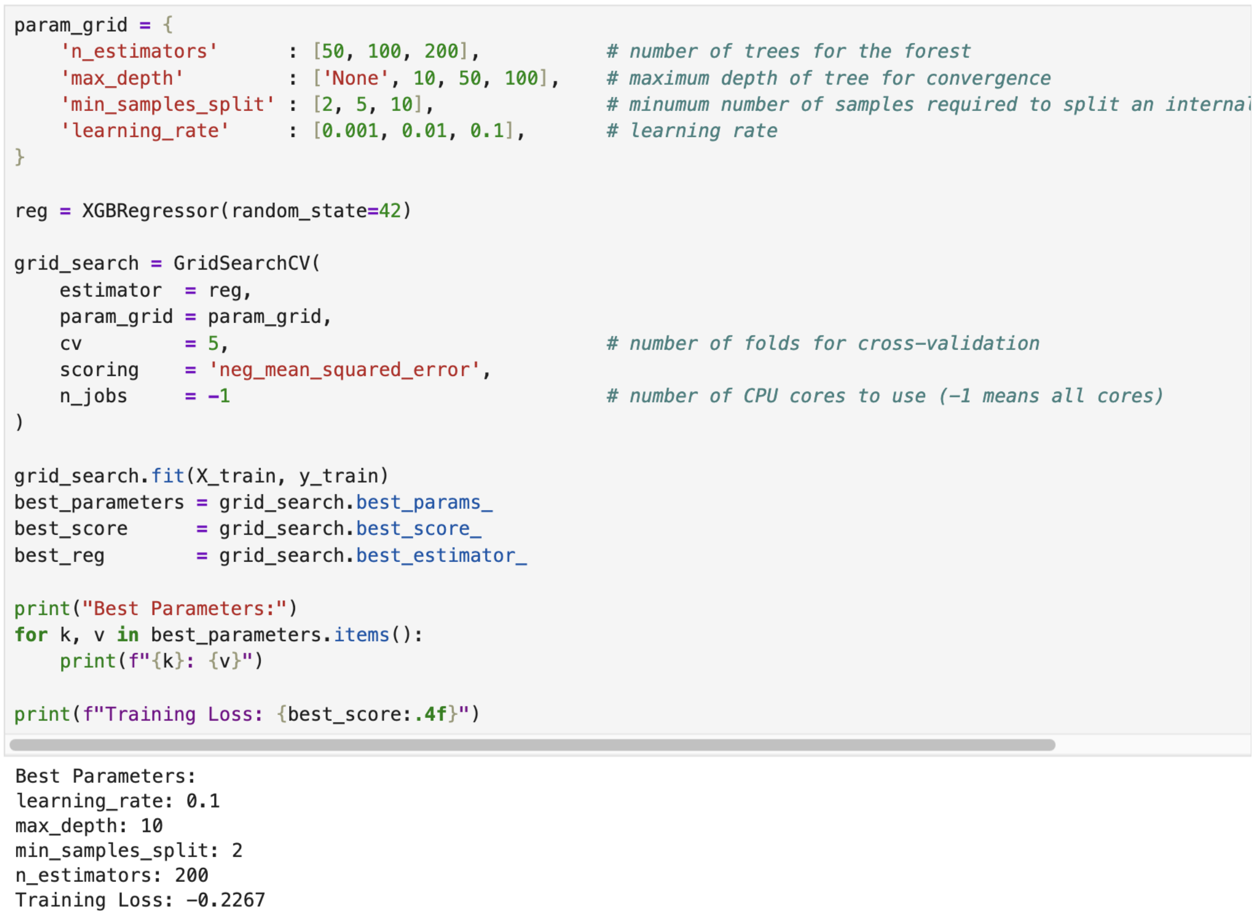


```
y_pred = best_clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)
print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 95.61
```

Regressor

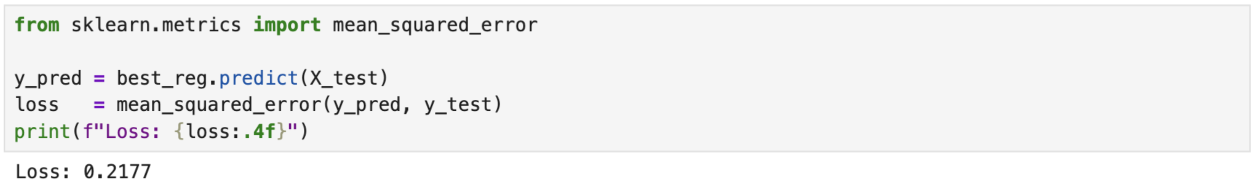
We developed an XGBoost regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process of the regressor, where **Grid Search Cross-Validation** was employed to tune the optimal hyperparameters.



```
param_grid = {
    'n_estimators'      : [50, 100, 200],          # number of trees for the forest
    'max_depth'         : ['None', 10, 50, 100],   # maximum depth of tree for convergence
    'min_samples_split' : [2, 5, 10],              # minumum number of samples required to split an internal
    'learning_rate'     : [0.001, 0.01, 0.1],      # learning rate
}

reg = XGBRegressor(random_state=42)

grid_search = GridSearchCV(
    estimator  = reg,
    param_grid = param_grid,
    cv         = 5,                                # number of folds for cross-validation
    scoring    = 'neg_mean_squared_error',
    n_jobs     = -1                                # number of CPU cores to use (-1 means all cores)
)

grid_search.fit(X_train, y_train)
best_parameters = grid_search.best_params_
best_score      = grid_search.best_score_
best_reg        = grid_search.best_estimator_

print("Best Parameters:")
for k, v in best_parameters.items():
    print(f"{k}: {v}")

print(f"Training Loss: {best_score:.4f}")
```

```
Best Parameters:
learning_rate: 0.1
max_depth: 10
min_samples_split: 2
n_estimators: 200
Training Loss: -0.2267
```

After training the regressor, we evaluated its performance on the test dataset. The following screenshot displays the corresponding results. The complete implementation script is available on the GitHub page XGBoost Regressor.



```
from sklearn.metrics import mean_squared_error

y_pred = best_reg.predict(X_test)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.2177
```

5.6 Support Vector Machine

Support Vector Machine (SVM) is a linear model designed to find a decision boundary (or hyperplane) that separates classes in the feature space. The optimal hyperplane is the one that maximizes the margin, i.e., the distance between the hyperplane and the nearest points from each class (also called support vectors), which typically leads to better generalization and classification performance. Figure 5.5 illustrates the SVM with linear kernel wherein the red line is the separator surrounded by two dashed lines as the support vectors.

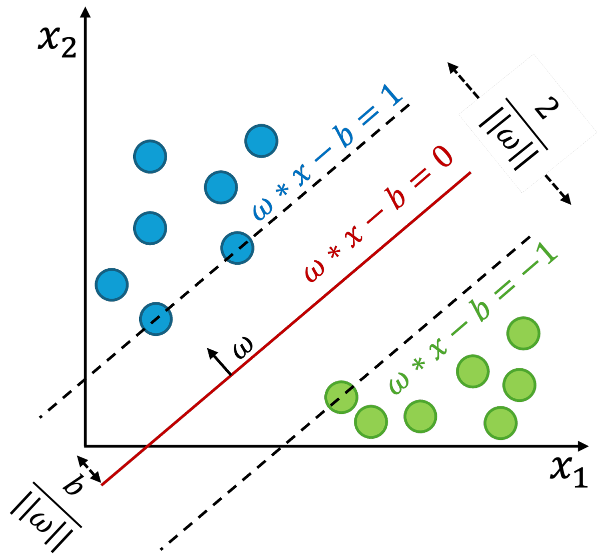


Figure 5.5: An illustration of SVM with linear kernel.

The linear hyperplane described above corresponds to the linear kernel in SVM, which produces a straight-line decision boundary suitable for linearly separable data. In addition to the linear kernel, SVM also supports non-linear kernels to capture more complex relationships. Two common non-linear kernels are the **Radial Basis Function (RBF)** and the **Polynomial** kernel. The RBF kernel generates a highly flexible, non-linear boundary capable of modeling intricate patterns, while the Polynomial kernel constructs non-linear boundaries using polynomial functions. However, the Polynomial kernel is more prone to overfitting when higher-degree polynomials are used.

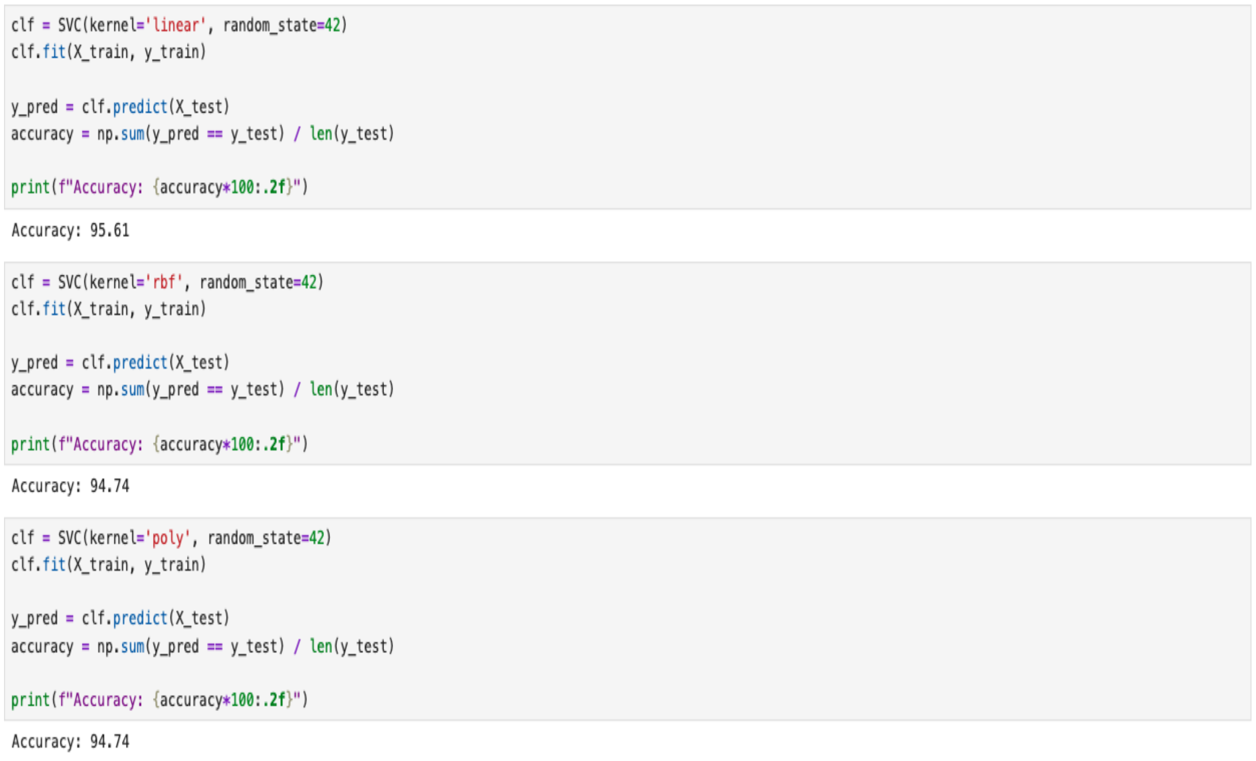
The key hyperparameter in SVM is the regularization parameter, denoted by C, which must be carefully tuned. The parameter C is a strictly positive value (commonly initialized as 1.0 by default) that controls the trade-off between maximizing the margin and minimizing classification error. Moreover, it is essential to normalize the input data in the preprocessing step for both training and testing. A commonly used normalization technique is z-score normalization, which standardizes features to have zero mean and unit variance.

5.6.1 Implementation

The machine learning SVM model in Python was developed from scratch following the guidelines provided in [26]. The complete implementation script is available on SVM from Scratch.

Classifier

We developed an SVM classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process and the evaluation results of the model. The model was imported from ***sklearn***.*svm*. The implementation includes all linear, rbf, and polynomial kernels. The complete implementation script is available on the GitHub page SVM Classifier.



```
clf = SVC(kernel='linear', random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 95.61
```

```
clf = SVC(kernel='rbf', random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy: {accuracy*100:.2f}")
```

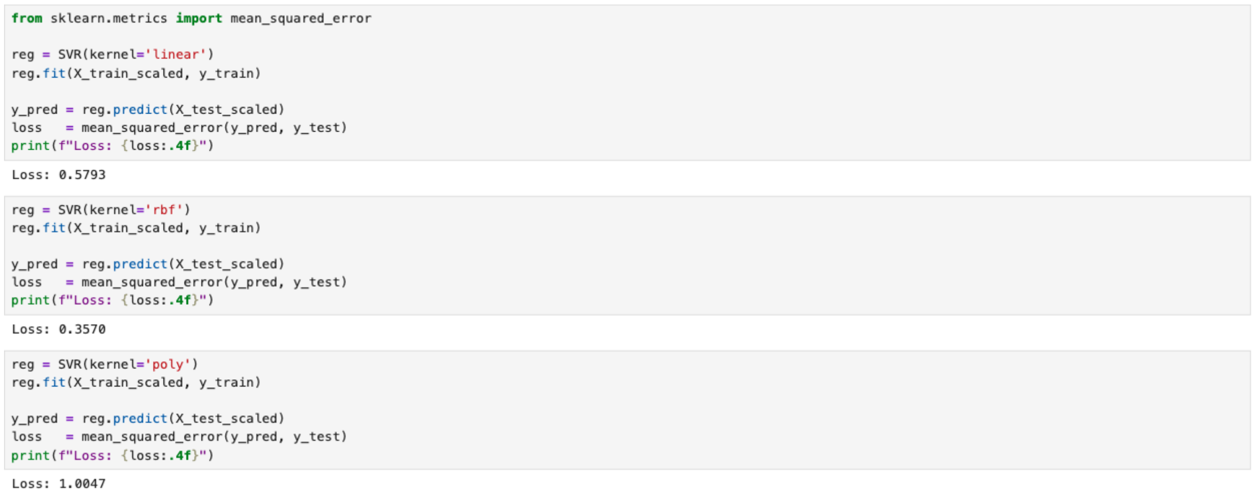
```
Accuracy: 94.74
```

```
clf = SVC(kernel='poly', random_state=42)
clf.fit(X_train, y_train)

y_pred = clf.predict(X_test)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 94.74
```

Regressor

We developed an SVM regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process the regressor, following by the evaluation results. The implementation includes all linear, rbf, and polynomial kernels. The complete implementation script is available on the GitHub page SVM Regressor.



```
from sklearn.metrics import mean_squared_error

reg = SVR(kernel='linear')
reg.fit(X_train_scaled, y_train)

y_pred = reg.predict(X_test_scaled)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.5793
```

```
reg = SVR(kernel='rbf')
reg.fit(X_train_scaled, y_train)

y_pred = reg.predict(X_test_scaled)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 0.3570
```

```
reg = SVR(kernel='poly')
reg.fit(X_train_scaled, y_train)

y_pred = reg.predict(X_test_scaled)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss: {loss:.4f}")
```

```
Loss: 1.0047
```

5.7 K-Nearest Neighbors

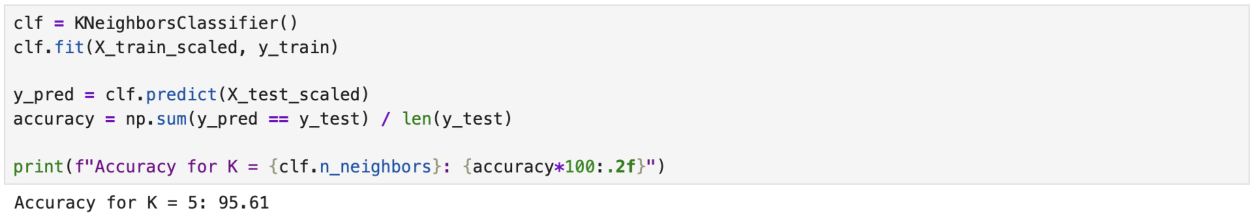
The K-Nearest Neighbors (KNN) algorithm is a lazy learning model, meaning it does not build an explicit training model but rather stores the entire dataset for prediction. For each data point in the test set, KNN computes the distance, commonly using Manhattan, Euclidean, or Minkowski metrics, between the query point and all points in the training set. It then selects the k nearest neighbors (typically $k = 5$) based on these distances. Finally, the model predicts the outcome by applying majority voting for classification tasks or averaging the neighbors' values for regression tasks.

5.7.1 Implementation

The machine learning KNN model in Python was developed from scratch following the guidelines provided in [27]. The complete implementation script is available on KNN from Scratch.

Classifier

We developed a KNN classifier for the **Breast Cancer** dataset using the built-in functions provided in *scikit-learn* in Python. The following screenshot illustrates the training process and the evaluation results of the model. The model was imported from ***sklearn***.*neighbors*. The complete implementation script is available on the GitHub page KNN Classifier.

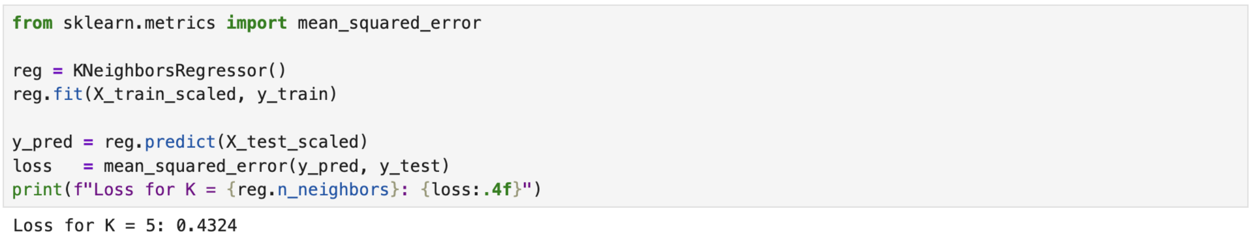


```
clf = KNeighborsClassifier()
clf.fit(X_train_scaled, y_train)

y_pred = clf.predict(X_test_scaled)
accuracy = np.sum(y_pred == y_test) / len(y_test)

print(f"Accuracy for K = {clf.n_neighbors}: {accuracy*100:.2f}")
```

```
Accuracy for K = 5: 95.61
```

Regressor

We developed a KNN regressor using the **California Housing** dataset available in scikit-learn. The following screenshot shows the training (fitting) process the regressor, following by the evaluation results. The complete implementation script is available on the GitHub page KNN Regressor.



```
from sklearn.metrics import mean_squared_error

reg = KNeighborsRegressor()
reg.fit(X_train_scaled, y_train)

y_pred = reg.predict(X_test_scaled)
loss   = mean_squared_error(y_pred, y_test)
print(f"Loss for K = {reg.n_neighbors}: {loss:.4f}")
```

```
Loss for K = 5: 0.4324
```

unsup

Unsupervised Learning

In machine learning, unsupervised data refers to unlabeled data, i.e., data without predefined output categories or target values. Unlike supervised learning, which involves classification or regression, unsupervised learning models such as **K-Means** and **DBSCAN** are used to cluster data based on their distances and/or distributions. These models are particularly useful for tasks like anomaly detection, where the goal is to identify data points that exhibit unusual patterns or behaviors. In the remainder of this chapter, we review K-Means and DBSCAN as two well-known unsupervised learning algorithms[1].

[1]**Note:** The main reference for the current section is the ***scikit-learn*** [20].

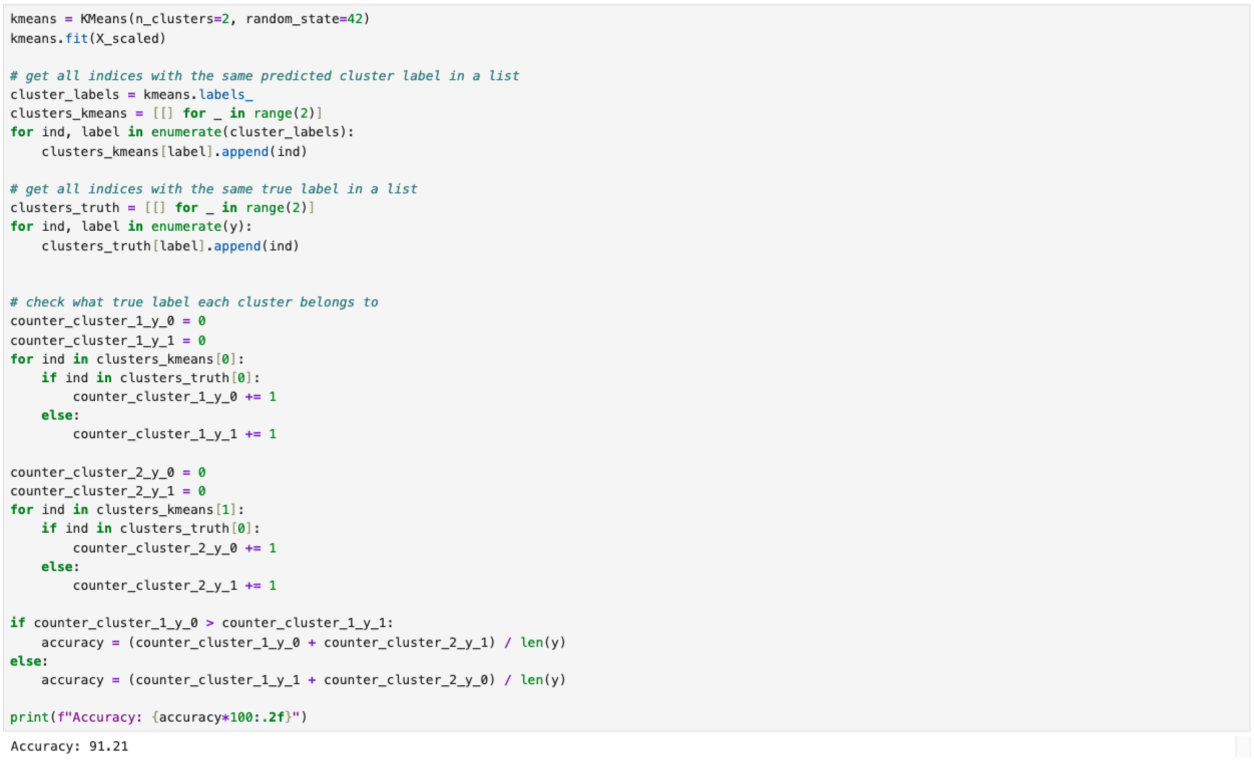
6.1 K-Means Clustering

K-Means is an unsupervised learning algorithm that partitions data into K distinct clusters. The algorithm begins by initializing K cluster centers, known as centroids. Each data sample is then assigned to the nearest centroid (commonly based on Euclidean distance) after which the centroids are updated as the mean of all samples within each cluster. This process repeats iteratively until the centroids no longer change significantly or the maximum number of iterations is reached.

6.1.1 Implementation

The K-Means clustering model in Python was developed from scratch following the guidelines provided in [28]. The complete implementation script is available on K-Means from Scratch.

We developed a K-Means clustering model for the breast cancer dataset using the built-in functions provided in *scikit-learn* in Python (we kept the target labels for evaluation purposes). The following screenshot illustrates the model's fitting and predicting process. The model was imported from ***sklearn***.*cluster*. The complete implementation script is available on the GitHub page K-Means Clustering.



```
kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(X_scaled)

# get all indices with the same predicted cluster label in a list
cluster_labels = kmeans.labels_
clusters_kmeans = [[] for _ in range(2)]
for ind, label in enumerate(cluster_labels):
    clusters_kmeans[label].append(ind)

# get all indices with the same true label in a list
clusters_truth = [[] for _ in range(2)]
for ind, label in enumerate(y):
    clusters_truth[label].append(ind)


# check what true label each cluster belongs to
counter_cluster_1_y_0 = 0
counter_cluster_1_y_1 = 0
for ind in clusters_kmeans[0]:
    if ind in clusters_truth[0]:
        counter_cluster_1_y_0 += 1
    else:
        counter_cluster_1_y_1 += 1

counter_cluster_2_y_0 = 0
counter_cluster_2_y_1 = 0
for ind in clusters_kmeans[1]:
    if ind in clusters_truth[0]:
        counter_cluster_2_y_0 += 1
    else:
        counter_cluster_2_y_1 += 1

if counter_cluster_1_y_0 > counter_cluster_1_y_1:
    accuracy = (counter_cluster_1_y_0 + counter_cluster_2_y_1) / len(y)
else:
    accuracy = (counter_cluster_1_y_1 + counter_cluster_2_y_0) / len(y)

print(f"Accuracy: {accuracy*100:.2f}")
```

```
Accuracy: 91.21
```

6.2 Density-Based Spatial Clustering of Applications with Noise

K-Means is an efficient clustering algorithm that performs well when the number of clusters, K, is appropriately specified. In contrast, Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a density-based algorithm that groups points according to their local density that enables the detection of clusters of arbitrary shapes and the identification of noise points.

To estimate density, DBSCAN relies on two key parameters: ε (or *eps*) and *minPts*. The former defines the maximum radius of a neighborhood around a point, while the latter specifies the minimum number of points required within this neighborhood for a point to be considered a core point. Based on these parameters, three types of points are defined:

- **Core Point:** A point that has at least *minPts* neighbors (including itself) within its *eps* neighborhood. Core points represent dense regions that form the backbone of clusters.
- **Border Point:** A point with fewer than *minPts* neighbors in its *eps* neighborhood, yet lies within the *eps* neighborhood of a core point.
- **Noise Point:** A point that is neither a core point nor a border point. Such points are considered outliers and do not belong to any cluster.

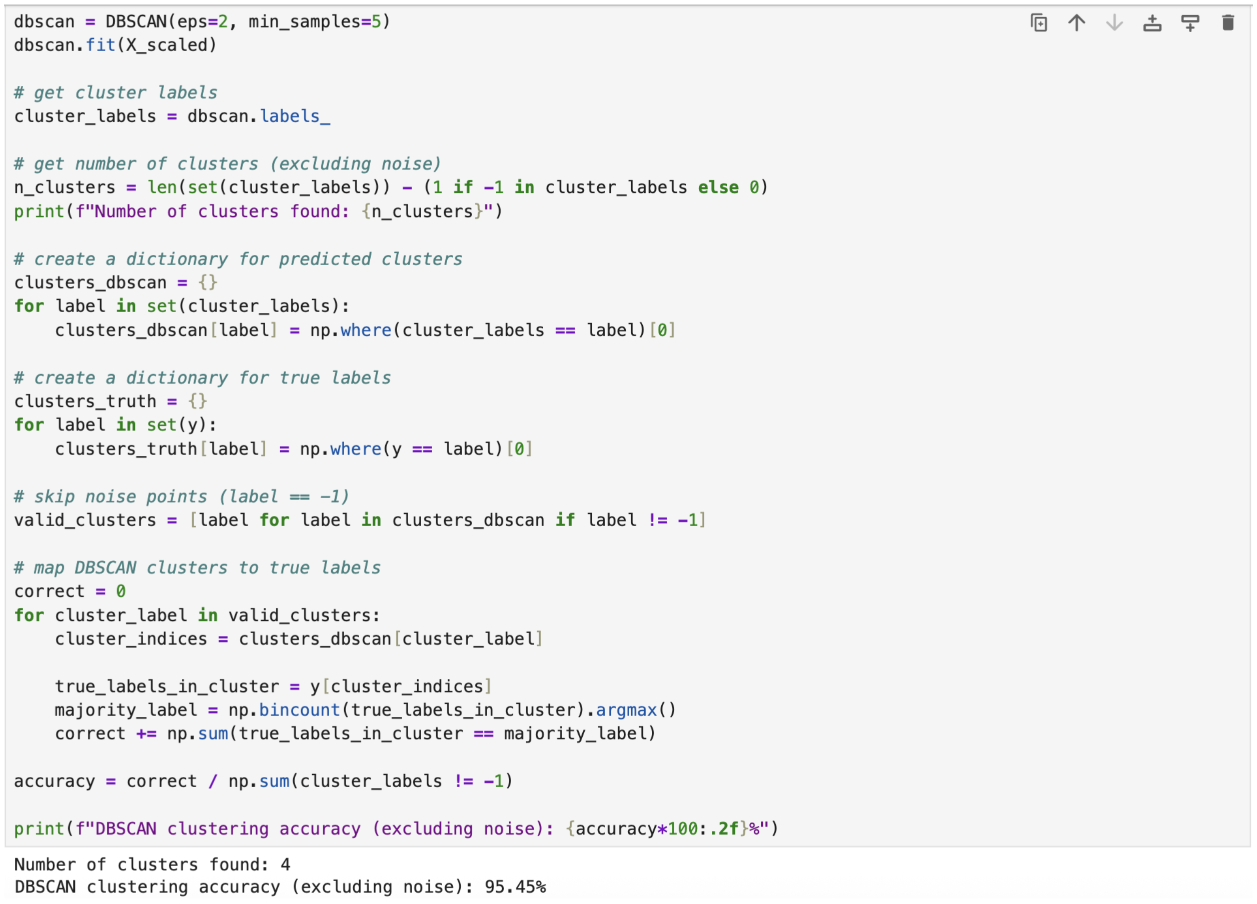
The algorithm begins with an unvisited data point, referred to as the *point of interest*. Then, It identifies all points within the *eps* distance of this point. If the number of neighboring points is less than *minPts*, the point is temporarily labeled as noise (it may later become a border point if it falls within the *eps* neighborhood of a core point). Otherwise, the point is identified as a core point, and a new cluster is initiated.

DBSCAN then recursively expands this cluster by adding all density-reachable points, i.e., all core points and their connected border points. This expansion continues by iterating through each newly added core point and including its neighbors if they also satisfy the core point condition or lie within the *eps* neighborhood of an existing core point. The process repeats until all points have been visited and assigned a label, either as part of a cluster or as noise.

6.2.1 Implementation

We developed a DBSCAN clustering model for the breast cancer dataset using the built-in functions provided in *scikit-learn* in Python (we kept the target labels for evaluation purposes). Although DBSCAN is a powerful clustering model, it is highly sensitive to the values of *eps* and *minPts*. By default, these parameters are set to $eps = 0.5$ and $minPts = 5$. However, the obtained results indicate that $eps = 4$ leads to better clustering performance.

The following screenshot illustrates the model's fitting and predicting process. The model was imported from ***sklearn***.*cluster*. The complete implementation script is available on the GitHub page DBSCAN Clustering.



```
dbscan = DBSCAN(eps=2, min_samples=5)
dbscan.fit(X_scaled)

# get cluster labels
cluster_labels = dbscan.labels_

# get number of clusters (excluding noise)
n_clusters = len(set(cluster_labels)) - (1 if -1 in cluster_labels else 0)
print(f"Number of clusters found: {n_clusters}")

# create a dictionary for predicted clusters
clusters_dbscan = {}
for label in set(cluster_labels):
    clusters_dbscan[label] = np.where(cluster_labels == label)[0]

# create a dictionary for true labels
clusters_truth = {}
for label in set(y):
    clusters_truth[label] = np.where(y == label)[0]

# skip noise points (label == -1)
valid_clusters = [label for label in clusters_dbscan if label != -1]

# map DBSCAN clusters to true labels
correct = 0
for cluster_label in valid_clusters:
    cluster_indices = clusters_dbscan[cluster_label]

    true_labels_in_cluster = y[cluster_indices]
    majority_label = np.bincount(true_labels_in_cluster).argmax()
    correct += np.sum(true_labels_in_cluster == majority_label)

accuracy = correct / np.sum(cluster_labels != -1)

print(f"DBSCAN clustering accuracy (excluding noise): {accuracy*100:.2f}%")
```

```
Number of clusters found: 4
DBSCAN clustering accuracy (excluding noise): 95.45%
```

Deep Learning

Some datasets contain highly complex patterns that cannot be effectively captured by the models discussed in Chapter 5. Deep Neural Networks (DNNs) are powerful and flexible models designed to handle such intricate data structures. In this chapter, we review three well-known types of DNNs, namely the Multi-Layer Perceptron (MLP) [20, 29, 30], Convolutional Neural Network (CNN) [31], and Recurrent Neural Network (RNN) [32].

7.1 Multi-Layer Perceptron Network

Multi-Layer Perceptron (MLP) networks are inspired by the biological neural connections in the brain. Each neuron receives inputs from other neurons, processes them, and transmits the output to the subsequent neurons [20, 29, 30].

In neural networks, a *perceptron* acts as a computational unit that receives multiple inputs, computes a weighted sum, and applies an activation function to introduce non-linearity. This non-linearity enables the network to learn non-linear relationships between input and output [29, 30].

7.1.1 Architecture

Figure 7.1 illustrates the overall architecture of an MLP, which typically includes one input layer, one or more hidden layers, and one output layer [29, 30].

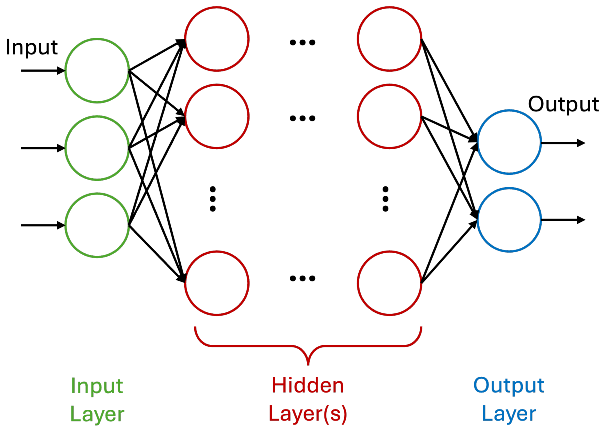


Figure 7.1: An overall architecture for an MLP.

- **Input Layer:** The number of neurons in the input layer corresponds to the number of features in the dataset.
- **Hidden Layer:** The network may include one or multiple hidden layers, each containing an arbitrary number of neurons. These layers are responsible for learning hierarchical feature representations.
- **Output Layer:** This layer generates the final prediction or output of the model.

Since neurons in an MLP are typically densely connected, most architectures are *fully connected*, i.e., each neuron in one layer is connected to all neurons in the subsequent layer [29, 30].

7.1.2 Mechanisms

Each MLP operates based on four fundamental mechanisms: **forward propagation**, **loss computation**, **backpropagation**, and **optimization** [29, 30].

Forward Propagation

During forward propagation, data flows from the input layer to the output layer, passing through the hidden layers. Each neuron processes the input in two steps [29, 30]:

1. **Weighted Sum:** Each neuron computes a weighted sum of its inputs, as shown in Eq. 7.1:

$$z = \sum_{i=1}^{N} w_i x_i + b, \tag{7.1}$$

where N is the number of inputs, x_i represents the input data, w_i the corresponding weights, and b the bias term. This operation establishes a linear relationship among inputs, weights, and bias [29, 30].

2. **Activation Function:** To introduce non-linearity, an activation function is applied to z. Common activation functions include [29, 30]:

   - **Sigmoid:**
   $$\sigma(z) = \frac{1}{1+e^{-z}}$$
   - **Rectified Linear Unit (ReLU):**
   $$f(z) = \max\{0, z\}$$
   - **Hyperbolic Tangent (Tanh):**
   $$\tanh(z) = \frac{2}{1+e^{-2z}} - 1$$

Loss Function

The loss function quantifies the difference between predicted and actual values, that guides the optimization process. Selecting an appropriate loss function is crucial for effective training. Common loss functions include [29, 30]:

- **Binary Classification:** Binary Cross-Entropy Loss (BCELoss) is suitable when the output layer contains a single neuron. For N samples, with true labels y_i and predicted probabilities $\hat{y}_i$, the loss is defined as [29, 30]:
$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$
- **Multi-class Classification:** For tasks with C classes, Cross-Entropy Loss (CELoss) is used. The output layer includes C neurons, each representing a class [29, 30]:
$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic})$$
- **Regression:** Mean Squared Error (MSE) Loss is commonly used for regression tasks [29, 30]:
$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Backpropagation

The objective of training is to minimize the loss by updating the network's parameters (weights and biases). Backpropagation is the process through which this minimization is achieved [29, 30].

1. **Gradient Calculation:** In the backpropagation, the mode first computes the gradient of the loss function w.r.t. each weight and bias [29, 30].
2. **Error Propagation:** In this step, the computed errors are propagated backward from the output layer to the input layer [29, 30].

3. **Parameter Update:** Finally, weights and biases are updated using the gradient descent rule with a defined learning rate α [29, 30]:

$$w \leftarrow w - \alpha \frac{\partial \mathcal{L}}{\partial w},$$
$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

Optimization

Optimization algorithms are applied to adjust the weights and biases efficiently based on the gradients. Two widely used optimization techniques in MLPs are **Stochastic Gradient Descent (SGD)** [33] and **Adaptive Moment Estimation (Adam)** [34].

- **Stochastic Gradient Descent (SGD):** In SGD, the model updates parameters using a single (or small batch of) training samples at each iteration, rather than the entire dataset. This approach improves computational efficiency and helps escape local minima. The parameter update rule is [33]:

  $$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta; x_i, y_i),$$

  where α is the learning rate, θ denotes model parameters, and $\nabla_\theta \mathcal{L}$ represents the gradient of the loss [33].

- **Adaptive Moment Estimation (Adam):** SGD is prone to trapping in local minima. To address this limitation, Adam combines the advantages of SGD with momentum and adaptive learning rates. It uses *momentum* to accelerate the gradient descent process by incorporating an exponentially weighted moving average of past gradients. This helps smooth out the trajectory of the optimization allowing the algorithm to converge faster by reducing oscillations. The update rule with momentum is given by [34]

  $$w_{t+1} = w_t - \alpha m_t,$$

  where m_t is the moving average of the gradients at time t; parameter α shows the learning rate; and w_t and w_{t+1} represent the weights of the model at times t and $t+1$, respectively. The momentum term m_t is updated recursively as [34]

  $$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial \mathcal{L}}{\partial w_t},$$

  where β_1 is the momentum parameter (typically set to 0.9), $\partial \mathcal{L} / \partial w_t$ is the gradient of the loss function w.r.t. the weights at time t [34].

  Adam also exploits Root Mean Square Propagation (RMSprop) that helps overcome the problem of diminishing learning rates via exponentially weighted moving average of squared gradients. The update rule for RMSprop is [34]

  $$w_{t+1} = w_t - \frac{\alpha_t}{\sqrt{v_t + \epsilon}} \frac{\partial \mathcal{L}}{\partial w_t},$$

  where ϵ is a small constant to avoid division by zero. Also, v_t is the exponentially weighted average of squared gradients that is given by [34]

  $$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial \mathcal{L}}{\partial w_t} \right)^2,$$

where β_2 is the decay rate (typically set to 0.999) [34].

Overall, Adam maintains exponentially decaying averages of past gradients (first moment) and squared gradients (second moment), allowing for stable and fast convergence. Its update rules are [34]:

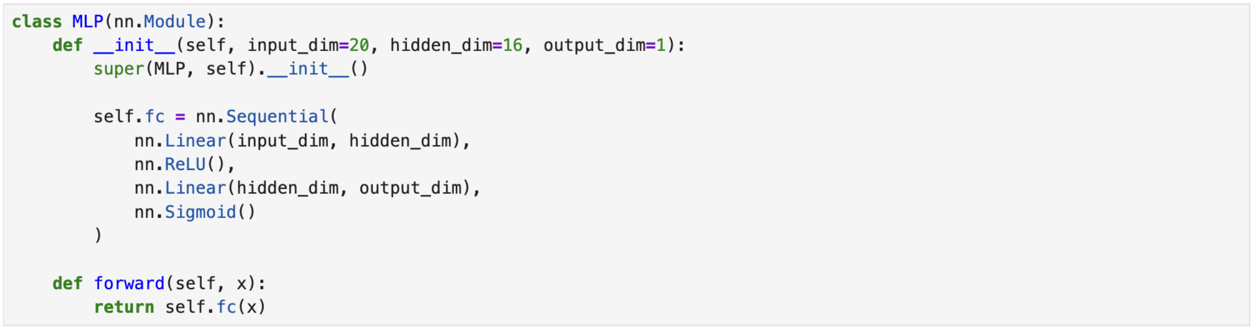
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon},$$

Adam generally outperforms vanilla SGD in practice due to its adaptive learning rate and momentum terms, especially in high-dimensional and sparse datasets [34].

7.1.3 Implementation

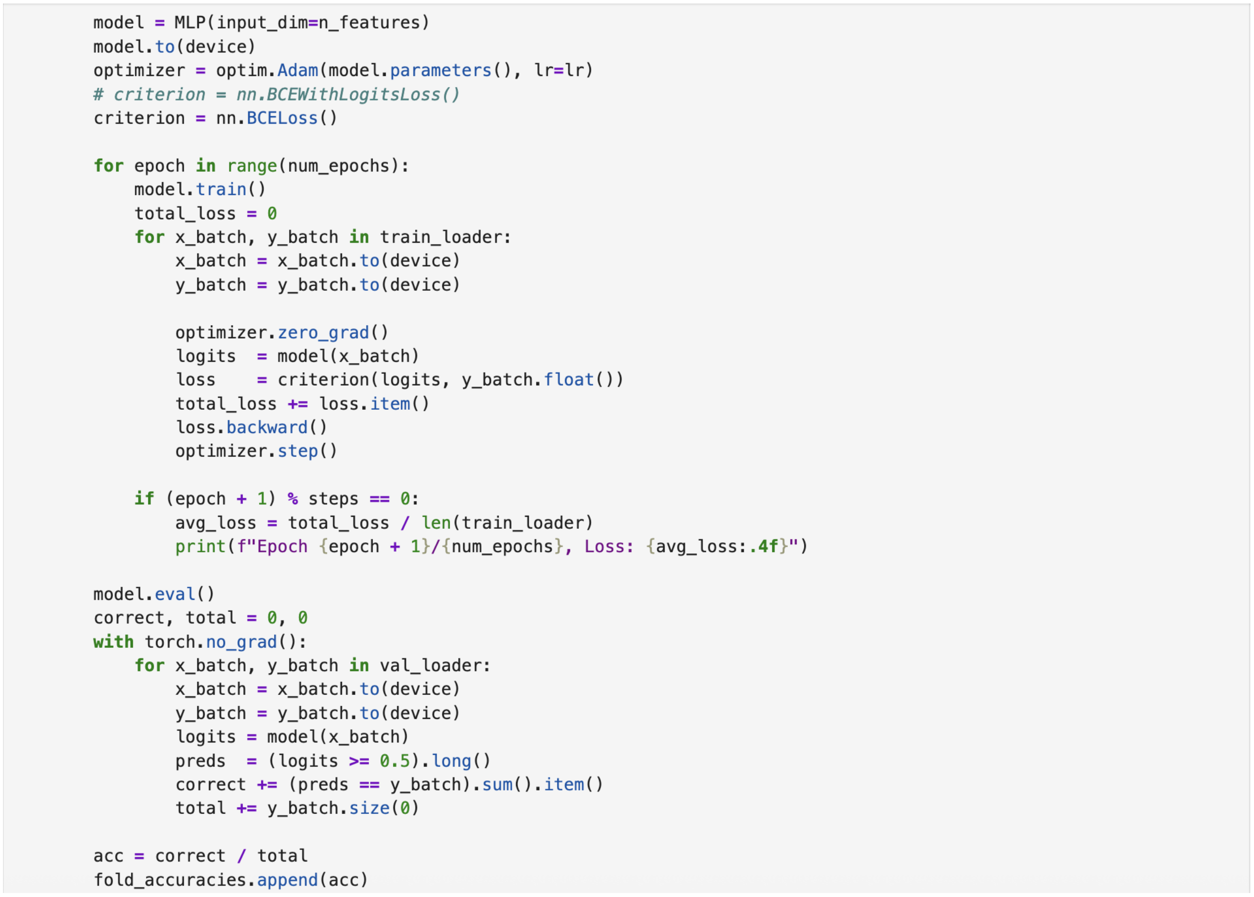
We developed two MLP networks for a binary classification task to demonstrate the difference between the **Binary Cross-Entropy Loss (BCELoss)** and the **Cross-Entropy Loss (CELoss)** functions. The models were implemented using the **PyTorch** [35] framework. Both MLPs consist of one input layer, one hidden layer, and one output layer. The input and hidden layers contain 20 (corresponding to the number of features) and 16 neurons, respectively. However, the output layers differ in the number of neurons depending on the selected loss function.

The first MLP uses BCELoss as its loss function. Therefore, the output layer consists of a single neuron with a **Sigmoid** activation function. The following screenshot illustrates the architecture of the corresponding MLP class.



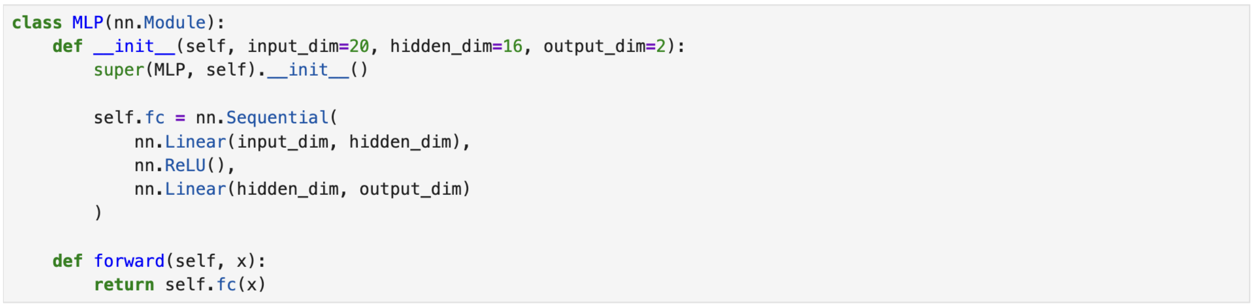
```python
class MLP(nn.Module):
    def __init__(self, input_dim=20, hidden_dim=16, output_dim=1):
        super(MLP, self).__init__()

        self.fc = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim),
            nn.Sigmoid()
        )

    def forward(self, x):
        return self.fc(x)
```

The training was conducted for 100 epochs, and **k-fold cross-validation** was applied to mitigate overfitting and improve model generalization. The following screenshot shows the training and validation procedures under k-fold cross-validation. The complete implementation script is available on the GitHub repository: BCELoss-based MLP.



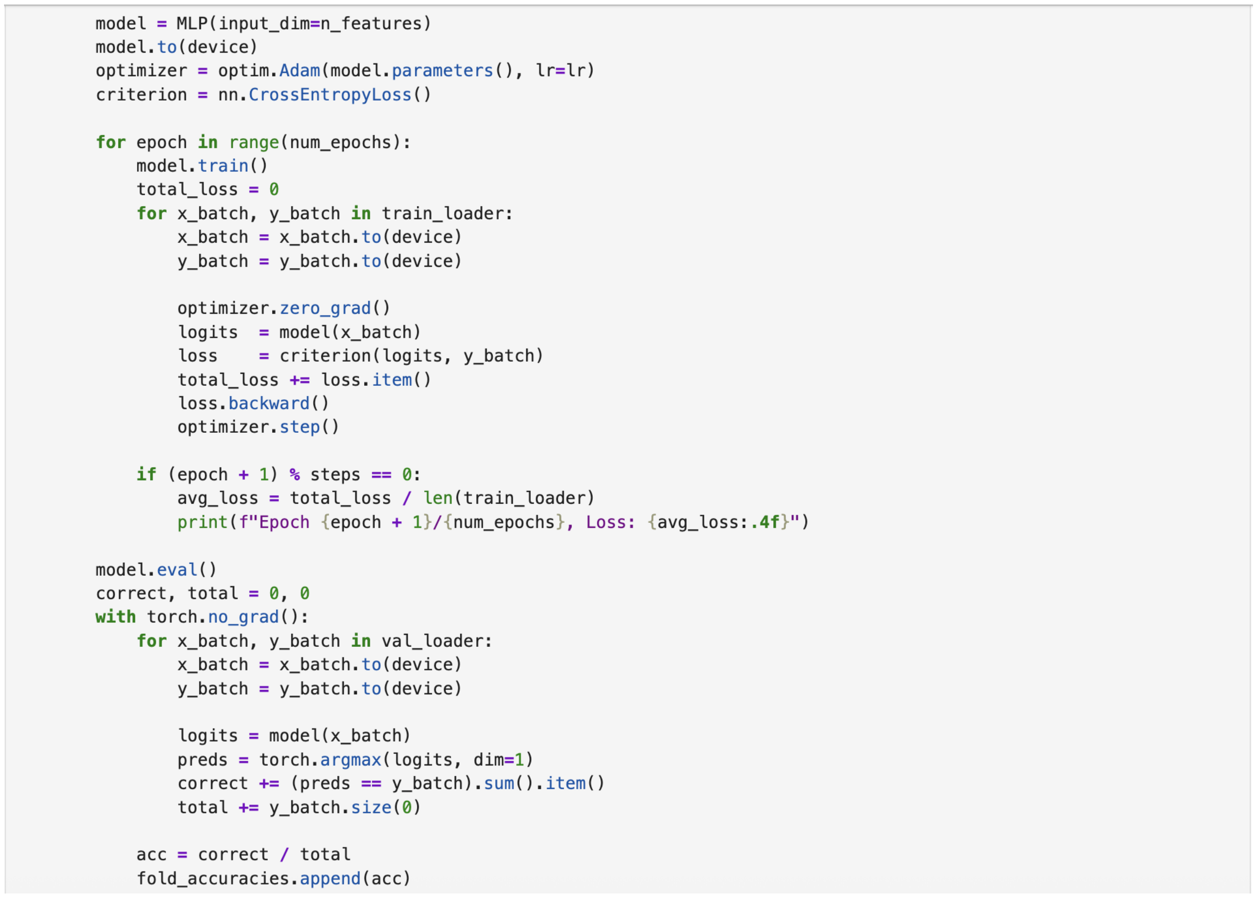
```python
        model = MLP(input_dim=n_features)
        model.to(device)
        optimizer = optim.Adam(model.parameters(), lr=lr)
        # criterion = nn.BCEWithLogitsLoss()
        criterion = nn.BCELoss()

        for epoch in range(num_epochs):
            model.train()
            total_loss = 0
            for x_batch, y_batch in train_loader:
                x_batch = x_batch.to(device)
                y_batch = y_batch.to(device)

                optimizer.zero_grad()
                logits  = model(x_batch)
                loss    = criterion(logits, y_batch.float())
                total_loss += loss.item()
                loss.backward()
                optimizer.step()

            if (epoch + 1) % steps == 0:
                avg_loss = total_loss / len(train_loader)
                print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}")

        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for x_batch, y_batch in val_loader:
                x_batch = x_batch.to(device)
                y_batch = y_batch.to(device)
                logits = model(x_batch)
                preds  = (logits >= 0.5).long()
                correct += (preds == y_batch).sum().item()
                total += y_batch.size(0)

        acc = correct / total
        fold_accuracies.append(acc)
```

The second model leverages CELoss as the loss function. In this case, the output layer contains two neurons corresponding to the two classes. The following screenshot shows the corresponding MLP class defined for this model.



```
class MLP(nn.Module):
    def __init__(self, input_dim=20, hidden_dim=16, output_dim=2):
        super(MLP, self).__init__()

        self.fc = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
        return self.fc(x)
```

Similar to the BCELoss-based MLP, this model was trained for 100 epochs using k-fold cross-validation. The following screenshot presents the training and validation process. As shown, to obtain the corresponding label, arg max function is applied to the logits. The complete implementation script is available on the GitHub repository: CELoss-based MLP.



```
        model = MLP(input_dim=n_features)
        model.to(device)
        optimizer = optim.Adam(model.parameters(), lr=lr)
        criterion = nn.CrossEntropyLoss()

        for epoch in range(num_epochs):
            model.train()
            total_loss = 0
            for x_batch, y_batch in train_loader:
                x_batch = x_batch.to(device)
                y_batch = y_batch.to(device)

                optimizer.zero_grad()
                logits  = model(x_batch)
                loss    = criterion(logits, y_batch)
                total_loss += loss.item()
                loss.backward()
                optimizer.step()

            if (epoch + 1) % steps == 0:
                avg_loss = total_loss / len(train_loader)
                print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {avg_loss:.4f}")

        model.eval()
        correct, total = 0, 0
        with torch.no_grad():
            for x_batch, y_batch in val_loader:
                x_batch = x_batch.to(device)
                y_batch = y_batch.to(device)

                logits = model(x_batch)
                preds = torch.argmax(logits, dim=1)
                correct += (preds == y_batch).sum().item()
                total += y_batch.size(0)

            acc = correct / total
            fold_accuracies.append(acc)
```

7.2 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are inspired by the human visual system. The eyes serve as input gates that capture visual information and transmit it to the brain, where the visual cortex processes the image step by step (layer by layer) to recognize the components of the scene.

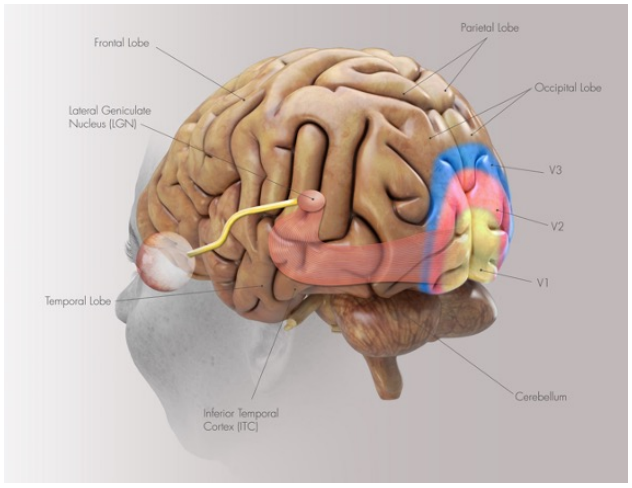


Figure 7.2: Visual cortex in the human brain [36].

Figure 7.2 shows the visual cortex in the human brain, including the **Retina**, **Lateral Geniculate Nucleus (LGN)**, **Primary Visual Cortex (V1)**, **Secondary Visual Cortex (V2)**, **Tertiary Visual Cortex (V3)**, and **Lateral Occipital Complex (LOC)** [36].

- **Retina:** A layer of photoreceptors at the back of the eye that detects light intensity and color (wavelength). The retina converts optical images into electrical signals and sends them via the optic nerve to the LGN [36].
- **Lateral Geniculate Nucleus (LGN):** Acts as a relay station between the retina and V1. It organizes, filters, and modulates visual signals while maintaining spatial mapping of the visual scene, a property known as retinotopy [36].
- **Primary Visual Cortex (V1):** Processes basic features such as edge orientation, contrast, and motion direction. Neurons in V1 have receptive fields that respond to stimuli in specific locations of the visual field [36].
- **Secondary Visual Cortex (V2):** Combines information from V1 to process more complex patterns, including contours, textures, and simple shapes. V2 serves as an intermediate stage that prepares visual information for higher-level processing [36].
- **Tertiary Visual Cortex (V3):** Divided into two parts [36]:
  - *V3 proper (dorsal V3):* Involved in motion and dynamic form processing. It is part of the dorsal stream ("where/how" pathway) that extends toward the parietal lobe, responsible for motion, depth, and spatial awareness [36].
  - *V3v (ventral V3):* Involved in object shape and form processing. It is part of the ventral stream ("what" pathway) that extends toward the inferotemporal cortex (IT) and is responsible for object recognition, color, and detailed form [36].
- **Lateral Occipital Complex (LOC):** Recognizes complete objects, responding to shapes and forms regardless of size, position, or viewpoint [36].

Inspired by this biological system, a CNN combines convolutional layers with a fully connected neural network. Figure 7.3 illustrates a typical CNN architecture including two convolutional layers followed by a fully connected MLP (see Sec. 7.1). The model is trained on the MNIST dataset and classifies each input image into one of ten classes (digits 0–9).

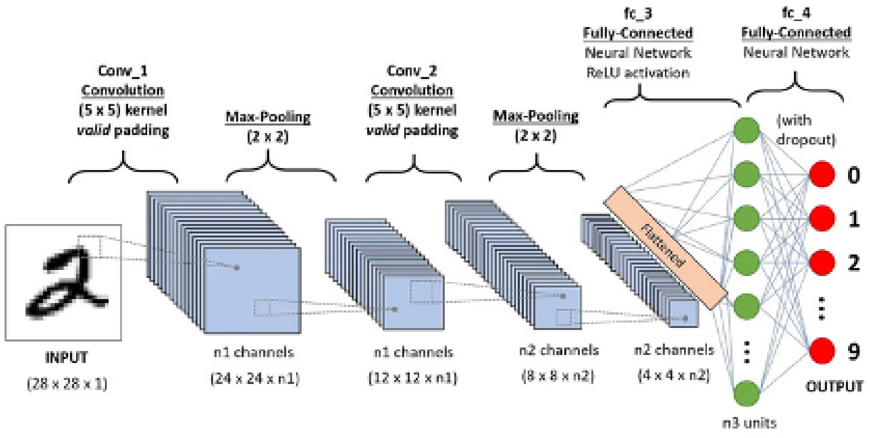


Figure 7.3: An example architecture of a CNN [31].

A convolutional layer in a CNN consists of number of channels (e.g., n_1 and n_2 in Fig. 7.3), each equipped with a fundamental building block called a *kernel* or *filter*. Filters in each layer learn to recognize patterns, progressively identifying more complex features in deeper layers. Figure 7.4 illustrates this hierarchical learning process.

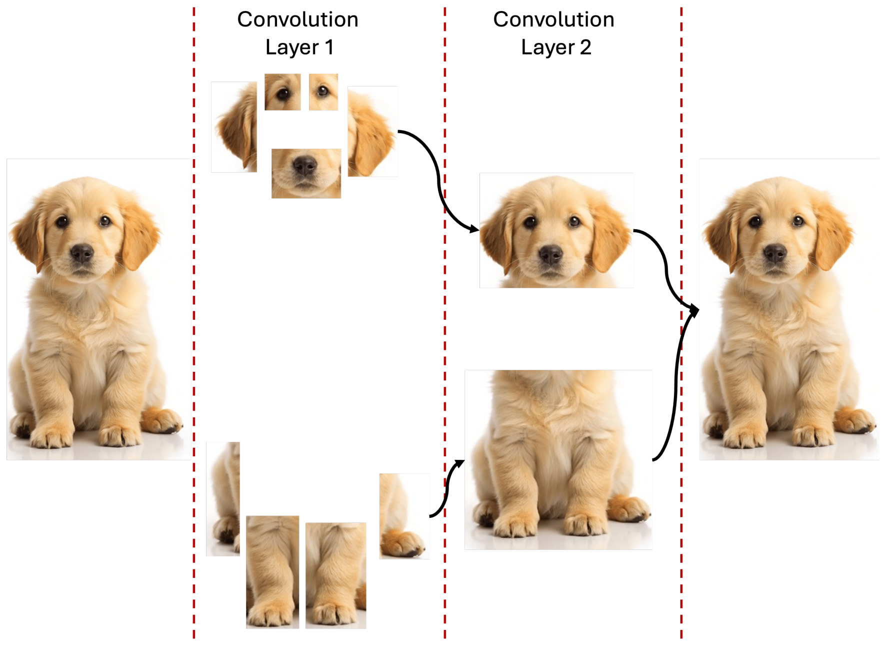


Figure 7.4: An example of convolutional layers in a CNN.

In this example, an initial image (a cute dog[1]) is filtered by kernels in the first convolutional layer. The upper filters learn to detect features such as eyes, nose, and ears, while the lower filters recognize paws. In

[1]Image source: Pinterest [1]

the second layer, the filters identify higher-level structures such as the head and body. This hierarchical learning enables CNNs to build complex visual understanding from simple features.
In the following subsections, we review the core components of a CNN, followed by implementing a CNN model.

7.2.1 Filter

A filter (or kernel) in a CNN detects local patterns in an image. Given an input image of size $(n \times n)$, a filter of size $(f \times f)$ (where $f < n$) slides over the image, performing element-wise multiplications and summations with corresponding pixel values. The result at each step forms the corresponding pixels in the output feature map [31]. Figure 7.5 shows an example of a (20×20) input image with a (5×5) filter.

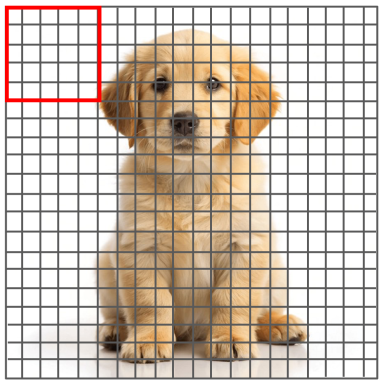


Figure 7.5: Filter operation in a CNN.

7.2.2 Stride

The *stride* determines how many pixels the filter moves each time it slides over the input image [31]. Figure 7.6 shows an example where the stride is $(5, 5)$, stating that the filter moves five pixels to the right after each operation and, upon reaching the end of a row, moves five pixels down to start the next pass.

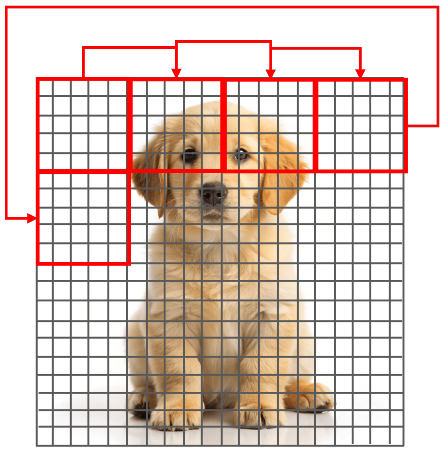


Figure 7.6: Stride in a CNN.

7.2.3 Padding

Padding controls the spatial dimensions of the output feature maps. Without padding, applying filters reduces the size of the feature maps. To preserve the input dimensions, zeros are added around the borders of the input image. Given an input of size $(n \times n)$, filter size $(f \times f)$, and stride s, the output dimension is calculated as [31]:

$$\text{output\_size} = \left(\frac{n-f}{s} + 1\right) \times \left(\frac{n-f}{s} + 1\right) \tag{7.2}$$

In our example (Fig. 7.5), the output size is (4×4). There are two common types of padding:

- **Valid:** No padding is applied; thus, $n' = n$, where n' is the new size of the input image [31].
- **Same:** Padding is applied such that the output size equals the input size. Let p be the padding size. Since padding happens for rows (up and down) and columns (left and right), the new input images size must follow $n' = n + 2p$. Therefore, we have [31]

$$\frac{n'-f}{s} + 1 = n \Rightarrow \frac{n+2p-f}{s} + 1 = n \Rightarrow p = \frac{n(s-1)+f-s}{2} \tag{7.3}$$

Figure 7.7 illustrates zero-padding (or Same) in an input image.

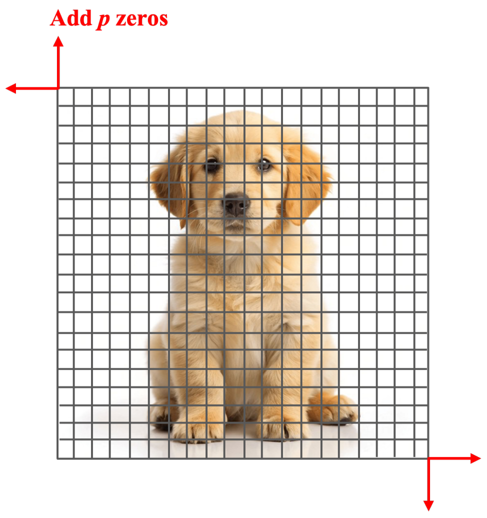


Figure 7.7: Padding in a CNN.

7.2.4 Pooling

Pooling is a downsampling technique used to reduce the spatial size of feature maps, which decreases computational cost and helps prevent overfitting by summarizing local features. The two most common pooling methods are *Max Pooling* and *Average Pooling*, which select the maximum or average value from each region, respectively [31]. Figure 7.8 shows an example using a (2×2) filter and a $(2, 2)$ stride.

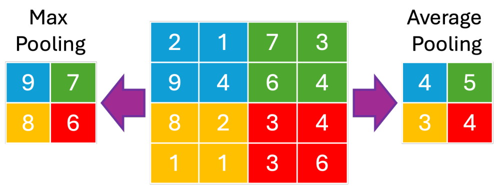


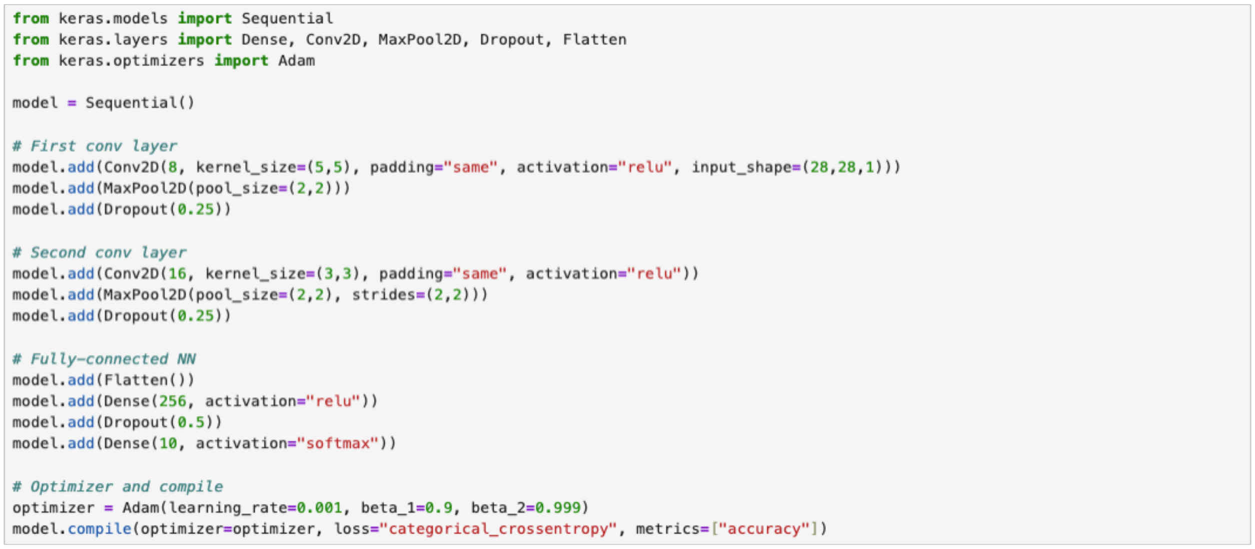
Figure 7.8: Pooling in a CNN. Left: Max Pooling; Right: Average Pooling.

7.2.5 Fully Connected Neural Network

The final stage of a CNN is a fully connected neural network (an MLP) that maps the extracted features to output predictions (e.g., class probabilities). The output of the convolutional layers is first flattened into a one-dimensional vector and then passed through one or more dense layers to produce the final outputs [31].

7.2.6 Implementation

The following screenshot illustrates the training process of a CNN model implemented in the TensorFlow [37] framework. The complete implementation script is available on the GitHub repository: CNN.



```python
from keras.models import Sequential
from keras.layers import Dense, Conv2D, MaxPool2D, Dropout, Flatten
from keras.optimizers import Adam

model = Sequential()

# First conv layer
model.add(Conv2D(8, kernel_size=(5,5), padding="same", activation="relu", input_shape=(28,28,1)))
model.add(MaxPool2D(pool_size=(2,2)))
model.add(Dropout(0.25))

# Second conv layer
model.add(Conv2D(16, kernel_size=(3,3), padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))
model.add(Dropout(0.25))

# Fully-connected NN
model.add(Flatten())
model.add(Dense(256, activation="relu"))
model.add(Dropout(0.5))
model.add(Dense(10, activation="softmax"))

# Optimizer and compile
optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
model.compile(optimizer=optimizer, loss="categorical_crossentropy", metrics=["accuracy"])
```

7.3 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are time-dependent architectures in which outputs from previous time steps participate in the prediction of the current output. Unlike MLPs and CNNs, which apply traditional backpropagation, RNNs employ **Backpropagation Through Time (BPTT)**, a variant designed to handle temporal dependencies. BPTT follows the same fundamental procedure as standard backpropagation, i.e., calculating errors from the output layer back to the input; yet, it propagates the *sum of errors over all time steps* that allows the network to learn from sequential dependencies. Figure 7.9 illustrates the conceptual difference between an MLP and an RNN [32].

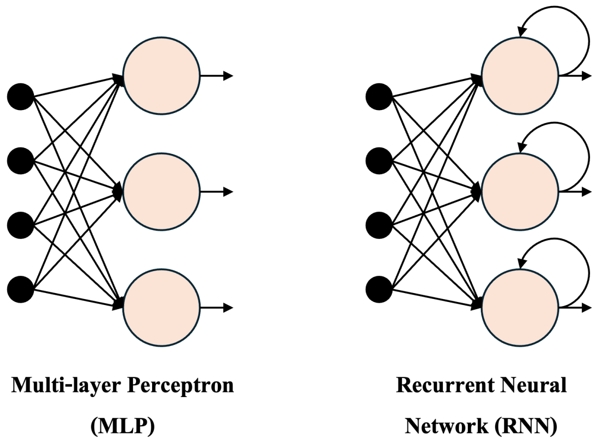


Figure 7.9: Comparison of an MLP and an RNN. Left: MLP; Right: RNN [38].

Figure 7.10 depicts the basic computational structure of an RNN unfolded across multiple time steps. At each time step t, the model takes as input the current feature vector x_t, the hidden state from the previous time step h_{t-1}, and produces both an updated hidden state h_t and an output o_t. The hidden state acts as the network's internal memory that carries temporal information across time. The hidden state at time step t is computed as [32]:

$$h_t = \tanh(Ux_t + Vh_{t-1} + b_h),$$

where U is the weight matrix associated with the input x_t, V shows the weight matrix associated with the previous hidden state h_{t-1}, and b_h represents bias term for the hidden layer [32].
Notabely, the RNN shares the same weights (U, W, and b_h) across all time steps, meaning that these parameters are time-invariant. This parameter sharing reduces model complexity and ensures consistent learning across temporal contexts [32].

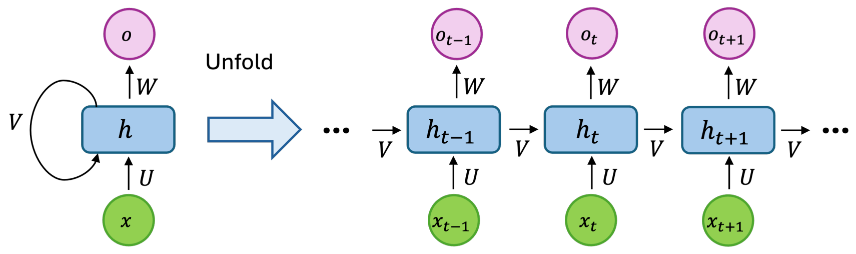


Figure 7.10: Building blocks of an RNN unfolded over time [32].

Once the hidden state h_t is computed, the output at time t is generated using an activation function, commonly the sigmoid function $\sigma(\cdot)$ [32]:

$$o_t = \sigma(W_o h_t + b_o),$$

where W and b_o denote the output-layer weights and bias, respectively. The term o_t represents the model's external output (used for tasks such as text generation, translation, or forecasting), while h_t serves as the internal output passed to the next time step $(t+1)$ [32].
The RNN architecture offers both advantages and disadvantages [32]:

- **Advantages:**
  - Can handle variable-length sequential inputs.
  - Maintains internal memory (hidden states) that captures temporal dependencies.
  - Shares parameters across time, reducing model size and complexity.
  - Well-suited for sequence-based tasks such as language modeling, translation, speech recognition, and text summarization.
- **Disadvantages:**
  - Prone to the *vanishing* and *exploding gradient* problems during training.
  - Struggles with long-term dependencies, making it less effective for tasks requiring long-range context.
  - Sequential computation limits parallelization, leading to slower training compared to feed-forward architectures.

7.3.1 Long Short-Term Memory

As mentioned earlier, RNNs are prone to the gradient vanishing problem. The Long Short-Term Memory (LSTM) architecture addresses this issue by introducing a long-term memory mechanism (see Fig. 7.11) [39, 40].

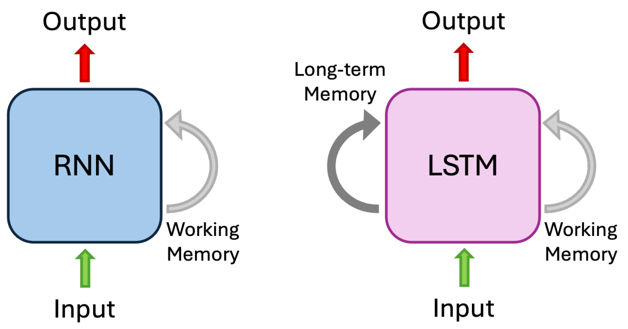


Figure 7.11: LSTM vs RNN. Left: RNN; Right: LSTM [41].

LSTMs maintain both short-term and long-term states across time steps. Figure 7.12 illustrates the architecture of an LSTM, which consists of three gates: *forget gate*, *input gate*, and *output gate* [39].

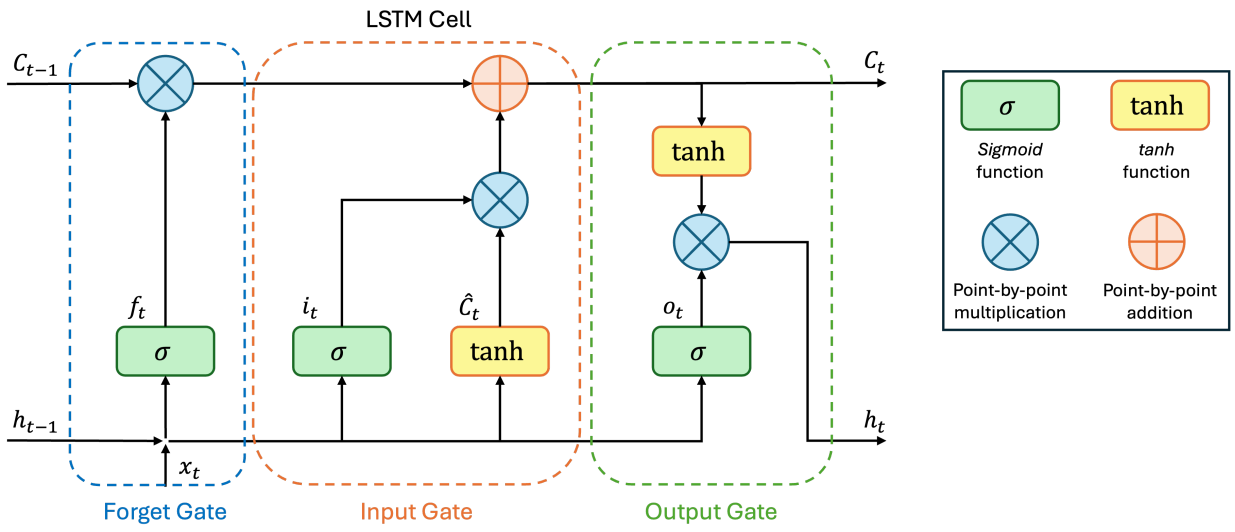


Figure 7.12: Building blocks of an LSTM [39].

The forget gate allows the model to selectively discard irrelevant information from the past. Specifically, the previous hidden state, h_{t-1}, and the current input, x_t, are processed to generate a forget vector, f_t, that controls how much of the previous cell state C_{t-1} should be retained [39]:

$$f_t = \sigma\Big(W_f \cdot [h_{t-1}, x_t] + b_f\Big),$$

where W_f and b_f represent the weights and bias of the forget gate, respectively. The updated cell state is then partially preserved as $C_{t-1} \odot f_t$ [39].

The input gate determines what new information should be added to the cell state. Both h_{t-1} and x_t are passed through a *sigmoid* layer and a tanh layer to generate candidate updates [39]:

$$i_t = \sigma\Big(W_i \cdot [h_{t-1}, x_t] + b_i\Big),$$
$$\hat{C}_t = \tanh\Big(W_c \cdot [h_{t-1}, x_t] + b_c\Big),$$

where i_t represents the input modulation gate, and $\hat{C}_t$ represents the candidate cell state. The new cell state C_t is then updated as follows [39]:

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t$$

Finally, the output gate controls what part of the cell state should be output at each time step. The gate first computes an activation o_t, then multiplies it with the tanh-transformed cell state to obtain the new hidden state h_t [39]:

$$o_t = \sigma\Big(W_o \cdot [h_{t-1}, x_t] + b_o\Big),$$
$$h_t = o_t \odot \tanh(C_t),$$

where W_o and b_o denote the weights and bias terms for the output gate. In summary, LSTMs effectively mitigate the vanishing gradient problem by maintaining a stable cell state, enabling the model to capture long-range temporal dependencies [39].

References

1. Pinterest, https://www.pinterest.com/.

2. A. Jonker and A. Gomstyn, “What are the key differences between structured and unstructured data?” https://www.ibm.com/think/topics/structured-vs-unstructured-data, International Business Machines (IBM), accessed: 2025.

3. GeeksforGeeks, “Ml | handling missing values,” https://www.geeksforgeeks.org/machine-learning/ml-handling-missing-values/, GeeksforGeeks, accessed: July 21, 2025.

4. Google, “Ml concepts - crash course,” https://developers.google.com/machine-learning/crash-course/prereqs-and-prework, Google, accessed: 2025.

5. Z. Wang *et al.*, “A comprehensive survey on data augmentation,” 2025. [Online]. Available: https://arxiv.org/abs/2405.09591

6. GeeksforGeeks, “Introduction to upsampling and downsampling imbalanced data in python,” https://www.geeksforgeeks.org/machine-learning/introduction-to-upsampling-and-downsampling-imbalanced-data-in-python/, GeeksforGeeks, accessed: July 23, 2025.

7. Jacob Murel, “What is upsampling?” https://www.ibm.com/think/topics/upsampling, International Business Machines (IBM), accessed: 2025.

8. ——, “What is downsampling?” https://www.ibm.com/think/topics/downsampling, International Business Machines (IBM), accessed: 2025.

9. Eryk Lewinson, “A comprehensive overview of regression evaluation metrics,” https://developer.nvidia.com/blog/a-comprehensive-overview-of-regression-evaluation-metrics/, Nvidia, accessed: April 20, 2023.

10. GeeksforGeeks, “Evaluation metrics for search and recommendation systems,” https://www.geeksforgeeks.org/machine-learning/clustering-metrics/, GeeksforGeeks, accessed: July 23, 2025.

11. Leonie Monigatti, “Clustering metrics in machine learning,” https://weaviate.io/blog/retrieval-evaluation-metrics, Weaviate, accessed: May 28, 2024.

12. GeeksforGeeks, “Bias and variance in machine learning,” https://www.geeksforgeeks.org/machine-learning/bias-vs-variance-in-machine-learning/, GeeksforGeeks, accessed: July 12, 2025.

13. ——, “Regularization in machine learning,” https://www.geeksforgeeks.org/machine-learning/regularization-in-machine-learning/, GeeksforGeeks, accessed: September 18, 2025.

14. ——, “Cross validation in machine learning,” https://www.geeksforgeeks.org/machine-learning/cross-validation-machine-learning/, GeeksforGeeks, accessed: July 23, 2025.

15. ——, “Vanishing and exploding gradients problems in deep learning,” https://www.geeksforgeeks.org/deep-learning/vanishing-and-exploding-gradients-problems-in-deep-learning/, GeeksforGeeks, accessed: July 23, 2025.

16. ——, “Kaiming initialization in deep learning,” https://www.geeksforgeeks.org/deep-learning/kaiming-initialization-in-deep-learning/, GeeksforGeeks, accessed: July 23, 2025.

17. ——, “Xavier initialization,” https://www.geeksforgeeks.org/deep-learning/xavier-initialization/, GeeksforGeeks, accessed: July 23, 2025.

18. ——, “What is batch normalization in deep learning?” https://www.geeksforgeeks.org/deep-learning/what-is-batch-normalization-in-deep-learning/, GeeksforGeeks, accessed: July 23, 2025.

19. ——, “What is layer normalization?” https://www.geeksforgeeks.org/deep-learning/what-is-layer-normalization/, GeeksforGeeks, accessed: July 23, 2025.

20. F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

21. Misra Turp, “How to implement logistic regression from scratch with python,” https://www.youtube.com/watch?v=YYEJ_GUguHw, AssemblyAI, accessed: Sep 14, 2022.

22. ——, “How to implement linear regression from scratch with python,” https://www.youtube.com/watch?v=ltXSoduiVwY, AssemblyAI, accessed: Sep 13, 2022.

23. ——, “How to implement decision trees from scratch with python,” https://www.youtube.com/watch?v=NxEHSAfFlK8, AssemblyAI, accessed: Sep 15, 2022.

24. ——, “How to implement random forest from scratch with python,” https://www.youtube.com/watch?v=kFwe2ZZU7yw, AssemblyAI, accessed: Sep 16, 2022.

25. ritvikmath, “Gradient boosting : Data science’s silver bullet,” https://www.youtube.com/watch?v=en2bmeB4QUo&t=608s, ritvikmath, accessed: Sep 29, 2021.

26. Patrick Loeber, “How to implement svm (support vector machine) from scratch with python,” https://www.youtube.com/watch?v=T9UcK-TxQGw, AssemblyAI, accessed: Sep 20, 2022.

27. Misra Turp, “How to implement knn from scratch with python,” https://www.youtube.com/watch?v=rTEtEy5o3X0, AssemblyAI, accessed: Sep 11, 2022.

28. Patrick Loeber, “How to implement k-means from scratch with python,” https://www.youtube.com/watch?v=6UF5Ysk_2gk, AssemblyAI, accessed: Sep 21, 2022.

29. GeeksforGeeks, “Multi-layer perceptron learning in tensorflow,” https://www.geeksforgeeks.org/deep-learning/multi-layer-perceptron-learning-in-tensorflow/, GeeksforGeeks, accessed: September 30, 2025.

30. Adrian Tam, “Building multilayer perceptron models in pytorch,” https://machinelearningmastery.com/building-multilayer-perceptron-models-in-pytorch/, Machine Learning Mastery, accessed: April 8, 2023.

31. Zoumana Keita, “An introduction to convolutional neural networks (cnns),” https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns, datacamp, accessed: November 14, 2023.

32. Cole Stryker, “What is a recurrent neural network (rnn)?” https://www.ibm.com/think/topics/recurrent-neural-networks, International Business Machines (IBM), accessed: 2025.

33. GeeksforGeeks, “Ml - stochastic gradient descent (sgd),” https://www.geeksforgeeks.org/machine-learning/ml-stochastic-gradient-descent-sgd/, GeeksforGeeks, accessed: September 30, 2025.

34. ——, “What is adam optimizer?” https://www.geeksforgeeks.org/deep-learning/adam-optimizer/, GeeksforGeeks, accessed: October 4, 2025.

35. Adam Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

36. Ophthalmology Training.com, “Visual cortex,” https://www.ophthalmologytraining.com/core-principles/ocular-anatomy/visual-pathway/visual-cortex, Ophthalmology Training.com, accessed: 2022.

37. Martín Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

38. Jeff Shepard, "How does a recurrent neural network (rnn) remember?" https://www.microcontrollertips.com/how-does-a-recurrent-neural-network-remember/, Microcontrollertips, accessed: May 8, 2024.

39. GeeksforGeeks, "What is lstm - long short term memory?" https://www.geeksforgeeks.org/machine-learning/regularization-in-machine-learning/, GeeksforGeeks, accessed: October 7, 2025.

40. Nvidia, "Long short-term memory (lstm)," https://developer.nvidia.com/discover/lstm, Nvidia, accessed: 2025.

41. Robail Yasrab *et al.*, "Phenomnet: Bridging phenotype-genotype gap: A cnn-lstm based automatic plant root anatomization system," *bioRxiv*, 2020.