Mihnea-Bogdan Jurca Ion Giosan

MACHINE LEARNING I LAB GUIDE

U.T.PRESS
Cluj-Napoca, 2025
ISBN 978-606-737-786-6

Mihnea-Bogdan JURCA Ion GIOSAN

MACHINE LEARNING I

Lab Guide

U.T.PRESS
Cluj - Napoca, 2025
ISBN 978-606-737-786-6

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Recenzia: Prof.dr.ing. Camelia Lemnaru
Conf.dr.ing. Raluca Brehar

Pregătire format electronic on-line: Gabriela Groza

ISBN 978-606-737-786-6

Preface

This book aims to serve as the laboratory guide for the first-year master's students in the Data Science program at the Technical University of Cluj-Napoca. It provides a gentle introduction to fundamental concepts in shallow machine learning. The chapters are structured in such a way that students can enjoy an interactive experience, alternating between theory and hands-on practice.

We would like to express our gratitude to the course by [?], which served as the primary inspiration for this work. Our hope is that this guide will help the new generation of AI engineers and scientists develop a solid understanding of the fundamental concepts while ensuring that the implementation details of the topics discussed are clearly explained in the accompanying notebooks.

Below is a table listing the links to the notebook resources:

Lab Number	**GitHub Repository URL**
Lab 1	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_1`
Lab 2	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_2`
Lab 3	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_3`
Lab 4	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_4`
Lab 5	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_5`
Lab 6	`https://github.com/mbjurca/UTCN_DS_ML_1/tree/main/Lab_6`

Table 1: List of Lab Notebooks and GitHub Repositories

Introduction to Python

Overview

This chapter introduces the Python programming language along with some important libraries like NumPy, Pandas, Matplotlib, Seaborn, and Scikit-learn, which are often used for Machine Learning applications. Several environments for developing Python-based applications are also discussed.

The main objective is to set up a development environment, get familiar with Python, and conduct some simple experiments with the aforementioned libraries.

Python Programming Language

Python is one of the most popular programming languages. Although it is a general-purpose language, it is used in various areas of applications such as Machine Learning, Artificial Intelligence, web development, IoT, and more. `https://www.tutorialspoint.com/python/index.htm`

Development Environments

There are several Integrated Development Environments (IDEs) that are popular for working with Python and Machine Learning due to their support for scientific computing, data analysis, and machine learning libraries:

- **Jupyter Notebook, PyCharm, Spyder, etc.** For data science and machine learning, Jupyter Notebook is highly recommended for exploration and prototyping, while PyCharm, Spyder or similar IDEs are better for more extensive development work.

- **Google Colab:** `https://colab.research.google.com/` (requires Google Drive account) - best for cloud computing with free GPU/TPU access and ease of use.

- **JupyterLab and Spyder / PyCharm Professional IDE via Anaconda:** `https://www.anaconda.com/products/individual` - recommended for offline work (with both Python notebooks and Python code) and complete control over the local environment.

Libraries

- **NumPy:** An open-source Python library consisting of multidimensional and single-dimensional array elements. NumPy is a standard library for numerical computations and is utilized in various domains including Pandas, SciPy, Matplotlib, and Scikit-learn.
  `https://numpy.org/doc/stable/user/absolute_beginners.html`

  `https://www.tutorialspoint.com/numpy/index.htm`
- **Pandas:** An open-source library providing high-performance data structures and data analysis tools for Python.
  `https://www.tutorialspoint.com/python_pandas/index.htm`
- **Matplotlib:** A library that offers various data visualization tools such as line plots, histograms, scatter plots, and more.
  `https://www.tutorialspoint.com/matplotlib/index.htm`
- **Seaborn:** A library for data visualization based on Matplotlib, providing high-level interfaces for drawing attractive and informative statistical graphics.
  `https://www.tutorialspoint.com/seaborn/index.htm`
- **Scikit-learn:** The most useful and robust library for machine learning in Python, providing tools for classification, regression, clustering, and dimensionality reduction.
  `https://www.tutorialspoint.com/scikit_learn/index.htm`

Note: We recommend checking out the resources mentioned above and visiting `https://cs231n.github.io/python-numpy-tutorial/` to run the Colab version of the tutorial. Many of the notions described here will be further used in this and subsequent works. Although you are probably familiar with many of the concepts, please make sure you understand concepts like defining numpy arrays of different types, reduction operations, and broadcasting after completing the readings.

Note: All labs code templates can be found here: `https://github.com/mbjurca/UTCN_DS_ML_1`

.

Implementation 1: Solve the notebook **introduction.ipynb** using the learned concepts from the tutorials.

A brief introduction of Linear Models

Overview

Before exploring different linear models, let's take a broader look at the general concepts of learning and linear approaches. Over the next four chapters, we will introduce various algorithms and methods for mapping data in a linear fashion.

The main objective is to develop a solid understanding of this broader family of linear methods, which will serve as a strong foundation for the practical work that follows.

A Brief Fundamental Recap

In 1998, Tom Mitchell defined machine learning as follows: a computer program is said to learn from experience E with respect to some class of **tasks** T and **performance** measure P, if its performance at tasks in T, as measured by P, improves with **experience** E.

Since we will focus only on the **supervised** and **unsupervised** taxonomies of machine learning, the experience typically refers to the amount of data provided to the model, and the performance is measured by evaluation metrics specific to a given task.

All algorithms and models presented in this work fall under the supervised learning taxonomy.

Consider a dataset represented as a set of pairs (x_i, y_i), where each x_i is an **input example** or **input features** and y_i is the corresponding target variable. The subscript i indexes each individual pair in the dataset. In supervised learning, the objective is to learn a function h, called the **hypothesis**, that approximates the true underlying function, by mapping the input space X to the target space Y. This means finding $h : X \to Y$ that best represents the relationship between the examples and their targets in the training data.

The term **training data/set** refers to the collection of examples used by models during the **learning** or **training** phase. The goal is for the model to **generalize** well(have good performance) to new, unseen data drawn from the same distribution, which is usually referred to as the **test data/set**. For further reading, [1] provides an excellent resource for understanding why learning from data is possible, which lies at the core of machine learning.

Supervised learning is so named because the **target variable** y_i that the hypothesis function predicts is provided in the training set. The goal is to use this information to optimize the parameters of the hypothesis function.

The linearity of the models discussed in this work refers to the fact that the hypothesis function is linear with respect to the learned parameters. As a general guideline, it is often advisable to begin by applying linear models before progressing to more complex, nonlinear approaches such as neural networks. In many practical applications, data may exhibit linear relationships, yielding effective results even with simpler models.

From an implementation perspective, it is usually not beneficial to use sets. We prefer using linear algebra-based representations such as matrices and vectors. The training set is usually represented as a matrix $\mathbf{X}$ with M rows (the number of examples in the dataset) and D columns (the number of features). There is also a vector $\mathbf{Y}$ with M rows, representing the target variable for each input example. The matrix $\mathbf{X}$ is also called the **design matrix**.

PLA - Perceptron Learning Algorithm

Overview

The Perceptron Learning Algorithm (PLA) is one of the most fundamental linear classification algorithms and is considered by some as a foundational building block for modern deep learning models. In simple terms, the perceptron algorithm classifies an input example x_i as either a positive or negative example (binary classification problem). Developed by Frank Rosenblatt in the 1960s, the perceptron sparked significant debate in the field. In response to Rosenblatt's achievement, Marvin Minsky and Seymour Papert quickly countered by demonstrating that, although the perceptron is an intuitive and powerful algorithm, its linear nature limits its ability to represent certain types of functions.

The main objective is to implement PLA for a simple binary classification problem, following all steps: generating and analyzing the data, training, testing, and visualizing the results.

Approach

Say we aim to classify whether an email is spam or not. At first glance, a seemingly simple solution would be to feed the entire email to an algorithm, which would then predict if the email is spam or not. However, this approach is not as straightforward due to the many variables an email contains, such as varying lengths, content, and formats. Moreover, processing an entire email in its raw format can be computationally expensive and may not scale well with a large volume of emails.

To make the inference process more efficient, it is crucial to extract only the most meaningful information, thereby reducing the amount of data per email. This allows the algorithm to focus only on the relevant aspects of the email with respect to the task at hand. Each piece of information that is passed to the model is referred to as a feature or property of the input example. Depending on the nature of the data, these features can be transformed and represented in different ways for the final mathematical representation.

Question 1: What properties can be extracted from an email to perform spam detection? Provide three examples and explain why each might indicate whether an email is spam or not.

Mathematical data representation

Let X, Y compose the training set, where $(x_0, y_0), (x_1, y_1), \ldots$ represent the individual training instances. Each pair (x_i, y_i), x_i denotes the input example, and y_i is the label attributed to it. Returning to the email classification problem, we can imagine X as the set of all emails we want to classify in order to train the model, with each x_i representing an individual email.

Each x_i is a vector of dimension D, containing D features of the corresponding email. Each dimension of the vector represents a specific feature, for instance, one feature might be the number of words in the email's subject line. By convention, we refer to the d-th feature of the i-th training example as $x_i^{(d)}$. At this stage, we will not focus on how these features are transformed or represented in the final feature vector. The main idea is that each x_i encapsulates all the meaningful modeled features for each email, and this is referred to as the feature vector.

For this first model, we will generate our own data and assign fictional meaning to the features for didactic purposes.

Implementation 1: Implement section 1 of the **perceptron.ipynb** file where you will have to generate and analyze the data.

Learning Algorithm

To choose a hypothesis function $h(x)$ that will decide whether an email is classified as spam or not, we can intuitively assign a weight to each feature of the email. Each feature of an email $x_i^{(d)}$ will be associated with a learned weight parameter $w^{(d)}$. The summation of the products between each feature and its corresponding weight parameter gives us a scalar value, which is then compared to a threshold.

The email is classified as spam if $\sum_{d=1}^{D} w^{(d)} x_i^{(d)} >$ threshold, and not spam if $\sum_{d=1}^{D} w^{(d)} x_i^{(d)} <$ threshold.

This formula can be written more compactly as

$$h(\mathbf{x_i}) = \text{sign}\left(\left(\sum_{d=1}^{D} w^{(d)} x_i^{(d)}\right) + b\right),$$

In order to facilitate the implementation, we will rewrite this in vector form. To do so, we will use the bias trick, meaning that the term b will be encapsulated in the parameter vector w. Now, the problem is that $w \in D+1$ and $x_i \in D$ in order to write it in vector form, x_i and w must have the same dimension so that we can compute the dot product. In order to do so an extra dimension $x_i^0 = 1$ will be added. Mathematically, this can be rewritten as follows:

$$\mathbf{X} = \{1\} \times \mathbb{R}^d = \{[x^{(0)}, x^{(1)}, \cdots, x^{(d)}]^T \mid x^{(0)} = 1, x^{(1)} \in \mathbb{R}, \cdots, x^{(d)} \in \mathbb{R}\}$$

With this convention, $\mathbf{w}^T \mathbf{x_i} = \sum_{d=0}^{D} w^{(d)} x_i^{(d)}$, and the hypothesis becomes:

$$h(\mathbf{x_i}) = \text{sign}(\mathbf{w}^T \mathbf{x_i}).$$

To learn the w parameters, the Perceptron Learning Algorithm (PLA) is an iterative process that updates the weight vector based on each misclassified example. Geometrically, it can be seen as adjusting the decision boundary to better separate the two classes.

For each misclassified x_i in X:

$$w := w + y_i x_i$$

Implementation 2: Implement sections 2 and 3 of the perceptron.ipynb file to train, test, and visualize the perceptron algorithm.

Question 2: The perceptron is guaranteed to converge (i.e., find a solution that perfectly classifies both categories) in certain cases. What are these cases, and why does the algorithm converge?

Linear Regression

Overview

In the previous chapter, we have described the Perceptron Learning Algorithm (PLA), which is designed for binary classification. Now, we will explore a different model—Linear Regression. Instead of categorizing data points into different classes, our goal is to predict a continuous variable $y_i \in \mathbb{R}$.

We aim to define a hypothesis function that will map any data point from the input space $\mathbf{x}_i \in \mathbb{R}^d$ to the target space $Y \in \mathbb{R}$. In this section, we will analyze the Linear Regression model and optimize it using two different methods based on **LMS** (Least Mean Squares). One solution will be an iterative one also known as **gradient descent** and the other an **analytical solution** where we can compute the solution directly.

The main objective is to implement linear regression for a simple regression problem, following all steps: loading and preprocessing the data, implementing both gradient-based and closed-form solutions, defining the prediction and loss functions, optimizing, training, and visualizing the obtained weights.

Approach

For the linear regression model, we will work with the California housing dataset [2]. This dataset contains information about houses in California and their values, based on various features such as the number of rooms, number of bedrooms, or proximity to the ocean. The task is to accurately predict the value of a house given these features.

A common issue with real-world data is that different features often have varying ranges of values. This will also be explored in Section 1 of the practical work on linear regression. To ensure that each feature contributes equally to the model, a normalization step is usually required. This step helps making the training process more stable and improves convergence.

Implementation 1: Implement Section 1 from the **linear_regression.ipynb** file in order to load and preprocess the dataset.

Let's draw a comparison with the previously discussed PLA algorithm and see how we can adapt it for this task. As before, multiple features are provided, and each feature may have a different impact on the final value of the house. For instance, the number of bedrooms is likely to be directly proportional to the value of the house, while the age of the building could be a depreciating factor, the newer the building, the more amenities it will likely offer. This reasoning leads us to a similar idea as before: we assign each feature a weight $w^{(d)}$, forming a weighted sum to compute a score. This is exactly what we will do. Furthermore, we no longer need the sign function, as the output is now a continuous value that we aim to predict directly.

The only remaining question is: how do we optimize the weights of the model?

Before delving into the optimization, let's summarize the parts we already know. Our chosen hypothesis function is:

$$h(\mathbf{x}_i) = \left(\sum_{d=1}^{D} w^{(d)} x_i^{(d)} \right) + b,$$

In vector form, using the bias trick (where the bias term is incorporated into the weight vector), this can be written as:

$$h(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i,$$

where $\mathbf{w} \in \mathbb{R}^{D+1}$ is the weight vector and $\mathbf{x}_i \in \mathbb{R}^{D+1}$ is the input vector augmented with a bias term.

Gradient descent optimization

The first optimization algorithm starts with the observation that, to achieve good performance, the predicted values by the hypothesis $h(\mathbf{x}_i)$ should be as close as possible to the true labels y_i, for all the m examples. To compare these two values, we define a cost function L as follows:

$$L(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{m} \left(h\left(\mathbf{x}_i\right) - y_i\right)^2$$

The goal is to choose **w** such that we minimize this cost function, which is also known as the loss function.

We begin with a random set of weights w and iteratively take the partial derivatives of the loss function with respect to the weights, so that we **step** in the direction of the minimum. This iterative optimization process is known as gradient descent. We continue this process until we converge to the minimum or a value very close to it. Gradient descent can be written as follows:

$$w^{(d)} := w^{(d)} - \alpha \frac{\partial}{\partial w^{(d)}} L(\mathbf{w}),$$

where the formula is written with respect to one dimension of the weight vector $\mathbf{w} \in \mathbb{R}^D + 1$.

Here, α is a fixed parameter, such as 0.001, called the **learning rate**. These kinds of parameters, which are set initially and not learned through the optimization process, are referred to as **hyperparameters**.

An interesting property of this particular loss function is that it is convex, meaning it has only one local minimum, which coincides with the global minimum.

Question 1: What happens if we set α to a high value? What happens if we set α to be too small? Why did we say that gradient descent will converge to the minimum of the function or a value close to the minimum?

Now, let's compute the partial derivatives:

$$\begin{aligned}
\frac{\partial}{\partial w^{(d)}} L(\mathbf{w}) &= \frac{\partial}{\partial w^{(d)}} \frac{1}{2} \left(h_w(\mathbf{x}) - y\right)^2 \\
&= \left(h_w(\mathbf{x}) - y\right) \cdot \frac{\partial}{\partial w^{(d)}} \left(h_w(\mathbf{x}) - y\right) \\
&= \left(h_w(\mathbf{x}) - y\right) \cdot \frac{\partial}{\partial w^{(d)}} \left(\sum_{j=0}^{D} w^{(j)} x^{(j)} - y \right) \\
&= \left(h_w(\mathbf{x}) - y\right) x^{(d)}.
\end{aligned}$$

So far, we have seen how to update the weights with respect to one training example. Let's now see how we can update them for an entire training set:

Repeat until convergence:

$$w^{(d)} := w^{(d)} - \alpha \sum_{i=1}^{m} \left(h_w(x_i) - y_i\right) x_i^{(d)} \quad \text{for every } d.$$

This is known as **batch gradient descent** because one update step is performed after the entire dataset has been processed. In many cases, the dataset is too large to handle efficiently using batch gradient descent, which is why another popular method is often used: **stochastic gradient descent**. In this algorithm, we update the weights after each training example is processed. Here, m is the number of training examples in the dataset.

```
Loop {
        for $i = 1$ to $m$, {
            $w^{(d)} := w^{(d)} - \alpha \left(h_w(x_i) - y_i\right) x_i^{(d)}$, (for every $d$)
        }
    }
```

Implementation 2: In the provided `linear_regression.ipynb` notebook (Sections 2 and 3), implement the gradient-based solution for linear regression. Section 2 will guide you through creating the prediction and loss functions, while Section 3 will walk you through the optimization process and training.

Closed-form Solution

Due to the quadratic nature of the loss function and its positive coefficient, the function is convex, as previously mentioned. This implies that the function has only one local (and global) minimum. Consequently, we can solve the problem analytically in a straightforward manner. In order to do so we write the data in vectorial form, with the help of the design matrix and target variables represented as a vector.

$$X = \begin{bmatrix} (x_1)^T \\ (x_2)^T \\ \vdots \\ (x_m)^T \end{bmatrix} \qquad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

We can now rewrite the hypothesis function in vector form as follows:

Since $h_w(x_i) = (x_i)^T w$, we can represent the prediction error for all examples as:

$$Xw - \vec{y} = \begin{bmatrix} (x_1)^T w \\ \vdots \\ (x_m)^T w \end{bmatrix} - \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} h_w(x_1) - y_1 \\ \vdots \\ h_w(x_m) - y_m \end{bmatrix}$$

Using the fact that for a vector $\mathbf{z}$, $\mathbf{z}^T\mathbf{z} = \sum_i z_i^2$, we can rewrite the loss function as:

$$L(w) = \frac{1}{2}(Xw - \vec{y})^T(Xw - \vec{y}) = \frac{1}{2}\sum_{i=1}^{m}(h_w(x_i) - y_i)^2$$

which is the same loss function we saw before, now expressed in matrix form.

To find the weights w that minimize the cost function, we take the derivative of $L(w)$ with respect to w and set it to zero, yielding the point of the local minimum. This gives us the following closed-form solution:

$$w = \left(X^T X\right)^{-1} X^T \vec{y}.$$

For further details on this process, please refer to [3].

Implementation 3: In the provided `linear_regression.ipynb` notebook (Section 5), implement the closed-form solution for linear regression. Then, proceed to Section 6 for the weight analysis visualization.

Question 2: When do you think is recommended to use each of the presented solutions. Briefly , explain the obtained results from all the optimization methods (stochastic gradient descent, batch gradient descent, closed form).

Logistic Regression

Overview

Despite its name, logistic regression is actually a binary classification algorithm that applies a threshold to a probability obtained from a linear model combined with the sigmoid function.

The main objective is to implement logistic regression for a simple binary classification problem, following all steps: data preprocessing, implementing the prediction and loss functions, training, and testing the model.

Approach

We will work with the breast cancer dataset [4], where the task is to classify whether a person has breast cancer based on a set of symptoms and patient features. For this model, we will approach the problem from a probabilistic perspective, which will help us in formulating the optimization process.

Since the labels in our classification problem are 0 or 1, we can interpret the output of the model as the probability that a person has breast cancer. Therefore, our model should return a value between 0 and 1. Unlike the PLA, which uses a hard threshold, we need a function that outputs continuous values between 0 and 1. Fortunately, there is a function that does exactly this, known as the **sigmoid function** 1.

The sigmoid function is defined as:

$$g(x) = \frac{1}{1 + e^{-x}}$$

Next, we need to choose a hypothesis function. Recall that the hypothesis function takes as input a feature vector of dimension D and outputs a continuous value between 0 and 1. By combining a linear model with the sigmoid function, and ensuring that the output lies between 0 and 1, we obtain the final hypothesis:

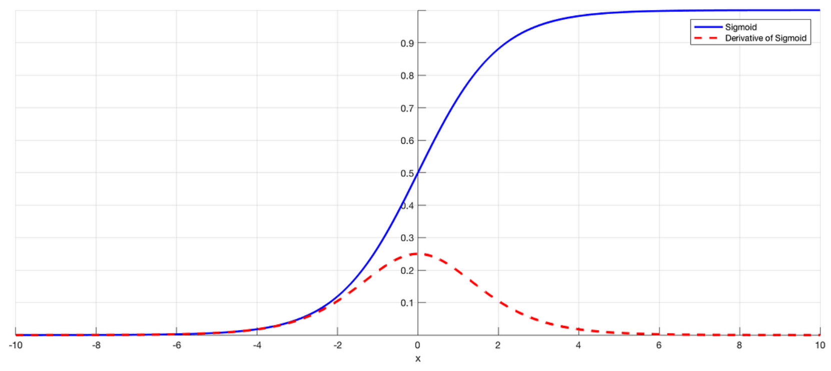

Figure 1: The sigmoid function and its derivative.

$$h(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x}),$$

where $h(\mathbf{x})$ represents the probability that the label is 1 (i.e., the person has breast cancer) given the input features $\mathbf{x}$.

Maximum Likelihood Estimation

As with other models, we aim to optimize the parameters $\mathbf{w}$ to achieve the best performance of our logistic regression model given the data. As already mentioned, let's approach this problem probabilistically. The probability of the data is as follows:

$$P(y = 0 \mid \mathbf{x}, \mathbf{w}) = 1 - h(\mathbf{x}), \quad P(y = 1 \mid \mathbf{x}, \mathbf{w}) = h(\mathbf{x}),$$

where $h(\mathbf{x})$ is the hypothesis function (the sigmoid function in this case). We can rewrite the probability for a given data point $(\mathbf{x}, y)$ as follows:

$$p(y \mid \mathbf{x}; \mathbf{w}) = (h_{\mathbf{w}}(\mathbf{x}))^y \left(1 - h_{\mathbf{w}}(\mathbf{x})\right)^{1-y}.$$

Note that $\mathbf{w}$ is a fixed parameter to be optimized, and we treat this expression as a function of the target variable y. Since our goal is to optimize $\mathbf{w}$, we focus on the **likelihood function**, which is the probability of the entire dataset given $\mathbf{w}$:

$$L(\mathbf{w}) = L(\mathbf{w}; \mathbf{X}, \vec{y}) = p(\vec{y} \mid \mathbf{X}; \mathbf{w}).$$

The likelihood function is what we aim to maximize. Assuming all examples in **X** are IID (Independent and Identically Distributed), we can express the likelihood as:

$$L(\mathbf{w}) = p(\vec{y} \mid \mathbf{X}; \mathbf{w}) = \prod_{i=1}^{m} p(y_i \mid \mathbf{x}_i; \mathbf{w}) = \prod_{i=1}^{m} \left(h_{\mathbf{w}}(\mathbf{x}_i)\right)^{y_i} \left(1 - h_{\mathbf{w}}(\mathbf{x}_i)\right)^{1-y_i}.$$

Since dealing with products can be cumbersome, it is common to maximize the **log-likelihood** instead:

$$\ell(\mathbf{w}) = \log L(\mathbf{w}) = \sum_{i=1}^{m} \left(y_i \log h(\mathbf{x}_i) + (1 - y_i) \log \left(1 - h(\mathbf{x}_i)\right)\right).$$

At this point, we need to choose an optimization method. A simple solution that we've already seen is gradient descent. Maximizing the **log-likelihood** via gradient descent is equivalent to minimizing the negative log-likelihood, also known as the **Binary Cross Entropy Loss (BCE Loss)**. Thus, the formula we wish to minimize is:

$$\ell(\mathbf{w}) = -\log L(\mathbf{w}) = -\sum_{i=1}^{m} \left(y_i \log h(\mathbf{x}_i) + (1 - y_i) \log \left(1 - h(\mathbf{x}_i)\right)\right).$$

Finally, to complete the gradient descent process, we need to compute the update rule by deriving the BCE Loss with respect to the parameters. The update rule is given by:

$$w^{(d)} := w^{(d)} - \alpha \sum_{i=1}^{m} \left(h_{\mathbf{w}}(\mathbf{x}_i) - y_i\right) x_i^{(d)}).$$

In the current setup, there is a mismatch between the hypothesis function and the target variables. The hypothesis predicts a real value in the range $[0, 1]$, while the target labels are discrete values of 0 or 1. To address this, we introduce a **hyperparameter** called a **threshold**, typically set to 0.5.

If the model predicts a value less than 0.5, the final prediction is considered to be 0. If the predicted value is greater than or equal to 0.5, the prediction is considered to be 1.

Note: This thresholding is only applied during test time. During training, we work with the predicted probabilities directly so we can apply the appropriate loss function.

Interestingly, this update rule is very similar to the one for linear regression. This is not a coincidence—both models belong to the family of **Generalized Linear Models (GLMs)**. For more details, see [3].

Implementation 1: Implement the logistic regression model in the provided file `logistic_regression.ipynb`. The following tasks will be necessary to complete this assignment:

- Preprocess the data
- Implement the prediction and loss function
- Create the training pipeline
- Test the obtained model

Polynomial Regression

Overview

Polynomial regression is part of the family of linear regression models, because the linearity of the learned parameters is considered.

The main objective is to implement polynomial regression for a simple problem and adjust the hyperparameters (regularization strength and polynomial degree) to prevent both overfitting and underfitting.

Approach

We define the hypothesis as follows:

$$\hat{y} = w_0 + w_1 x + w_2 x^2 + \cdots + w_n x^n = \sum_{d=0}^{D} w_d x^d$$

where **D** represents the maximum degree of the polynomial. Although linearity is retained at the level of parameters, the features themselves exhibit a nonlinear behavior. For this model, we have adjusted the notation for clarity. The subscript now refers to the dimensionality as well as to the instance in the training set, so please be mindful of the context in which it is used.

Note: The formulas and examples in this model are presented for single-feature data.

Question 1: How can we generalize to multi-feature data? What modifications are necessary?

To optimize the model, we will minimize the **Mean Squared Error (MSE) loss** using the Stochastic Gradient Descent algorithm covered in the Linear Regression chapter.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 + \lambda\sum_{k=0}^{n} w_k^2$$

where m is the number of samples, y_i is the true output for the i-th sample, $\hat{y}_i$ is the predicted output for the i-th sample, and λ is the regularization strength, a hyperparameter that controls the level of regularization applied.

Before discussing regularization and its associated hyperparameter, let us address the phenomena of overfitting and underfitting. These concepts are closely related to the bias-variance trade-off, which is covered in greater theoretical detail in [1], Chapter 2.3. However, here we will focus on practical aspects.

Underfitting occurs when our hypothesis is too simple to adequately approximate the underlying distribution represented by $\mathbf{f}(\mathbf{x})$. This typically results in high in-sample error (or high training loss in practical terms), indicating that the model lacks the complexity needed to capture the data's structure.

Overfitting, on the other hand, refers to the model's inability to generalize to unseen data. While the model may be optimized to minimize in-sample error, it performs poorly on out-of-sample data (test loss in practice). This issue often arises when the model is excessively complex, leading it to fit noise or outliers within the training data, or when certain features dominate the training set, causing the model to focus narrowly on minimizing training error rather than achieving broader generalization.

Practical Solutions for Dealing with Overfitting and Underfitting

In practice, addressing overfitting and underfitting is a delicate process that should be handled progressively. First, we must resolve the underfitting issue by ensuring that the hypothesis set chosen is sufficiently complex to model the underlying data.

Step 1: Ensure that the training error is sufficiently low and that the hypothesis accurately predicts the training data; for now, the test loss can be ignored.

Step 2: Once the training loss and performance metrics on the training set are satisfactory, proceed to the next step, considering the test loss. However, there's a catch: remember the **hyperparameters** that need tuning? What

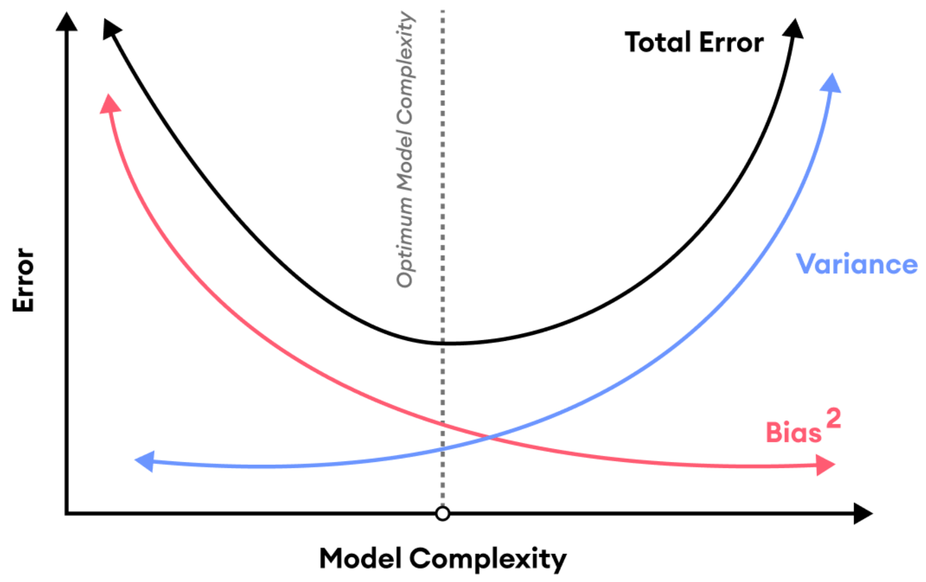

Figure 2: Illustration of the bias-variance trade-off. High bias leads to underfitting, while high variance leads to overfitting. Achieving a balance between bias and variance is crucial for model generalization. Source: [5].

happens if we tune these **hyperparameters** using the test set? This introduces a bias (not to be confused with the bias in the bias-variance trade-off) based on those observations, meaning we don't truly know how the model will perform on unseen data.

To address this, we define a new set called the **validation set**, which is used to tune these hyperparameters and address overfitting. The test set should remain untouched until all hyperparameters and model parameters are finalized, serving as a measure of how the final hypothesis generalizes to unseen data.

Step 3: With the three distinct sets now defined—training, validation, and test—and underfitting resolved, we can tackle overfitting.

There are several approaches to address overfitting, including weight decay, weight regularization, data augmentation, adding more data. Here, we will experiment with weight regularization, which adds an extra term to the loss function. This additional loss term not only penalizes mispredicted values but also discourages large weights by applying a penalty proportional to a chosen norm. The intuition behind the L2 norm, for example, is that it discourages any single weight from overpowering others, ensuring that no feature becomes disproportionately important.

For more guidance on diagnosing and addressing issues when training machine learning models, see this excellent resource: `https://karpathy.github.io/2019/04/25/recipe/`.

Implementation 1: Implement Section 1 of **polynomial_regression.ipynb** in order to preprocess and analyze the data.

Optimization

Now that we have defined the hypothesis and loss function, we need to compute the gradient with respect to the parameters to obtain the gradient vector for the optimization steps.

To minimize the MSE loss, we use the gradient descent approach. The gradient of the MSE loss with respect to each weight w_d is computed as follows.

Define the error term e_i for each data point as:

$$e_i = y_i - \hat{y}_i$$

The gradient of the MSE loss with respect to w_d (for $d = 0, 1, \ldots, D$) is given by:

$$\frac{\partial \text{MSE}}{\partial w_d} = -\frac{2}{m} \sum_{i=1}^{m} e_i \cdot x_i^d + 2\lambda w_d$$

where the term $-\frac{2}{m} \sum_{i=1}^{m} e_i \cdot x_i^d$ represents the gradient of the MSE loss without regularization, and $2\lambda w_d$ is the gradient of the regularization term with respect to w_d.

Thus, the gradient vector for all parameters w is:

$$\nabla_w \text{MSE} = -\frac{2}{m} \sum_{i=1}^{m} (y_i - \hat{y}_i) \cdot \begin{bmatrix} 1 \\ x_i \\ x_i^2 \\ \vdots \\ x_i^n \end{bmatrix} + 2\lambda w$$

This gradient vector is then used in gradient descent to iteratively update w in order to minimize the MSE loss.

Implementation 2: Implement Section 2 of `polynomial_regression.ipynb` by defining the hypothesis, loss function, and evaluation metric. Then, run multiple experiments to observe the impact of different hyperparameter choices (regularization strength and the polynomial degree).

Support Vector Machines (SVM)

Overview

SVM was initially seen as one of the most promising algorithms in machine learning. It can be extended to regression and can be formulated both geometrically and analytically. The strength of SVMs lies in their flexibility, allowing the use of kernel methods which are very powerful and standard optimization algorithms.

To recap, in the Perceptron Learning Algorithm (PLA), the main idea was to iteratively adjust the decision boundary until it separated the classes without errors. This iterative process would stop when all training examples were classified correctly. No constraints were imposed on the decision boundary. In the Polynomial Regression chapter, we discussed the bias-variance tradeoff, regularization methods, and their effects on decision boundaries. Regularizing the model leads to better generalization. For SVMs, the question remains: how should we choose the decision boundary? Figure 3 shows that there are multiple ways to select it. To decide on the optimal boundary, the notion of margin is introduced. A detailed explanation can be found in [3] (Chapter 5).

The main objective is to implement Hard-Margin and Soft-Margin SVMs and select appropriate kernels for handling non-linearly separable data.

Intuition and Margin

In Logistic Regression, we aimed for the absolute value of $w^T x$ to be as high as possible, making the sigmoid $g(w^T x)$ function approach either 0 or 1 as $w^T x$ tended to -∞ or $+\infty$. Using binary cross-entropy loss, we had a high penalty for misclassified points even if they were near the correct label (0 or 1). The idea was that greater confidence in predictions reduced the penalty.

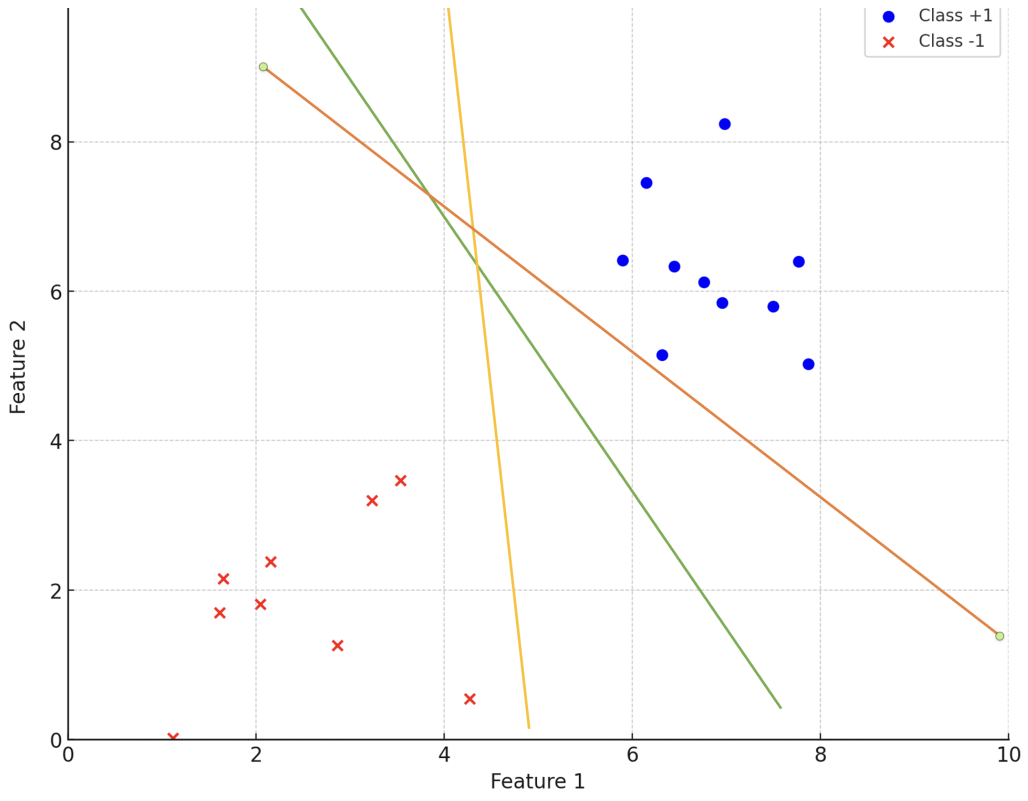

Figure 3: Different ways to select the decision boundary.

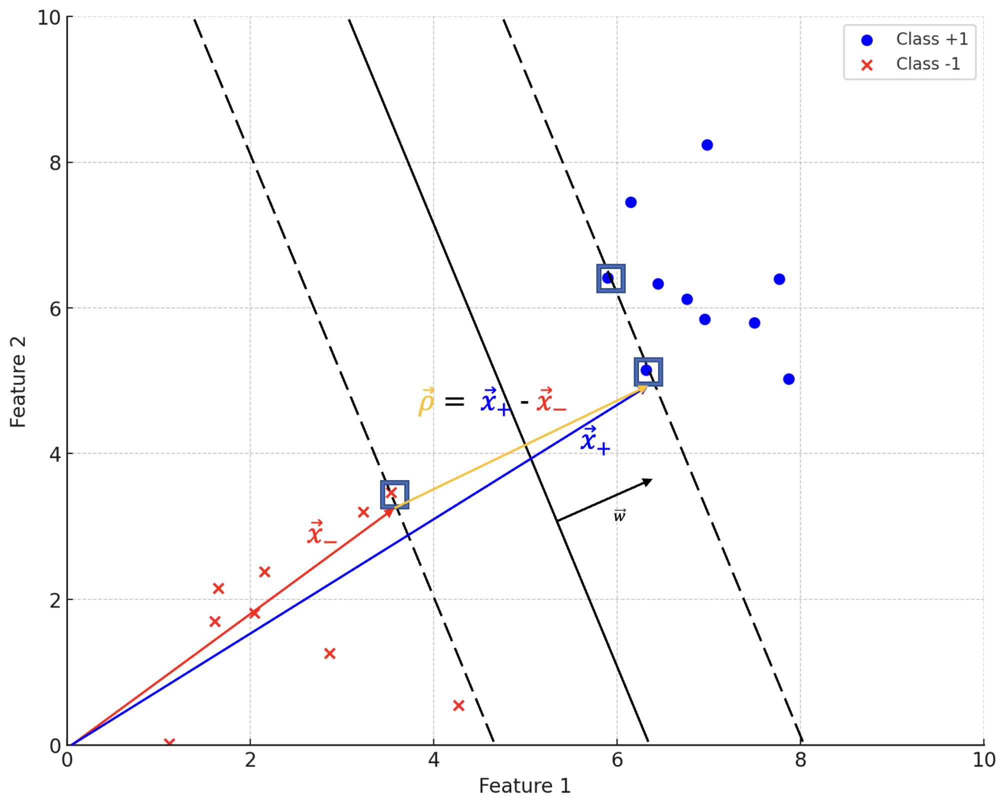

Figure 4: Widest street approach maximizing the distance between the support vectors.

The SVM extends this idea by defining a margin, and our goal is to select the decision boundary that maximize this margin. The intuition is that this margin can be seen as if you would build a street between the points of different classes such that no point lays on the street and the width of the street is maximal. You can see the so called street forming in Fig. 4

Let X be a set of training examples with labels $Y = \{+1, -1\}$. The hypothesis is defined as $\text{sign}(w^T x + b)$, resembling the perceptron. Here, w is the weight vector and b is the bias term. Unlike before, we do not incorporate b into w using a bias trick but keep it separate.

Let's revisit the mathematical framework to define what we aim to achieve. Specifically, we want to create a clear separation between positive and negative samples. This separation can be expressed as:

$$w^T x_+ + b \geq 1 \quad \text{and} \quad w^T x_- + b \leq -1$$

Here, we choose the value 1 as a threshold, meaning that the dot product plus the bias not only needs to have the correct sign but must also exceed a certain distance from the boundary.

This can be rewritten in a more compact form as:

$$y_i(w^T x_i + b) \geq 1$$

Having defined our decision rule, let's recall that our objective is to maximize the width of the "street" that separates the two classes. Figure 4 illustrates the boundaries of this margin as dotted lines, representing the distance we want to maximize. Let's formulate this mathematically.

For points lying on the dotted lines, the following constraints hold:

$$w^T x_{s-} + b = -1$$

$$w^T x_{s+} + b = +1$$

where the samples x_{s+} and x_{s-} are called support vectors, as they satisfy these constraints and are the only points that influence the decision boundary. Other points do not affect the boundary and can be ignored.

Now, to calculate the width of the "street," we take the vector defined by x_{s+} and x_{s-} difference as shown in Figure 4. This vector is crossing the "street," but we need a perpendicular distance to measure the width. Fortunately, we know that w is perpendicular to the decision boundary, so the magnitude of the projection of this vector onto w is the distance we need. This gives:

$$(x_{s+} - x_{s-})\frac{w}{\|w\|}$$

where $\frac{w}{\|w\|}$ is the normalized weight vector, ensuring unit magnitude. By taking the projection of $(x_{s+} - x_{s-})$ onto $\frac{w}{\|w\|}$, we obtain exactly the width of the "street".

Expanding $(x_{s+} - x_{s-})w$, we get $x_{s+}^T w = 1 - b$ and $x_{s-}^T w = b - 1$. Thus, the "street" width becomes:

$$\frac{2}{\|w\|}$$

which we aim to maximize. Maximizing $\frac{2}{\|w\|}$ is equivalent to minimizing $\|w\|$, or more conveniently, minimizing $\frac{1}{2}\|w\|^2$, as this results in a convex function.

In summary, our goal is to maximize the margin by minimizing $\frac{1}{2}\|w\|^2$, subject to the constraint:

$$y_i(w^T x_i + b) \geq 1$$

This setup leads directly to a Lagrangian formulation, which we will briefly outline. For further details, please refer to the suggested resources.

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}\|w\|^2 && (1) \\ \text{s.t.} \quad & y^{(i)}\left(w^T x^{(i)} + b\right) \geq 1, \quad i = 1, \ldots, m && (2) \end{aligned}$$

Writing the Lagrangian Formulation

The Lagrangian formulation for the SVM optimization problem is given by:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{m} \alpha_i \left[y_i(w^T x_i + b) - 1\right]$$

To optimize, we set the derivative to 0 with respect to w and b:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0$$

This implies that:

$$w = \sum_{i=1}^{m} \alpha_i y_i x_i$$

For the derivative with respect to b, we have:

$$\frac{\partial}{\partial b}\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y_i = 0$$

Substituting these results back, we obtain:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \left(x_i\right)^T x_j - b\sum_{i=1}^{m} \alpha_i y_i$$

The last term involving b must be zero, so we finally have:

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} y_i y_j \alpha_i \alpha_j (x_i)^T x_j$$

This leads to the dual optimization problem:

$$\begin{aligned}\max_{\alpha}\ W(\alpha) &= \sum_{i=1}^{m}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle x_i, x_j\rangle \\ \text{s.t.}\quad & \alpha_i \geq 0, \quad i = 1, \dots, m, \\ & \sum_{i=1}^{m}\alpha_i y_i = 0\end{aligned}$$

From here, we can use a convex optimization algorithm to find the Lagrange multipliers. Notice that w depends only on the support vectors, the points for which the Lagrange multipliers α_i are greater than 0.

Since the support vectors satisfy

$$y_i\left(\mathbf{w}\cdot\mathbf{x}_i + b\right) = 1,$$

we can isolate b as:

$$b = y_i - \mathbf{w}\cdot\mathbf{x}_i$$

In practice, we compute b by averaging across all support vectors to get a stable estimate:

$$b = \frac{1}{|S|}\sum_{s\in S}(y_s - \mathbf{w}\cdot\mathbf{x}_s) = \frac{1}{|S|}\sum_{s\in S}\left(y_s - \sum_{i=1}^{m}\alpha_i y_i \langle x_i \cdot \mathbf{x}_s\rangle\right)$$

where S is the set of indices of the support vectors.

We know how to compute b and w let's see how can we compute $w^T x + b$ leveraging the lagrangians coefficients.

$$w^T x + b = \left(\sum_{i=1}^{m}\alpha_i y_i x_i\right)^T x + b = \sum_{i=1}^{m}\alpha_i y_i \langle x_i, x\rangle + b.$$

Now we have all the pieces in order to predict to a new data point we only need to compute the sign of the above formula to get the desired class.

This is known as the Hard Margin SVM, where the margin is fixed, and no classification errors are allowed. In the next section, we will introduce the Soft Margin SVM, which allows for a small margin of error.

Kernels in SVM

So far, the SVM model assumes linearity in the data distribution. For non-linear data distributions, SVM can be extended using a concept called kernels. Recall from polynomial regression that we could enhance features by raising them to powers, thereby mapping data into a higher-dimensional feature space. A similar approach can be applied to SVM, where we use a function ϕ to map original features to a new feature space:

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}$$

Here, ϕ is called a feature map, transforming data from the original space to a higher-dimensional space. Instead of passing x to the SVM, we pass $\phi(x)$.

Question 1: What happens if we choose a very high degree k for the polynomial mapping, making the feature space excessively large?

This brings us to the concept of kernels. In SVM, we are interested not in the explicit mapping $\phi(x)$ itself but in the dot product $\phi(x)^T\phi(z)$. This leads to the kernel function $K(x, z) = \phi(x)^T\phi(z)$, which allows us to compute the dot product in the new feature space without explicitly computing $\phi(x)$. Kernels make SVM efficient and powerful for non-linear problems.

Here are examples of commonly used kernels:

1. Linear Kernel

$$K(x, z) = x^T z$$

Feature Mapping Vector:

$$\phi(x) = x$$

2. Polynomial Kernel

$$K(x, z) = (x^T z + c)^d$$

where c is a constant and d is the polynomial degree.
Feature Mapping Vector (for $d = 2$):

$$\phi(x) = \begin{bmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1x_2 \end{bmatrix}$$

3. Radial Basis Function (RBF) Kernel

$$K(x, z) = \exp\left(-\gamma\|x - z\|^2\right)$$

where γ controls the width of the Gaussian.

4. Sigmoid Kernel

$$K(x, z) = \tanh(\alpha x^T z + c)$$

where α and c are parameters.

The power of kernel functions relies on the fact that we work with a multi dimensional features space, even infinite without the need to explicitly compute the operations in that space. The processing is done entirely in the initial space but the optimization takes place by projecting the input features into the feature maps which with the help of Kernel functions can be used in SVM models.

In practice, a Kernel matrix K will be stored, containing elements equal to $\phi(x_i)^T\phi(x_j)$, where $i, j \in m$ represent training index samples. This matrix is used to directly look up the results of the inner product computed with the help of the Kernel function. In the implementation section, you will probably find it useful when computing b or making the final prediction.

Soft-Margin SVM

In Soft-Margin SVM, we introduce a penalty term C to allow some margin violations. In the Hard Margin SVM no error ware allowed and in order to successfully separate the data, kernels that projected the data into new feature spaces were needed. What if outliers are present and the kernel is not powerful enough? We would like to be a little more permissive with the model. C is a hyper parameter that will specify the degree of permittivity of the model.

$$\min_{w,b,\xi} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \tag{3}$$

$$\text{s.t.} \quad y_i\left(w^T x_i + b\right) \geq 1 - \xi_i, \quad \xi_i \geq 0 \tag{4}$$

The Lagrangian for this problem is:

$$\mathcal{L}(w, b, \xi, \alpha, r) = \frac{1}{2}w^T w + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i\left[y_i\left(w^T x_i + b\right) - 1 + \xi_i\right] - \sum_{i=1}^{m} r_i\xi_i$$

where α_i and r_i are the Lagrange multipliers constrained to ≥ 0. Without going through the full derivation, setting derivatives to zero and simplifying leads to the dual formulation:

$$\max_{\alpha} \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y_i y_j \alpha_i \alpha_j \langle x_i, x_j \rangle$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad \sum_{i=1}^{m} \alpha_i y_i = 0$$

Implementation 1: Implement **svm.ipynb**. In order to complete the assignment, you need to:

- Implement the Hard Margin approach by mapping the corresponding formulas to the `cvxopt` optimization library.
- Implement the Soft Margin approach by mapping the corresponding formulas to the `cvxopt` optimization library.
- Experiment with different settings for both Hard-Margin and Soft-Margin SVMs to observe in which scenarios each approach works or fails, and provide explanations as to why these outcomes occur.

Decision Trees

Overview

The decision tree algorithm is a straightforward algorithm to understand. It can be used for both classification and regression tasks. Here, we are primarily focused on the classification task, but it can be extended to regression tasks quite easily.

The main objective is to implement a decision tree for a simple classification task, following these steps: defining the splitting and gain functions, determining the nodes and corresponding decisions within the tree, and testing the model.

Approach

Let's consider the task of classifying whether it is feasible to ski based on the latitude and the time of year. We have generated a dataset, shown in Figure 5, which you will work with. You can observe that the different regions cannot be separated linearly; instead, the example given attempts to divide the space into different regions, with each region corresponding to a single label. Note that, although it may appear that the central classification region is one area, it is actually a union of multiple regions producing the same label.

To formulate this mathematically, we aim to divide our input space into k different regions, R_k, such that there is no overlap between any of the regions R_i and R_j. Each region will predict a value $\hat{y_k}$ in both classification and regression tasks.

We start with a parent region R_p and split it into subregions R_l and R_r, based on a chosen threshold t and a given feature j of the data points. Mathematically, we can express this as:

$$S_p(j,t) = \left(\{X \mid X_j < t, X \in R_p\}\,,\ \{X \mid X_j \geq t, X \in R_p\}\right)$$

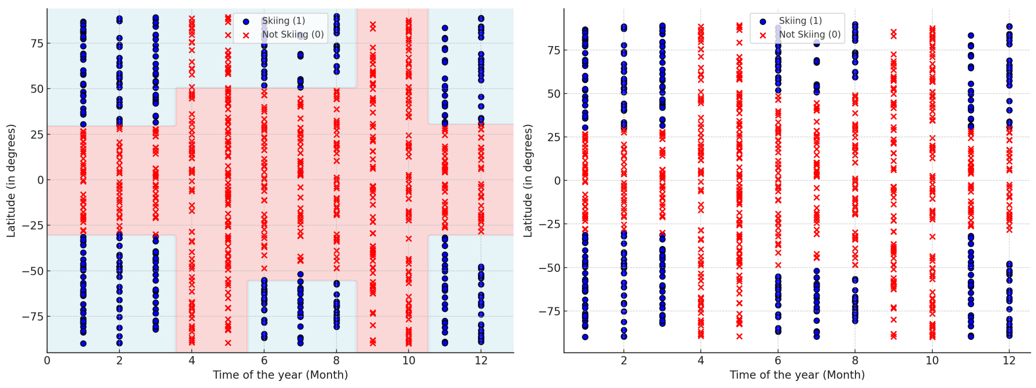

Figure 5: Ski classification dataset based on latitude and month

where S_p represents the split of the parent region R_p at a threshold t for the feature j.

Splits and Loss Functions

Now that we've introduced the concept of decision trees, let's discuss how we determine the splits. The idea is to choose a loss function per region to measure its performance.

One possible loss function is $L(R_k)$, where we define p_c as the proportion of samples in region R_k with the class $y_i = c$. The misclassification loss function is then given by:

$$L_{misclassified}(R_k) = 1 - \max_c p_c$$

If the region contains samples from only one class, the loss is zero; if the region has mixed classes, the loss approaches one.

Returning to the original question of determining the best split, we can now maximize the difference between the loss of the parent region and the sum of the post-split losses. This approach can be formulated as:

$$\max_{j,t} \left(L(R_p) - (L(R_l) + L(R_r)) \right)$$

In other words, we aim to minimize the loss achieved by the split relative to the parent region. This is known as information gain.

Before implementing this, let's consider alternative loss functions. Figure 6 shows three different loss functions: the misclassification loss (in red, as discussed) and two new ones defined as follows (where C is the total number of classes):

$$L_{Entropy}(R_k) = -\sum_{i=1}^{C} p_i \log_2(p_i)$$

$$L_{Gini}(R_k) = 1 - \sum_{i=1}^{C} p_i^2$$

Imagine we have split the region into two new regions R_l and R_r. We already have the loss for R_p, and we can compute the mean loss for the two subregions. Both the Entropy and Gini functions are concave, meaning that the mean loss of the new splits will always be less than the parent region's loss, maximizing information gain. This may not always be the case for misclassification loss.

There is a question that may arise because we have defined that we aim to maximize information gain as the difference between the loss of the parent region and the sum of the losses in the two regions obtained after the split. In Figure 6, the information gain is defined as the difference between the loss of the parent region and the mean loss of the two resulting regions. This approach applies when the number of samples in the two newly obtained partitions is equal.

However, if the number of samples differs between these regions, we must instead use a weighted average. The formula for information gain in this case becomes:

$$\text{Information Gain} = L(R_p) - \left(\frac{S_l}{S} \cdot L(R_l) + \frac{S_r}{S} \cdot L(R_r) \right)$$

where S is the number of samples in the parent region, S_l is the number of samples in the left newly obtained region, and S_r is the number of samples in the right newly obtained region.

Question 1: Under what conditions does the misclassification loss behave as we desire?

A note on Figure 6: we assume equal sample distribution between R_l and R_r when taking the mean. If the samples are unevenly split, the mean should be weighted accordingly. See implementation details in *decision_trees.ipynb*.

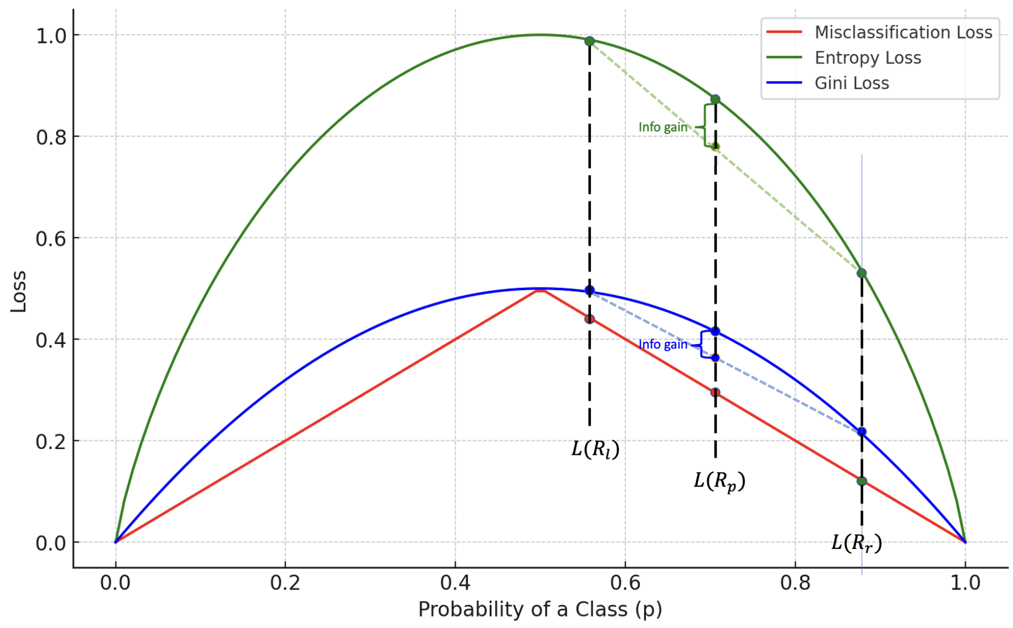

Figure 6: Information gain with respect to different loss functions

Regularization Techniques and Explainability

Decision trees are widely appreciated for their simplicity and interpretability, making them a popular choice for many applications. Key advantages include their high interpretability, robustness to outliers, and ability to handle both continuous and discrete features. Decision trees can also achieve a reasonably good fit relatively quickly, even with large datasets.

However, decision trees also come with notable drawbacks. They tend to overfit to training data, leading to high variance and limited generalization to new data. Even with regularization, they often generalize poorly and can be highly sensitive near boundary values, where small changes in the input may lead to large changes in the model's predictions. To mitigate these issues, regularization techniques can be applied, such as limiting the number of nodes, restricting the tree depth, and setting a minimum sample size required to split a node. We will explore some of these techniques during the implementation to improve model stability and predictive performance.

Implementation 1: Implement and run the **decision_trees.ipynb** file. The following tasks must be accomplished:

- Implement the function that handles the splitting process
- Implement the information gain functions: Gini and Entropy
- Run the model

Ensemble Methods

Overview

Ensemble learning involves combining the outputs of multiple models to make a final decision. This approach can be likened to democratic decision-making, where the majority opinion determines the final outcome. Mathematically, we can represent this as:

$$h(x) = \sum_{t=1}^{T} \alpha_t g_t(x),$$

where α_t are the weights assigned to each model g_t participating in the decision-making process. In some ensemble methods, these weights are equal (e.g., majority voting), while in others, the weights vary depending on model performance.

From a bias-variance perspective, ensemble methods help reduce variance by relying on multiple models, which collectively provide a more robust prediction than a single model setup. This reduces overfitting without significantly increasing bias, as the complexity of individual models remains unchanged.

In summary, ensemble methods involve multiple models (of the same type or different types) contributing to the final prediction, thereby leveraging their collective strengths.

The main objective is to implement the Random Forest algorithm, which uses decision trees, and the AdaBoost algorithm, which uses decision stumps, for simple classification tasks.

Bagging, Random Forests

The first family of ensemble methods we will examine is bagging. Bagging, short for bootstrap aggregating, trains multiple models g_i on different subsets of the dataset. These subsets are generated through sampling with replacement, ensuring diversity in training data for each model. The predictions of these models are then aggregated, for example, by averaging for regression or majority voting for classification.

Question 1: Why not train n different models on the entire dataset?

Random Forests. Previously, we studied decision trees and noted their susceptibility to overfitting. Random forests address this limitation by using bagging combined with additional randomness. The steps for creating a random forest are:

1. From a dataset $X = \{x_1, x_2, ..., x_m\}$, generate n datasets $X_1, X_2, ..., X_n$ by sampling with replacement.
2. Train a decision tree on each sampled dataset.
3. To further reduce correlation among trees, at each node in a tree, randomly select k features (out of D) to determine the best split. This ensures that different trees focus on different features.

This added randomness enhances model diversity, leading to improved generalization.

The prediction for an instance is determined by the majority vote of all the trees within the random forest.

Implementation 1: Implement the Random Forest algorithm in `RandomForests.ipynb`. Since this implementation builds on Decision Trees, ensure you have a solid understanding of them. You will need to implement the required functions to *fit* (train) the model and to make predictions for any given input.

Boosting, AdaBoost

Boosting is another ensemble technique, where models are trained sequentially. Each subsequent model focuses on correcting the errors made by its predecessor. The idea is to combine multiple weak models into a single strong model.

Each model g_t is assigned a weight α_t based on its performance on the training data. Data points that are misclassified by a model are given higher weights, ensuring that subsequent models pay more attention to these difficult cases. This iterative process leads to a combined model that performs better on the overall dataset.

AdaBoost. AdaBoost, short for Adaptive Boosting, is an ensemble learning method that builds a strong classifier, $h(x)$, from a collection of weak classifiers g_t. The key intuition behind AdaBoost is to focus the learning process on the data points that are harder to classify by sequentially training weak models and assigning higher importance to previously misclassified examples.

In each iteration, a weak learner is trained on the dataset, where every data point is assigned a weight. Initially, all data points are weighted equally. After the weak learner makes predictions, the algorithm calculates its error rate. If the weak learner performs poorly, it still contributes to the final model, but with lower importance. Conversely, if it performs well, its influence on the final model increases. This is achieved by assigning a weight (α) to the weak learner based on its error rate.

Next, the data points that were misclassified are given higher weights, making them more significant in the next round of training. By iteratively reweighting the data and combining multiple weak learners, AdaBoost creates a final strong classifier that focuses on the most challenging examples, leading to improved accuracy.

Note: For clarity we would like to mention that there are two types of weights. One that represent the importance of the model in the final decision $h(x)$, denoted with α_t, and the data weights w_i for each data point that are used to represent how difficult is to correctly map the specific example to the correct label.

Mathematically, the final classifier aggregates the predictions of all weak learners, weighted by their importance (α):

$$h(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t g_t(x)\right),$$

where T is the number of weak learners, $g_t(x)$ is the prediction of the t-th weak learner, and α_t is its weight.

Pseudocode for AdaBoost Algorithm

Algorithm 1 AdaBoost

```
Input: Training data $(X, y)$, number of weak learners $T$.
Output: Final strong classifier $h(x)$.
1: Initialize sample weights $w_i = \frac{1}{n}, \forall i = 1, ..., n$.
2: for each round $t = 1, ..., T$ do
3:     Train a weak learner $g_t(x)$ using weights $w_i$.
4:     Compute the weighted error:
           $$\mathrm{err}_t = \frac{\sum_{i=1}^{n} w_i \mathbb{1}[g_t(x_i) \neq y_i]}{\sum_{i=1}^{n} w_i}.$$
5:     Compute the weight of the weak learner:
           $$\alpha_t = \frac{1}{2} \log \left( \frac{1 - \mathrm{err}_t}{\mathrm{err}_t} \right).$$
6:     Update the sample weights:
           $$w_i \leftarrow w_i \exp\big( - \alpha_t y_i g_t(x_i) \big), \quad \forall i.$$
7:     Normalize the sample weights:
           $$w_i \leftarrow \frac{w_i}{\sum_{j=1}^{n} w_j}, \quad \forall i.$$
8: end for
9: Combine weak learners into the final strong classifier:
           $$h(x) = \mathrm{sign} \left( \sum_{t=1}^{T} \alpha_t g_t(x) \right).$$
```

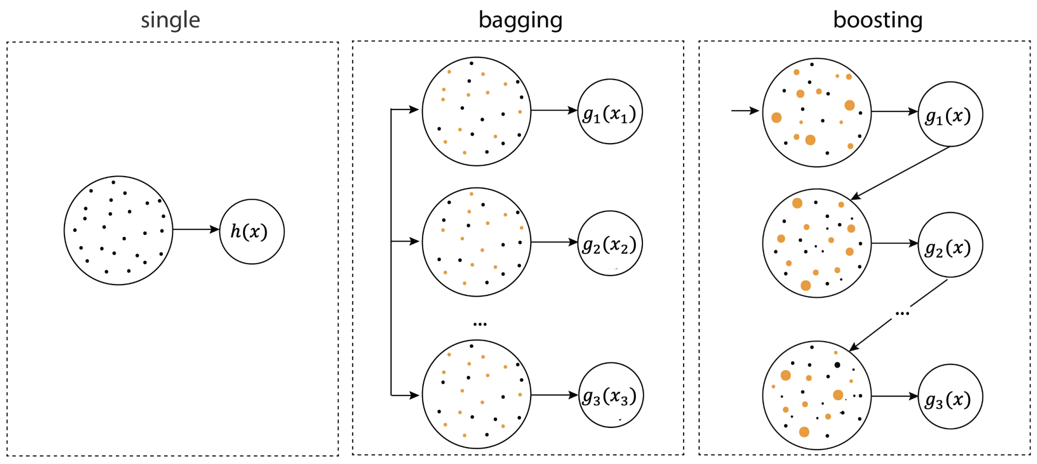

Figure 7: Differences between Boosting and Bagging methods compared to the naive model training approach. Image source [6]

In the implementation of the AdaBoost classifier:

1. The class labels are $y_i \in \{-1, 1\}$. If the dataset does not use these label values, ensure they are converted to the correct format.

2. Decision stumps can be used as weak learners $g_t(x)$. A decision stump is a decision tree with a depth of 1 (a threshold applied to a single feature). When computing the decision tree's loss functions (e.g., Gini impurity or entropy) for splits, the class probabilities should be calculated as the sum of the sample weights w_i for each class normalized by the total sum of all weights.

Implementation 2: Implement the AdaBoost method in `AdaBoost.ipynb`, make sure you also add your *RandomForest* solution in order to make a side by side comparison

Conclusions To conclude this chapter, we would like to clarify the difference between the two ensemble techniques by illustrating them in Fig. 7. Additionally, we want to emphasize once again that ensemble methods are largely agnostic to the models that form the final decision. The complexity constraints are a hyperparameter determined by the user.

Question 2: Should ensemble techniques always be used? Justify your answer.

Naive Bayes Classifier

Overview

Naive Bayes is a probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions between the features. In this chapter, we will delve into the intuition and mathematics behind the Naive Bayes classifier.

The main objective is to implement the Naïve Bayes classifier for classifying handwritten digits from the MNIST dataset [7], following these steps: image preprocessing, probability computation, training and testing the model, and computing accuracy.

Approach

The fundamental idea of Naive Bayes is to calculate the probability of each class $C = c$ given a feature vector $\mathbf{x} \in \mathbb{R}^d$:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}, \tag{5}$$

where x_i represents the i-th feature of the instance. The goal is to assign the class label $c \in \{1, 2, \ldots, K\}$ that maximizes the posterior probability $P(C = c \mid \mathbf{x})$.

Using Bayes' theorem, the probability of class $C = c$ given the instance $\mathbf{x}$ is:

$$P(C = c \mid \mathbf{x}) = \frac{P(\mathbf{x} \mid C = c)P(C = c)}{P(\mathbf{x})}.$$

Since $P(\mathbf{x})$ is constant for all classes during classification, we can focus on maximizing the numerator $P(\mathbf{x} \mid C = c)P(C = c)$. The *naive* assumption is that all features x_i are conditionally independent given the class label C:

$$P(\mathbf{x} \mid C = c) = \prod_{i=1}^{d} P(x_i \mid C = c).$$

This assumption simplifies computation significantly and allows the model to handle high-dimensional data efficiently.

Bernoulli Naive Bayes

In the Bernoulli Naive Bayes model, we assume that all features are binary random variables, taking values 0 or 1. This model is appropriate for data where features represent the presence or absence of a particular attribute, such as word occurrence in text documents or pixel activation in binary images.

For each feature x_i, the likelihood given the class $C = c$ is:

$$P(x_i \mid C = c) = [P(x_i = 1 \mid C = c)]^{x_i} [P(x_i = 0 \mid C = c)]^{1-x_i}.$$

Thus, the joint likelihood is:

$$P(\mathbf{x} \mid C = c) = \prod_{i=1}^{d} \left[P(x_i = 1 \mid C = c)^{x_i} P(x_i = 0 \mid C = c)^{1-x_i}\right].$$

Mathematical Formulation

To predict the class of a new instance $\mathbf{x}$, we compute the posterior probability for each class $C = c$:

$$P(C = c \mid \mathbf{x}) \propto P(C = c) \prod_{i=1}^{d} P(x_i \mid C = c).$$

Taking the logarithm to prevent numerical underflow and to simplify multiplication into addition:

$$\log P(C = c \mid \mathbf{x}) = \log P(C = c) + \sum_{i=1}^{d} \left[x_i \log P(x_i = 1 \mid C = c) + (1 - x_i) \log P(x_i = 0 \mid C = c)\right].$$

We then choose the class with the highest $\log P(C = c \mid \mathbf{x})$:

$$\hat{c} = \arg \max_{c \in \{1,2,\ldots,K\}} \log P(C = c) + \sum_{i=1}^{d} \left[x_i \log P(x_i = 1 \mid C = c) + (1 - x_i) \log P(x_i = 0 \mid C = c) \right].$$

Implementation Details with MNIST

In our implementation of the Bernoulli Naive Bayes classifier using the MNIST dataset, we model the problem as follows:

1. MNIST images are grayscale with pixel values ranging from 0 to 255. We binarize the pixel values such that:

$$x_i = \begin{cases} 1, & \text{if pixel intensity } \geq th, \\ 0, & \text{otherwise,} \end{cases}$$

where th is a threshold (e.g., 128).

2. Compute Class Priors $P(C = c)$ these are estimated from the training data:

$$P(C = c) = \frac{\text{Number of samples where } C = c}{\text{Total number of samples}}.$$

3. Compute Feature Likelihoods $P(x_i = 1 \mid C = c)$ of each feature given the class using Laplace smoothing (with smoothing parameter α) to handle zero probabilities:

$$P(x_i = 1 \mid C = c) = \frac{\text{Number of samples where } x_i = 1 \text{ and } C = c + \alpha}{\text{Total samples where } C = c + 2\alpha}.$$

The denominator 2α accounts for the binary nature of the features (0 or 1). We set $\alpha = 1$. For more details about Laplace smoothing you can check Chapter 4 of [3].

4. Log-Probability Computation, we compute the log-probabilities for numerical stability:

$$\log P(C = c \mid \mathbf{x}) = \log P(C = c) + \sum_{i=1}^{d} [x_i \log P(x_i = 1 \mid C = c) \\ + (1 - x_i) \log(1 - P(x_i = 1 \mid C = c))] .$$

5. We compute the class Prediction by assigning the class label with the highest log-probability:

$$\hat{c} = \arg \max_{c \in \{0,1,\ldots,9\}} \log P(C = c \mid \mathbf{x}).$$

By modeling the MNIST dataset in this way, we treat each pixel as a binary feature indicating whether the pixel is active ($x_i = 1$) or inactive ($x_i = 0$). This allows the Bernoulli Naive Bayes classifier to efficiently handle the high-dimensional data of the MNIST images.

Implementation 1: Implement the Naive Bayes classifier in `NaiveBayes.ipynb`. The following tasks must be accomplished:

- Preprocess the data by image thresholding
- Compute the necessary probabilities
- Implement the training stage
- Test the classifier and compute the accuracy

Unsupervised Learning, Clustering

Overview

Up to this point, we have seen a couple of supervised algorithms. We will briefly recap that supervised methods take as input multiple tuples of the form (x_i, y_i), where x_i is the input instance and y_i is the associated label. We train the algorithms using a training set in order to hopefully obtain good predictions on unseen data. To test generalization capabilities, we evaluate the model on a so-called test set, using different evaluation metrics depending on the task at hand.

In this chapter, we will introduce the concept of unsupervised learning, and we will explain and implement two types of clustering algorithms: namely K-means and DBSCAN.

The main objective is to implement the K-means and DBSCAN algorithms and evaluate the results by applying two evaluation metrics: the Silhouette index and the Dunn coefficient.

Clustering

Clustering is one of the most important unsupervised learning tasks. In simple terms, it deals with the problem of grouping together closely related instances based on their associated features, without knowing the final cluster to which each instance should be assigned during the training stage. According to [8], the following criteria describe the clustering task:

- Instances within the same cluster must be as similar as possible.
- Instances in different clusters must be as different as possible.
- The measurement for similarity and dissimilarity must be clear and have practical meaning.

Clustering algorithms can be divided into multiple categories. Here, we will focus on two of them, with one example each: partition-based algorithms and density-based algorithms. For further reading on other categories and algorithms associated with each, you can refer to [8].

K-Means

One of the most well-known clustering algorithms is K-means. It belongs to the category of partition-based clustering algorithms. The intuition behind K-means is quite straightforward: we select a number k of clusters that we want to divide our data into, prior to training. Each cluster is represented by a centroid $c_i \in \mathbb{R}^d$, where $i \in \{1, \ldots, k\}$ and d is the dimensionality of the training instances. The algorithm works iteratively:

1. First, randomly initialize the centroids of each cluster.
2. Iteratively assign each instance to the closest centroid, update the centroids based on the formed clusters, and repeat this process until convergence.

Before writing the pseudocode for this first clustering algorithm, let's clarify some terms used in the steps outlined above. We refer to a cluster as a set of instances $C_i = \{x_1, x_2, \ldots, x_l\}$, where $x_1, x_2, \ldots, x_l$ are instances from the training set that belong to the same cluster. In the case of K-means, they belong to the same cluster because the centroid c_i associated with cluster C_i is the closest to them. It is also important to emphasize that the centroids are not fixed; they are initially chosen as random vectors and then iteratively updated to move closer to the mean of the instances in the cluster. Centroids can be viewed as representing the mean of each cluster.

Pseudocode

Algorithm 2 K-Means Clustering Algorithm

```
Input: Training data $X = \{x_1, x_2, \ldots, x_n\}$, number of clusters $k$.
Output: Final cluster assignments and centroids.
 1: Randomly initialize centroids $C = \{c_1, c_2, \ldots, c_k\}$.
 2: repeat
 3:     for each instance $x_i \in X$ do
 4:         Assign $x_i$ to the closest centroid $c_j$, based on a distance metric (e.g., Euclidean distance).
 5:     end for
 6:     for each cluster $C_j$ do
 7:         Update centroid $c_j$ as the mean of all instances assigned to it:
                $$c_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i.$$
 8:     end for
 9:     Check for convergence: If centroids do not change significantly, stop.
10: until convergence is reached
11: Return: Updated centroids $C = \{c_1, c_2, \ldots, c_k\}$ and cluster assignments.
```

Limitations

K-means is guaranted to converge in a finite number of steps but not to a global minima. For the proof check out [9].

Question 1: What is the main downside of K-Means?

Evaluation metrics

Before jumping to the next algorithm an important aspect has to be elaborated namely how do we evaluate the performance of an unsupervised learning algorithm since we do not have any labels attributed to them?

Silhouette Index

To calculate **silhouette coefficient**, cluster cohesion (a) and cluster separation (b) must be calculated. Cluster **cohesion** refers to the average distance between an instance and all other data points within the same cluster while cluster **separation** refers to the average distance between an instance and all other data points in the nearest cluster. Silhouette coefficient can be calculated as shown below. Silhouette coefficient is essentially the difference between cluster **separation** and **cohesion** divided by the maximum of the two.[10]

- A value close to $+1$ indicates that the instance is well-clustered and far from other clusters.
- A value close to -1 indicates that the instance might belong to another cluster.
- A value close to 0 indicates that the instance is on the boundary between two clusters.

For each instance x_i, the silhouette value $s(x_i)$ is defined as:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{\max(a(x_i), b(x_i))},$$

where:

$a(x_i)$ is the average distance between x_i and all other instances in the same cluster C_j to which x_i belongs:

$$a(x_i) = \frac{1}{|C_j| - 1} \sum_{x_k \in C_j, x_k \neq x_i} d(x_i, x_k),$$

where $d(x_i, x_k)$ represents the distance between two instances x_i and x_k.

$b(x_i)$ is the minimum average distance between x_i and all instances in any other cluster C_l:

$$b(x_i) = \min_{l \neq j} \frac{1}{|C_l|} \sum_{x_k \in C_l} d(x_i, x_k).$$

The overall **Silhouette Score** for the entire clustering solution is the average of silhouette values over all instances:

$$S = \frac{1}{n} \sum_{i=1}^{n} s(x_i).$$

A higher value of S indicates better-defined clusters.

Dunn Coefficient

The **Dunn Coefficient** is another clustering evaluation metric that focuses on two key aspects:

1. The minimum inter-cluster distance (i.e., the distance between the closest instances of different clusters).
2. The maximum intra-cluster distance (i.e., the distance between the farthest instances within the same cluster).

The Dunn Coefficient is defined as:

$$D = \frac{\min_{i \neq j} \min_{x_i \in C_i, x_j \in C_j} d(x_i, x_j)}{\max_k \max_{x_k, x_l \in C_k} d(x_k, x_l)},$$

where:

$\min_{i \neq j} \min_{x_i \in C_i, x_j \in C_j} d(x_i, x_j)$ represents the minimum distance between any two instances from different clusters C_i and C_j.

$\max_k \max_{x_k, x_l \in C_k} d(x_k, x_l)$ represents the maximum distance between any two instances within the same cluster C_k.

The Dunn Coefficient is designed such that a higher value of D indicates better clustering, where clusters are well-separated, and intra-cluster distances are small. In practice, maximizing D often leads to better-defined and more compact clusters.

Implementation 1: Implement `kmeans.ipynb`. The following tasks must be accomplished:

- Initialize the centroids
- Iteratively update the centroids
- Ensure the solution converges
- Implement Dunn Index and Silhouette Score.
- Test the solution

DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

In the previous section we saw one of the most famous clustering algorithms based on partition based techniques. A big downside of K-Means is that the number of clusters has to be specified. We will now talk about a density based algorithm where the number of clusters do not need to be specified beforehand. DBSCAN is a density-based clustering algorithm that groups together instances that are closely packed together (groups together denser regions), marking as outliers those instances that lie alone in low-density regions. It is particularly useful for handling data with noise and outliers.

The intuition behind DBSCAN is simple: it identifies dense regions of data points separated by sparse regions. The algorithm requires two parameters:

- ϵ: the radius of the neighborhood around a point.
- *MinPts*: the minimum number of points required to form a dense region (i.e., a cluster).

The algorithm proceeds as follows:

1. **Core Points**: A point is considered a **core point** if at least *MinPts* points are within a distance ϵ of it.

2. **Border Points**: A point is considered a **border point** if it is within ϵ of a core point but does not have enough neighbors to be considered a core point itself.

3. **Noise Points**: A point is considered **noise** if it is not a core point or a border point.

4. **Clustering**: Starting from a random unvisited core point, the algorithm expands the cluster by recursively adding neighboring points that are either core points or border points. This process continues until all reachable points have been assigned to the cluster. If a point cannot be reached from any core point, it is classified as noise.

Pseudocode

Algorithm 3 DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Input: Dataset D, Epsilon (ϵ), MinPts.
Output: Set of clusters and noise points.

```
 1: $C = 0$                                        ▷ Initialize cluster counter.
 2: for each unvisited point $P$ in dataset $D$ do
 3:     Mark $P$ as visited.
 4:     NeighborPts = regionQuery($P, \epsilon$)
 5:     if |NeighborPts| < MinPts then
 6:         Mark $P$ as NOISE.
 7:     else
 8:         $C = C + 1$                            ▷ Create a new cluster.
 9:         expandCluster($P$, NeighborPts, $C$, $\epsilon$, MinPts)
10:     end if
11: end for
12: procedure EXPANDCLUSTER(P, NeighborPts, C, epsilon, MinPts)
13:     Add $P$ to cluster $C$.
14:     for each point $P' \in$ NeighborPts do
15:         if $P'$ is not visited then
16:             Mark $P'$ as visited.
17:             NeighborPts′ = regionQuery($P', \epsilon$)
18:             if |NeighborPts′| ≥ MinPts then
19:                 NeighborPts = NeighborPts ∪ NeighborPts′
20:             end if
21:         end if
22:         if $P'$ is not yet a member of any cluster then
23:             Add $P'$ to cluster $C$.
24:         end if
25:     end for
26: end procedure
27: function REGIONQUERY(P, epsilon )
28:     return all points within $P$'s $\epsilon$-neighborhood (including $P$).
29: end function
```

Implementation 2: Implement **dbscan.ipynb**. In order to complete the assignment, the following tasks must be accomplished:

- Implement the *region query* and **expand** functions.
- Implement the DBSCAN algorithm.
- Implement Dunn Index and Silhouette Score.
- Test the solution.

Question 2: What can you say about the evaluation metrics results for both clustering algorithms?

Bibliography

[1] Y. S. Abu-Mostafa, M. Magdon-Ismail, and H.-T. Lin, *Learning from data.* AMLBook New York, 2012, vol. 4.

[2] R. K. Pace and R. Barry, "California housing dataset," https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html, 1997, dataset derived from the 1990 U.S. Census. See also: Pace, R. Kelley and Barry, Ronald, "Sparse Spatial Autoregressions," *Statistics and Probability Letters*, 33:291–297, 1997.

[3] A. Ng, "Cs229 lecture notes," *CS229 Lecture notes*, vol. 1, no. 1, pp. 1–3, 2000.

[4] W. Wolberg, O. Mangasarian, N. Street, and W. Street, "Breast Cancer Wisconsin (Diagnostic)," UCI Machine Learning Repository, 1993, DOI: https://doi.org/10.24432/C5DW2B.

[5] V. Tassopoulou, "An exploration of deep learning architectures for handwritten text recognition," Master's thesis, School of Electrical and Computer Engineering (ECE), Division of Signal, Control and Robotics, Computer Vision, Speech Communication and Signal Processing Group, November 2019. [Online]. Available: https://www.researchgate.net/publication/337368016_An_Exploration_of_Deep_Learning_Architectures_for_Handwritten_Text_Recognition

[6] A. Senozan, "Ensemble: Boosting, bagging, and stacking machine learning," 2023, accessed: 2024-11-23. [Online]. Available: https://medium.com/@senozanAleyna/ensemble-boosting-bagging-and-stacking-machine-learning-6a09c31df778

[7] Y. LeCun, C. Cortes, and C. J. C. Burges, "The mnist database of handwritten digits," http://yann.lecun.com/exdb/mnist/, 1998, accessed: 2025-02-25.

[8] D. Xu and Y. Tian, “A comprehensive survey of clustering algorithms,” *Annals of data science*, vol. 2, pp. 165–193, 2015.

[9] S. Z. Selim and M. A. Ismail, “K-means-type algorithms: A generalized convergence theorem and characterization of local optimality,” *IEEE Transactions on pattern analysis and machine intelligence*, no. 1, pp. 81–87, 1984.

[10] H. Belyadi and A. Haghighat, *Machine learning guide for oil and gas using Python: A step-by-step breakdown with data, algorithms, codes, and applications.* Gulf Professional Publishing, 2021.