Machine Learning Applications in Physics

A thesis submitted in partial fulfilment of the requirements for the degree of

Bachelor of Sciences

in

Physics

By

Ali Shahryar Khokhar

24100266

Supervisor: Dr. Adam Zaman Chaudary

School of Science and Engineering

Lahore University of Management Sciences

May 2023

Abstract: This paper provides an overview of Machine Learning (ML) techniques and their potential applications in physics for solving intricate problems. We cover several ML algorithms, with emphasis on Logistic Regression, Decision Trees, and Neural Networks, to give a complete understanding to undergraduate physics students. These algorithms are exemplified in physics through three separate problems. (1) We used logistic regression to classify air showers measured by the MAGIC telescope, achieving an accuracy of 78.9% and an AUC of 0.742. (2) We utilized Decision Trees to predict critical temperatures for superconductivity, obtaining a L^2 Loss of 12.12 and a R^2 score of 0.874. (3) We employed Neural Networks to separate signal and background in an exotic Higgs scenario, resulting in an accuracy of 70% and an AUC of 0.801. These instances highlight how ML can revolutionize solving complex physics problems, emphasizing the need to incorporate ML methods in physics education.

Contents

1 Introduction

The field of physics has witnessed remarkable advancements over the years, with discoveries and developments in various sub-domains continually pushing the boundaries of human knowledge. However, as the complexity of problems in physics grows, traditional analytical and computational methods often struggle to provide accurate and efficient solutions. Machine learning (ML) has emerged as a powerful interdisciplinary tool that can address these challenges and offer new avenues for research in physics.

In this paper, we focus on three machine learning techniques - decision trees, logistic regression, and artificial neural networks - and their applications to specific problems in physics. The motivation for using these ML techniques is multifaceted and is rooted in both mathematical and intuitive reasoning.

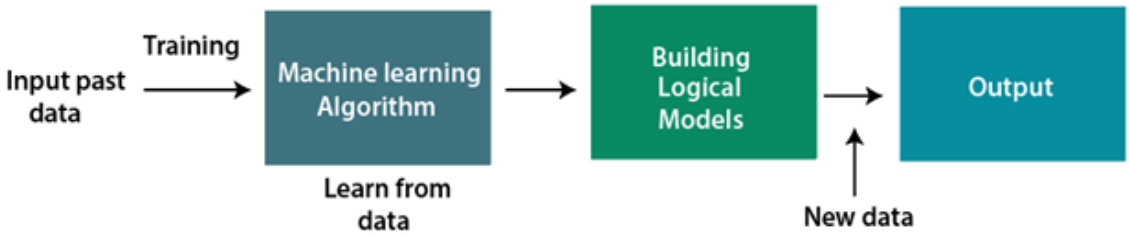
1.1 Understanding Machine Learning Algorithms

Machine learning algorithms are computational methods designed to learn patterns and extract insights from data. These algorithms enable the creation of models that can make predictions, classify data, or discover hidden relationships within the data without explicit programming. In this sub-section, we provide an intuitive overview of the core concepts underlying ML algorithms and the general steps taken by practitioners to apply them.

1.1.1 What is a Machine Learning Algorithm?

A machine learning algorithm is like a tool that takes in data and produces an output. Its purpose is to learn from the training data and find the most accurate way to map the input data to the output data. During the training process, the function's parameters are adjusted to make the model's predictions as close as possible to the actual data.

Figure 1. A block diagram of the general machine learning process.



Mathematically, an ML algorithm can be represented as:

$$f_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{y} \tag{1.1}$$

where $\boldsymbol{x}$ represents the input data, $\boldsymbol{y}$ represents the output data, $f_{\boldsymbol{w}}$ is the function representing the ML model, and $\boldsymbol{w}$ is the set of parameters (or weights) that are learned

during training.

1.1.2 General Steps for Applying Machine Learning Algorithms

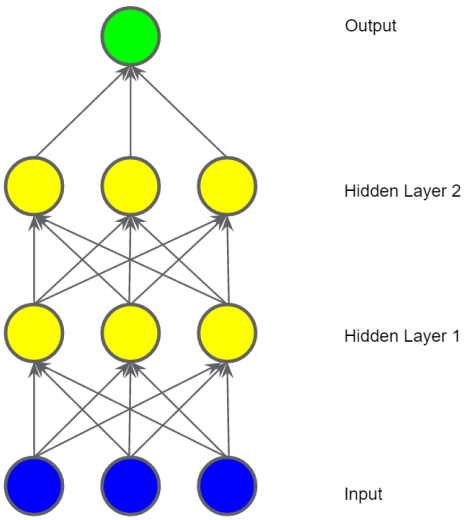
1. **Define the problem**: Clearly define the problem to be solved, which could be a classification, regression, clustering, or reinforcement learning task. This step involves understanding the underlying challenge and identifying the specific goals of the ML model.

2. **Collect and prepare the data**: Gather the data required for training and validation, and preprocess it to ensure that it is suitable for training the ML model. Data preparation typically involves cleaning, normalizing, encoding categorical variables, balancing class distribution, and splitting the data into training, validation, and testing sets.

3. **Select an appropriate machine learning algorithm**: Choose an ML algorithm that is well-suited to the problem at hand. This decision is based on the type of problem, the characteristics of the data, and any domain-specific requirements or constraints. Common ML algorithms include linear regression, logistic regression, decision trees, support vector machines, and neural networks.

4. **Design the model architecture**: Define the structure of the ML model, which includes selecting the appropriate features, defining the model's layers or components, and specifying any necessary hyperparameters. The model architecture should be tailored to the specific problem and data characteristics.

Figure 2. An example of the architecture of a machine learning model called a neural network.



5. **Train the model**: Use the training data to adjust the model's parameters (or weights) such that the model can make accurate predictions. This step typically involves minimizing a loss function, which quantifies the difference between the model's

predictions and the ground truth data. Common optimization algorithms include gradient descent, stochastic gradient descent, and more advanced techniques like Adam and RMSprop.

6. **Evaluate the model**: Evaluate how well the model performs by testing it on a different set of data specifically reserved for validation or testing purposes. This step helps to predict how accurately the model will perform on data that it has not been trained on. When evaluating a solution to a problem, common metrics used are accuracy, precision, recall, F1 score, and mean squared error. The specific metric used depends on the type of problem being solved.

7. **Fine-tune the model**: Based on the model's performance during evaluation, adjust the model architecture, hyperparameters, or training strategy as needed to improve its performance. This process may involve iterating through several cycles of training and evaluation.

8. **Deploy the model**: After completing the training, validation, and fine-tuning of the model, it is ready to be used for making predictions on new data that it has not seen before. This step may require you to do one of the following: - Combine the model with a bigger system - Create an interface that is easy for users to navigate - Turn the model into a web service or API. When you deploy a model, it's important to keep an eye on its performance and make updates as needed. This will help ensure that the model stays accurate and relevant as new data comes in or as the problem it's solving evolves over time.

1.1.3 Practical Considerations

When applying machine learning algorithms, practitioners should be mindful of several practical considerations that can impact the success of their models:

Feature engineering: The choice of input features can significantly affect the performance of an ML model. Practitioners should carefully consider which features are most relevant to the problem and may need to perform feature selection, dimensionality reduction, or feature extraction to obtain the most informative inputs.

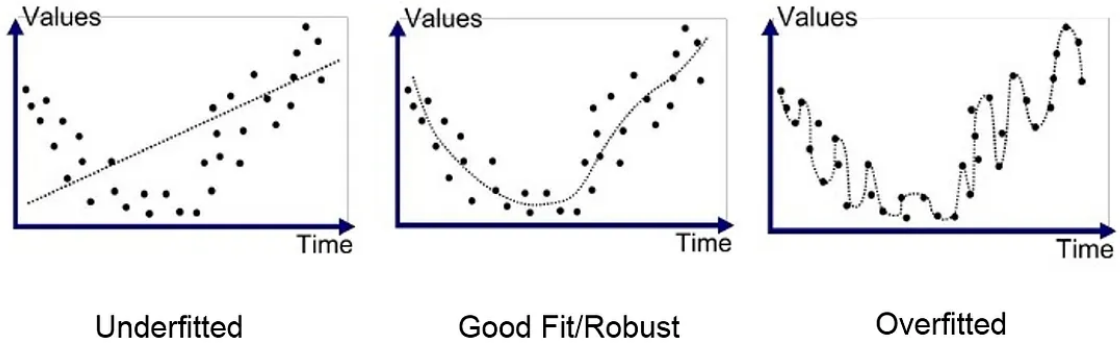
Figure 3. An example of feature engineering is transformation. This figure shows an example of how a log-transformation linearizes the data.

Model interpretability: Some ML models, such as deep neural networks, can be difficult to interpret and may act as "black boxes." In some applications, it is crucial to understand the inner workings of the model and be able to explain its predictions. In such cases, simpler models like linear regression or decision trees may be more appropriate.

Computational resources: Training and deploying ML models can be computationally intensive, particularly for large datasets or complex models. Practitioners should consider the available computational resources when selecting an algorithm and designing the model architecture. In some cases, it may be necessary to use distributed computing, specialized hardware like GPUs, or cloud-based services to handle the computational demands of the model.

Overfitting and underfitting: ML models can suffer from overfitting, where the model learns the noise in the training data rather than the underlying patterns, or underfitting, where the model is too simple to capture the complexity of the data. Practitioners should be aware of these issues and use techniques like regularization, early stopping, or model selection to mitigate their effects.

Figure 4. This figure demonstrates underfitting and overfitting on sample data.



Ethical considerations: The application of ML algorithms can raise ethical concerns, particularly in areas like data privacy, fairness, and accountability. Practitioners should be aware of these issues and take steps to ensure that their models are developed and deployed in a responsible and ethical manner.

In summary, applying machine learning algorithms to solve problems in physics or other domains involves a systematic process that includes defining the problem, selecting an appropriate algorithm, designing the model architecture, training and evaluating the model, and deploying the model for real-world use. By following this process and considering the practical aspects of working with ML algorithms, practitioners can develop effective models that address challenging problems and provide valuable insights.

1.2 Motivation for Using Supervised Machine Learning in Physics

Machine learning, particularly supervised learning, has gained considerable attention in physics. There are various reasons for this, predominantly stemming from the ability of supervised learning to handle complex datasets, discover underlying structures, and make accurate predictions.

1. **Model-free Predictions**: Physics is traditionally a model-based field, with theories and equations derived from principles and laws. However, many physical phenomena are too complex or high-dimensional to be easily modelled. For example, the three-body problem in celestial mechanics involves calculating the motion of three celestial bodies under mutual gravitation. Due to the complex interactions, it's impossible to obtain a general analytical solution. Supervised machine learning, on the other hand, can be trained on numerically generated data, allowing it to make accurate predictions about system behaviour, without needing an explicit analytical model.

2. **Detection of Non-linearities and Interactions**: Physical systems frequently exhibit non-linear relationships and interactions among variables. Consider phase transitions in condensed matter physics. The critical temperatures at which these transitions occur can be influenced by a multitude of non-linearly interacting factors. Traditional linear statistical models often fall short in such scenarios, while supervised learning algorithms can model these complex relationships effectively.

3. **Feature Learning and Representation**: In tasks such as particle tracking in high-energy physics or image analysis in astronomy, raw data often comprises high-dimensional images. Manual feature extraction from such data can be impractical. Supervised learning algorithms, particularly deep learning, excel at learning useful representations automatically, enabling more efficient and accurate analysis.

While these advantages are promising, there are also challenges and concerns that physicists must be aware of when using supervised machine learning:

1. **Training Data Generation**: In supervised learning, a model is trained on a set of labelled data. For many physical problems, generating these labels might require solving the problem using traditional methods, which can be computationally expensive. For instance, in quantum mechanics, obtaining the ground state of a many-body system often involves solving the Schrödinger equation, which can be computationally prohibitive for large systems. Yet, once the initial dataset is generated and the model trained, it can predict properties of similar systems with significantly less computational cost.

2. **Interpretability and Physical Consistency**: Many machine learning models, particularly neural networks, can appear as 'black boxes', making it difficult to interpret their predictions. In physics, where understanding the underlying mechanisms

is paramount, this can be a disadvantage. However, methods are being developed to increase the interpretability of such models. For instance, in the case of neural networks, techniques like saliency maps or feature visualization can be used. Alternatively, simpler models like decision trees may be used when interpretability is crucial.

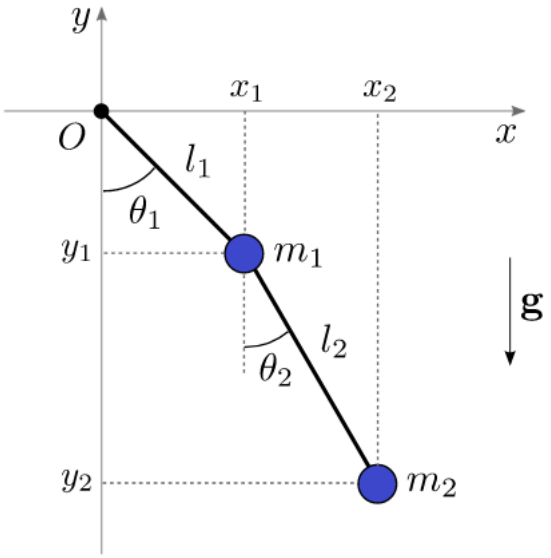
3. **Overfitting and Generalization**: A common challenge in machine learning is overfitting, where a model learns the noise in the training data, leading to poor generalization on unseen data. Physicists should use techniques like regularization and cross-validation to ensure their models generalize well. For example, a model predicting the phase of a material might fit well on the training data but fail on unseen test data if it has learned noise or specificities of the training set rather than the underlying physics.

Despite these challenges, the potential of machine learning in physics is vast. Physicists need to carefully consider the advantages and challenges, and decide on the best use of supervised learning in their specific research problem. Implementing these techniques requires careful understanding and fine-tuning of the models, considering the specifics of the physics problem at hand.

1.3 A Motivating Example: The Double Pendulum Problem

To further illustrate the potential of machine learning in physics, let us consider a classic toy problem, the double pendulum. The double pendulum consists of two pendulums attached end-to-end, with the first pendulum's pivot point fixed, and the second pendulum's pivot point connected to the end of the first pendulum. This system exhibits chaotic behavior, which can be challenging to model and predict using traditional methods.

Figure 5. A free body sketch of a double pendulum.



1.3.1 Mathematical Formulation of the Problem

The equations of motion for the double pendulum can be derived using the Lagrangian formalism. Let L represent the Lagrangian, which is defined as the difference between the system's kinetic energy T and potential energy V:

$$L = T - V \tag{1.2}$$

The kinetic energy of the system can be written as:

$$T = \frac{1}{2}m_1 v_1^2 + \frac{1}{2}m_2 v_2^2 \tag{1.3}$$

where m_1 and m_2 are the masses of the two pendulums, and v_1 and v_2 are their respective velocities.

The potential energy of the system is given by:

$$V = m_1 g h_1 + m_2 g h_2 \tag{1.4}$$

where g is the acceleration due to gravity, and h_1 and h_2 are the heights of the two pendulums' centers of mass.

Using the Euler-Lagrange equations, we can derive the equations of motion for the system:

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_1}\right) - \frac{\partial L}{\partial \theta_1} = 0 \tag{1.5}$$

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}_2}\right) - \frac{\partial L}{\partial \theta_2} = 0 \tag{1.6}$$

where θ_1 and θ_2 are the angular positions of the two pendulums, and $\dot{\theta}_1$ and $\dot{\theta}_2$ are their respective angular velocities.

Solving these equations simultaneously yields a system of two second-order, nonlinear, coupled ordinary differential equations (ODEs), which are computationally expensive to solve.

1.3.2 Challenges in Solving the Double Pendulum Problem

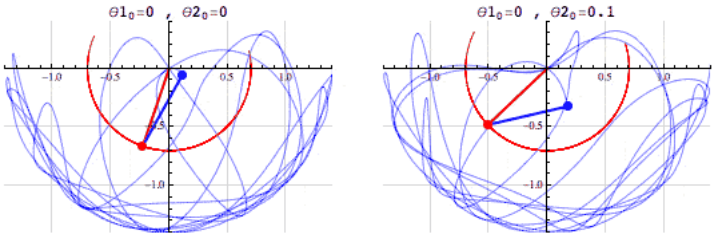
The double pendulum problem is challenging to solve for several reasons:

1. The equations of motion are nonlinear and coupled, making analytical solutions difficult to obtain.

2. Small differences in starting conditions can cause the system to follow very different paths, which is known as chaotic behavior.. This sensitivity to initial conditions makes numerical simulations challenging, as small errors in numerical approximations can quickly accumulate and lead to inaccurate predictions.

3. As the pendulums move through a wide range of positions and velocities, the computational cost of numerical methods, such as Runge-Kutta, can become prohibitively high, particularly for long-duration simulations.

Figure 6. An illustration of chaotic behaviour of a pendulum by perturbing the initial conditions by a small amount.



1.3.3 Applying Machine Learning to the Double Pendulum Problem

Machine learning offers a potential alternative approach to solving the double pendulum problem. Instead of attempting to solve the equations of motion directly, by using ML algorithms, we can understand how the system works based on a set of training data.. This data can be generated by simulating the double pendulum for various initial conditions and recording the corresponding trajectories. After the machine learning model has been trained, it can be used to predict the trajectories of new initial conditions, potentially with lower computational cost compared to traditional numerical methods.

1.4 Another Motivating Example: The N-Body Problem

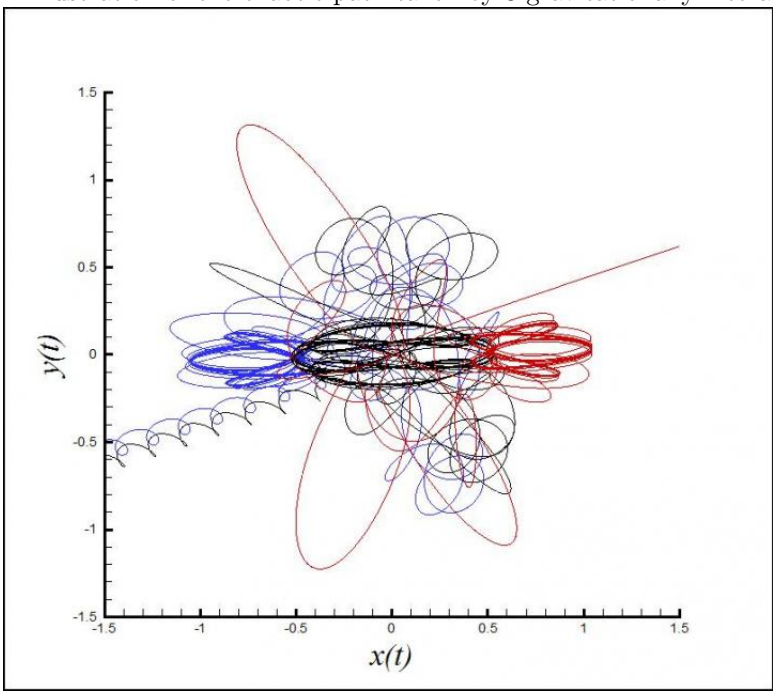
The N-body problem is another motivating problem that demonstrates the potential of machine learning in physics. It involves predicting the motion of N celestial bodies under the influence of their mutual gravitational interactions. This problem is important in areas such as astrophysics and celestial mechanics, where understanding the behavior of celestial bodies is crucial.

1.4.1 Mathematical Formulation of the Problem

Newton's law of universal gravitation describes the gravitational force between two celestial bodies.:

$$F = G\frac{m_1 m_2}{r^2} \tag{1.7}$$

Figure 7. An illustration of the chaotic path taken by 3 gravitationally interacting bodies.



where F is the force between the two bodies, G is the gravitational constant, m_1 and m_2 are their masses, and r is their distance apart.

The equations of motion for the N-body problem can be expressed as a system of nonlinear, coupled ordinary differential equations (ODEs) of the second order:

$$m_i \frac{d^2 \mathbf{r}_i}{dt^2} = \sum_{j=1,\, j \neq i}^{N} G\, m_i m_j \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3} \tag{1.8}$$

for $i = 1, 2, \ldots, N$, where m_i is the mass of the i-th body, and $\mathbf{r}_i$ is its position vector.

1.4.2 Challenges in Solving the N-Body Problem

The N-body problem is difficult to solve for the following reasons:

1. The equations of motion are nonlinear and coupled, making analytical solutions generally unattainable for $N > 2$.

2. Numerical methods for solving the N-body problem, such as direct integration or tree-based methods, can be computationally expensive, particularly for large values of N.

3. The complexity and number of interactions increase with the number of celestial

bodies, making the problem scale poorly with the number of objects involved.

1.4.3 Applying Machine Learning to the N-Body Problem

Machine learning provides a potential alternative approach to solving the N-body problem. Instead of attempting to solve the equations of motion directly, ML algorithms can be used to learn the underlying dynamics of the system from a set of training data. This data can be generated by simulating the N-body system for various initial conditions and recording the corresponding trajectories.

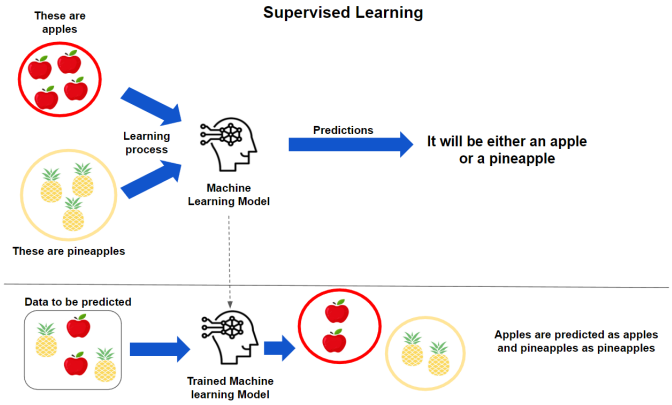
Once the ML model is trained, it can be used to predict the trajectories for new initial conditions, potentially with lower computational cost compared to traditional numerical methods.

1.5 Supervised vs Unsupervised Learning Algorithms

Machine learning algorithms are broadly classified into supervised and unsupervised learning algorithms.

1.5.1 Supervised Learning

Figure 8. A simple illustration of supervised learning.



Supervised learning is a form of machine learning in which an algorithm makes predictions based on training data that has been labeled. An algorithm for supervised learning analyzes training data and generates an inferred function for mapping new examples.

Mathematically, if we denote the training dataset as $D = \{(x_i, y_i)\}_i = 1^N$, where x_i represents the i-th feature vector and y_i is the corresponding label, the goal of a supervised learning algorithm is to learn a function $f : X \longrightarrow Y$ from the training data, where X denotes the input space and Y denotes the output space. The learned function f should be able to predict the correct label y for any new input x.

The learning process involves minimizing a loss function $L(y, f(x))$ that measures the discrepancy between the predicted label $f(x)$ and the true label y. For example,

in regression problems, a common choice of the loss function is the mean squared error: $L(y, f(x)) = (y - f(x))^2$

1.5.2 Unsupervised Learning

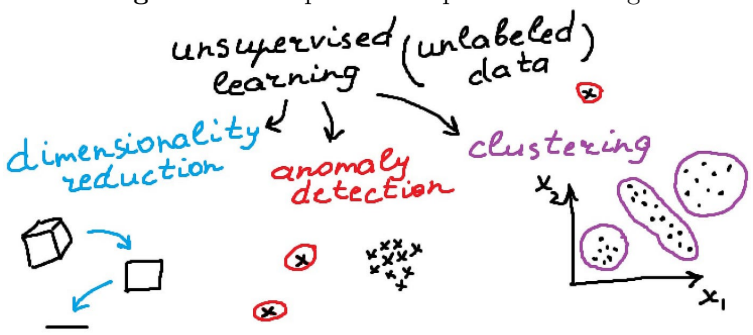
In contrast, unsupervised learning involves training an algorithm with data that is neither categorized nor labeled. Therefore, the algorithm operates on the data without direction. Here, the machine's mission is to group unsorted data based on similarities, patterns, and differences without prior data training.

Cluster analysis is a common unsupervised learning technique used for exploratory data analysis to uncover latent patterns or groupings. Modeling the clusters using a measure of similarity defined by metrics such as Euclidean or probabilistic distance.

Mathematically, given a set of points $D = \{x_i\}_{i=1}^N$ where x_i is an individual instance, clustering algorithms partition the dataset D into several specific groups or clusters $C = \{c_j\}_{j=1}^k$ so that the similarity between instances in the same cluster is maximized and the similarity between instances in different clusters is minimized. The similarity is a measure that reflects the strength of relationship between two data instances. Different similarity measures can be used depending on the type of data.

In the context of physics and material science, these machine learning methods can be utilized to predict material properties, analyze patterns, and make informed decisions regarding future studies and experiments.

Figure 9. Examples of unsupervised learning.



1.6 Types of Supervised Learning

1.6.1 Regression

Regression is a type of supervised learning task that is commonly used in machine learning and statistics. In regression, the goal is to predict a continuous outcome variable (Y) from several input (predictor) variables (X).

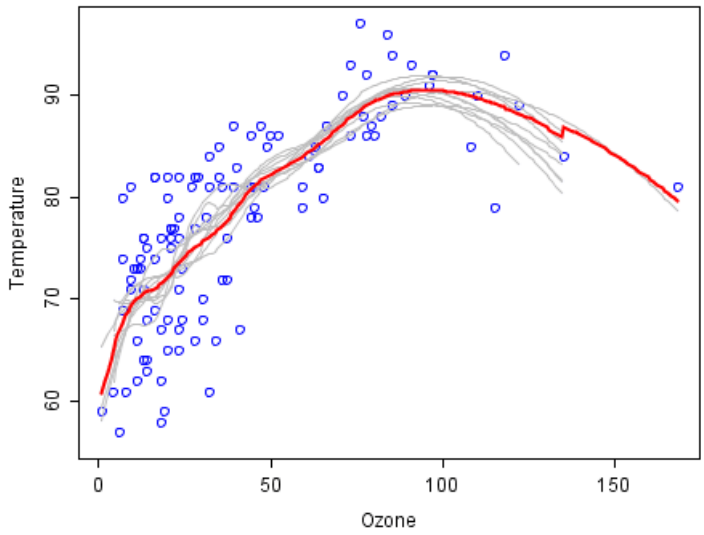
Mathematically, a regression problem is typically represented as:

$$Y = f(X) + \epsilon \tag{1.9}$$

where: - Y is the outcome variable we want to predict, - X is the vector of input variables, - f is the function that maps X to Y, - ϵ is the error term (often assumed to follow a normal distribution).

The goal in a regression problem is to estimate the function f such that the error term ϵ is minimized.

Figure 10. Plot of a regression model fitted to the data of Temperature vs Ozone.



1.6.2 Classification

Classification, on the other hand, is a supervised learning task where the goal is to predict a categorical outcome variable (Y) from several input (predictor) variables (X).

In a binary classification problem, Y can take on two possible outcomes. For multi-class classification problems, Y can take on more than two possible outcomes.

Mathematically, a classification problem can be represented similarly to a regression problem:

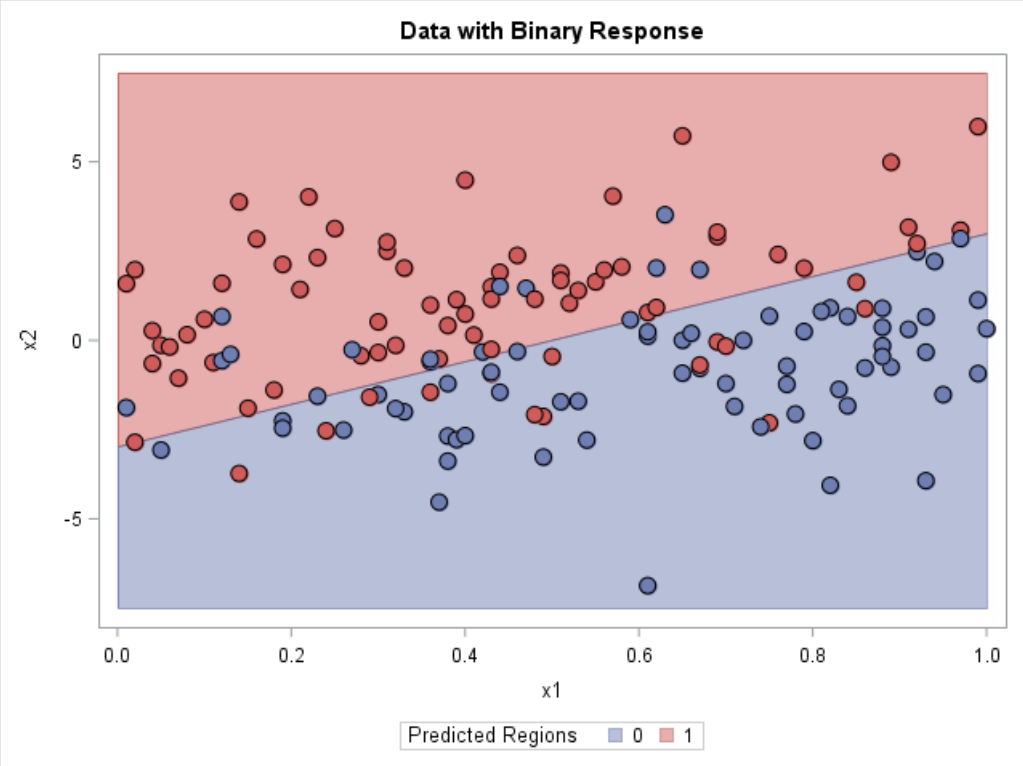
$$Y = f(X) \tag{1.10}$$

where: - Y is the outcome variable we want to predict, - X is the vector of input variables, - f is the function that maps X to Y.

In classification, however, f is a function that divides the feature space into decision regions that correspond to the various classes. These regions' borders are referred to as decision boundaries.

The goal in a classification problem is to estimate the function f such that the number of misclassifications is minimized.

Figure 11. Plot of a classification fitted to data with binary response.



2 Preliminaries

Mathematical description of some preliminary methods in machine learning is needed which will serve as the foundation of more complicated algorithms that we will derive later.

2.1 Multivariate Linear Regression

Multivariate linear regression is an extension of simple linear regression that is utilized when the dependent variable is a linear combination of multiple independent variables. The model of multivariate linear regression has the following mathematical form:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n + \varepsilon \tag{2.1}$$

where y is the dependent variable, $x_1, x_2, \ldots, x_n$ are the independent variables, $\beta_0, \beta_1, \ldots, \beta_n$ are the parameters of the model, and ε is the error term.

Let's denote the independent variables and the parameters as vectors for simplicity: $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$ and $\boldsymbol{\beta} = [\beta_0, \beta_1, \ldots, \beta_n]^T$. Then, we can rewrite the model in a more compact form:

$$y = \mathbf{x}^T \boldsymbol{\beta} + \varepsilon \tag{2.2}$$

Given a dataset of m examples $\{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(m)}, y^{(m)})\}$, we can construct the design matrix $\mathbf{X} = [\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(m)}]^T$ and the target vector $\mathbf{y} = [y^{(1)}, y^{(2)}, \ldots, y^{(m)}]^T$. Then, the problem of learning the parameters $\boldsymbol{\beta}$ reduces to minimizing the cost function:

$$J(\boldsymbol{\beta}) = \frac{1}{2m}(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \tag{2.3}$$

The solution to this problem can be found by setting the gradient of the cost function with respect to the parameters to zero:

$$\nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}) = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) = 0 \tag{2.4}$$

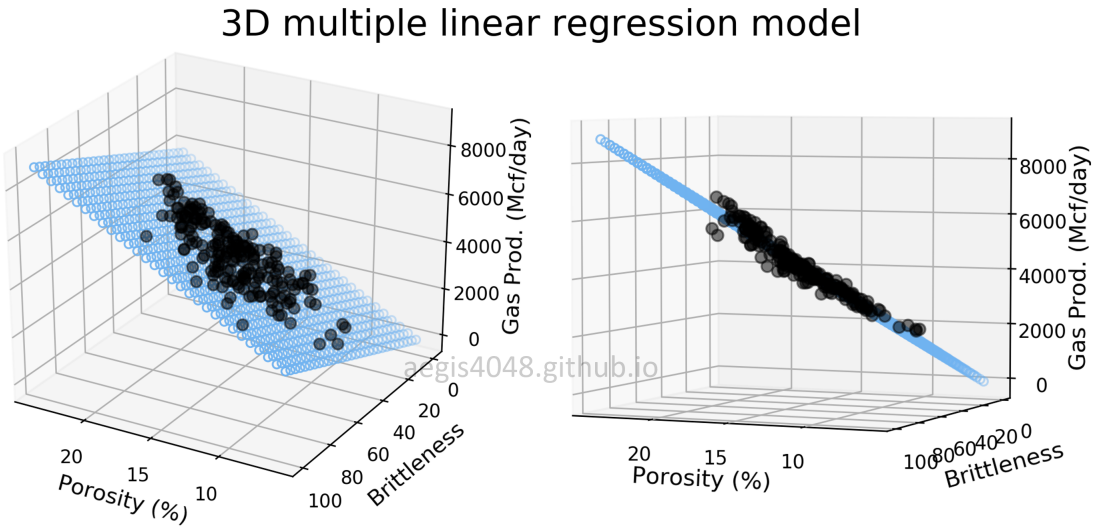
Solving this equation for $\boldsymbol{\beta}$, we get the normal equation:

$$\boldsymbol{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.5}$$

This equation gives the solution for the parameters that minimizes the cost function. Note that it requires the inversion of the matrix $\mathbf{X}^T\mathbf{X}$, which might be computation-

ally expensive or even impossible if the matrix is not invertible. In such cases, gradient descent methods are usually employed to find an approximate solution.

Figure 12. 3D Plot of a multivariate linear regression model fitted to physical parameters of a material. In 3D the fit generates a plane for predicitons.



2.2 Batch Gradient Descent

Gradient Descent is an optimization algorithm that leverages first-order derivatives to iteratively minimize a differentiable function. Its objective is to locate a local minimum of the function. The proposed approach involves iteratively traversing the negative gradient (or an estimated gradient) of the objective function at the present location, as it corresponds to the direction of maximal descent.

The Batch Gradient Descent method refers to the case where the gradient calculated from the cost function involves all training examples. This is a deterministic method as it always computes the same direction for a given position in the parameter space.

Suppose we have a cost function $J(\boldsymbol{\beta})$ which we want to minimize, where $\boldsymbol{\beta}$ is a vector of parameters. The gradient of J at $\boldsymbol{\beta}$, denoted $\nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta})$, is a vector that denotes the direction of maximal ascent of J and its magnitude corresponds to the rate of ascent in that direction.

The update rule for the Batch Gradient Descent method is given by:

$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} - \alpha \nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}_{old}) \tag{2.6}$$

where α is the learning rate, a hyperparameter that controls the step size in the direction of the gradient.

For the multivariate linear regression problem, the cost function is given by:

$$J(\boldsymbol{\beta}) = \frac{1}{2m}(\mathbf{X}\boldsymbol{\beta} - \mathbf{y})^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \tag{2.7}$$

The gradient of the cost function with respect to the parameters $\boldsymbol{\beta}$ is given by:

$$\nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}) = \frac{1}{m}\mathbf{X}^T(\mathbf{X}\boldsymbol{\beta} - \mathbf{y}) \tag{2.8}$$

The gradient vector denotes the direction of maximal increase of the objective function. The Batch Gradient Descent algorithm utilizes this fact to perform the iterative updating of the parameters $\boldsymbol{\beta}$ using:

$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} - \alpha \nabla_{\boldsymbol{\beta}} J(\boldsymbol{\beta}_{old}) \tag{2.9}$$

where α is the learning rate, a hyperparameter that controls the step size in the direction of the gradient.

The Batch Gradient Descent algorithm iterates this update until the algorithm converges, i.e., until the parameters $\boldsymbol{\beta}$ stop changing significantly, or a predefined number of iterations is reached.

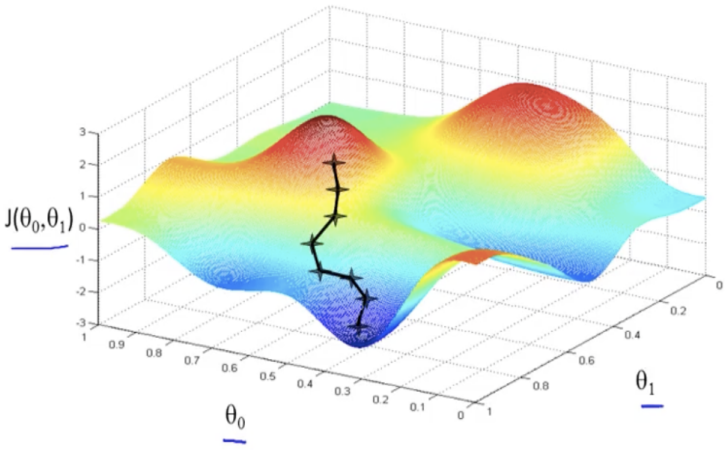
Batch Gradient Descent may incur high computational costs when handling voluminous datasets due to its utilization of all training examples in computing the gradient. In order to compute the gradients, it is necessary to generate predictions for each example in the training set during every iteration. Hence, the computational complexity of this algorithm could render it inefficient when dealing with such datasets. In scenarios where the true gradient is difficult to compute, one may consider utilizing Stochastic Gradient Descent or Mini-Batch Gradient Descent as viable alternatives. These methods involve approximating the true gradient by sampling a subset of the training examples at each iteration.

2.3 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that performs the parameter update for each training example, and hence addresses some of the shortcomings of Batch Gradient Descent for large datasets. The term "stochastic" comes from the fact that the gradient based on a single training example can be thought of as a "noisy" approximation of the true gradient.

For each training example $(\mathbf{x}^{(i)}, y^{(i)})$, we update the parameters $\boldsymbol{\beta}$ according to the rule:

Figure 13. 3D plot of batch gradient descent navigating through the parameters space in search of the minimum.



$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} - \alpha \nabla_{\boldsymbol{\beta}} J_i(\boldsymbol{\beta}_{old}) \tag{2.10}$$

where $J_i(\boldsymbol{\beta}) = \frac{1}{2}(\mathbf{x}^{(i)T}\boldsymbol{\beta} - y^{(i)})^2$ is the cost for the i-th example, and α is the learning rate. The gradient of J_i with respect to $\boldsymbol{\beta}$ is given by:

$$\nabla_{\boldsymbol{\beta}} J_i(\boldsymbol{\beta}) = \mathbf{x}^{(i)}(\mathbf{x}^{(i)T}\boldsymbol{\beta} - y^{(i)}) \tag{2.11}$$

Substituting this into the update rule, we get:

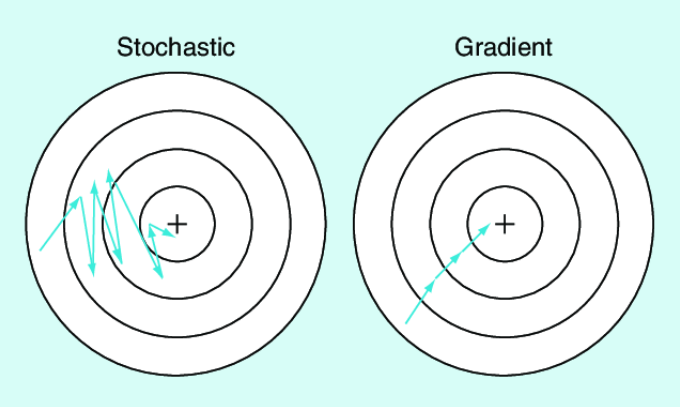
$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} - \alpha \mathbf{x}^{(i)}(\mathbf{x}^{(i)T}\boldsymbol{\beta}_{old} - y^{(i)}) \tag{2.12}$$

This update is performed for all training examples. One pass through all the training examples is called an epoch. We typically run SGD for many epochs, and it is common to shuffle the training examples at the start of each epoch.

SGD has a number of advantages over Batch Gradient Descent. It can make progress after looking at a single example, which can be much faster than Batch Gradient Descent. It can also sometimes escape local minima of the cost function because of the noise in the gradient.

However, this noise can also be a disadvantage as it can cause the parameters to bounce around the optimal value, and hence the algorithm may never truly converge but only fluctuate around the minimum. To mitigate this, we can gradually decrease the learning rate during training, in a process called learning rate annealing. This way, the steps become smaller and smaller, allowing the algorithm to settle at the minimum.

Figure 14. A contour-plot comparison of stochastic and batch gradient descent. See how haphazardly SGD converges to the minimum.



2.4 Mini-Batch Gradient Descent

Mini-Batch Gradient Descent (MBGD) is a variant of the Gradient Descent algorithm that lies between Batch Gradient Descent and Stochastic Gradient Descent. It aims to balance the computational efficiency of Stochastic Gradient Descent while still retaining the deterministic descent direction of Batch Gradient Descent.

MBGD works by dividing the training set into multiple mini-batches of size b. For each mini-batch, we calculate the gradient and update the parameters. This reduces the variance in the parameter updates, which can lead to more stable convergence. It also allows for efficient use of hardware optimizations in matrix operations, as these operations can be parallelized over the examples in a mini-batch.

Formally, let's denote the k-th mini-batch by $\mathbf{X}_k = [\mathbf{x}^{(k1)}, \mathbf{x}^{(k2)}, \ldots, \mathbf{x}^{(kb)}]^T$ and $\mathbf{y}_k = [y^{(k1)}, y^{(k2)}, \ldots, y^{(kb)}]^T$. The cost for the k-th mini-batch is then:

$$J_k(\boldsymbol{\beta}) = \frac{1}{2b}(\mathbf{X}_k\boldsymbol{\beta} - \mathbf{y}_k)^T(\mathbf{X}_k\boldsymbol{\beta} - \mathbf{y}_k) \tag{2.13}$$

The gradient of this cost function with respect to the parameters $\boldsymbol{\beta}$ is:

$$\nabla_{\boldsymbol{\beta}} J_k(\boldsymbol{\beta}) = \frac{1}{b}\mathbf{X}_k^T(\mathbf{X}_k\boldsymbol{\beta} - \mathbf{y}_k) \tag{2.14}$$

And the update rule is:

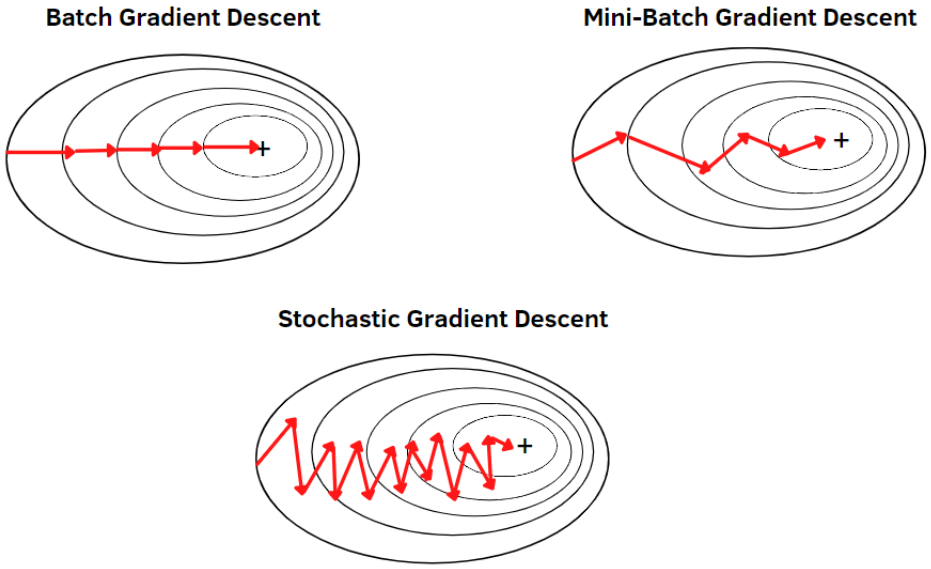
$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} - \alpha\nabla_{\boldsymbol{\beta}} J_k(\boldsymbol{\beta}_{old}) \tag{2.15}$$

where α is the learning rate.

Similar to Stochastic Gradient Descent, we typically run MBGD for many epochs, and it is common to shuffle the training examples at the start of each epoch. Also, learning rate annealing can be used to ensure convergence.

MBGD has been found to work well in practice and is widely used for training deep neural networks. However, the choice of mini-batch size is a hyperparameter of the model and can significantly affect the performance of the algorithm. A small mini-batch size can lead to a noisy gradient estimate and slow convergence, while a large mini-batch size can lead to less efficient use of memory resources and hardware.

Figure 15. A contour-plot comparison of stochastic, batch, and mini-batch gradient descent. Notice how the haphazardness decreases as batch size increases.



2.5 Overfitting and Regularization

Overfitting is when a machine learning model learns too much from the training data, including the noise and irrelevant details, which can cause the model to perform poorly on new data. The model learns concepts from the noise or random fluctuations present in the training data. These concepts are not applicable on new data and harm the model's ability to generalize.

Intuition for Overfitting The intuition behind overfitting is fairly straightforward. When we train a machine learning model, our goal is to make the model perform well on unseen data. We achieve this by optimizing the model parameters to make accurate predictions on the training data. However, if we make our model too complex, such as by adding too many parameters, the model will start to learn not only the underlying structure of the data but also the noise associated with it. This produces a model that performs exceptionally well on training data but poorly on unseen data. This phenomenon is known as overfitting.

Mathematical Justification In mathematical terms, overfitting is related to the balance between bias and variance. A machine learning model's error can be broken down into three parts: bias error, variance error, and irreducible error. Bias error is the error caused by the model's assumptions in the learning algorithm. Variance error is the model's sensitivity to fluctuations in the training set. Irreducible error refers to the error that cannot be minimized by the model and is caused by the presence of noise in the data.

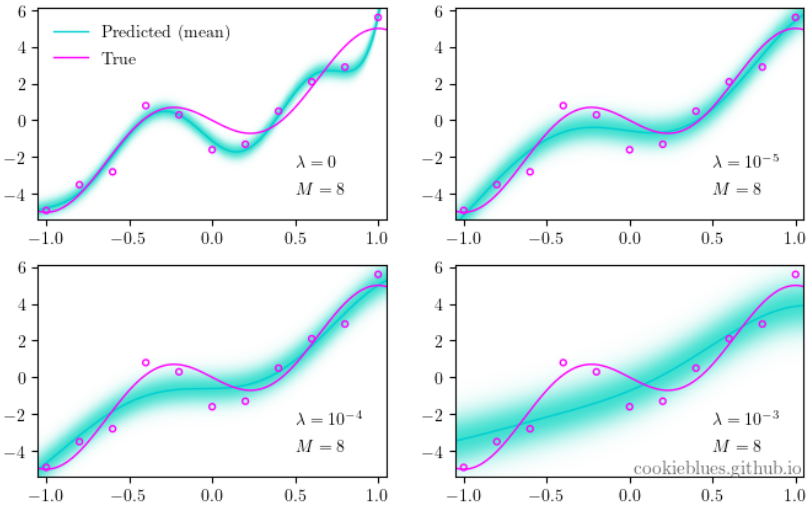
The more complex the model, the higher the bias and variance errors. When the model becomes more complex, the error caused by bias goes down while the error caused by variance goes up. Overfitting happens when the model is too complex, causing the variance error to increase more than the bias error decreases. This leads to a rise in the overall error.

Regularization Regularization prevents overfitting by penalizing complex models through the addition of a penalty term to the loss function. The penalty term helps prevent the learning algorithm from overemphasizing any single feature, which results in a simpler model that is less sensitive to noise in the training data.

Regularization techniques encompass L1 and L2 regularization. L1 regularization, or Lasso regularization, includes a penalty term that corresponds to the absolute value of the coefficients' magnitude. This may lead to models with sparse coefficients, where some features have a coefficient of zero. L2 regularization, or Ridge regularization, penalizes the coefficients by adding a term that is the square of their magnitude. This technique helps to achieve a more uniform distribution of coefficient values among the features.

Overfitting is a frequent issue in machine learning where a model learns the noise in the training data. Regularization helps avoid overfitting by adding a penalty term to the loss function and controlling model complexity.

Figure 16. Plot that show the change in fit as regularization parameter is raised.



2.5.1 Derivation of L1 and L2 Regularization

L1 Regularization (Lasso) L1 regularization, or Lasso regularization, penalizes the loss function by the absolute value of the coefficients. This can be mathematically represented as:

$$L_{L1} = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j|$$

where y_i are the target values, x_{ij} are the feature values, β_j are the model coefficients, n is the number of samples, p is the number of features, and λ is the regularization parameter.

L2 Regularization (Ridge) L2 regularization, or Ridge regularization, includes a penalty term in the loss function that is proportional to the square of the size of the coefficients. This can be mathematically represented as:

$$L_{L2} = \sum_{i=1}^{n}(y_i - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2$$

where the variables are the same as in the L1 regularization.

Effect on Algorithms The effect of L1 and L2 regularization on the learning algorithms can be visualized by considering a simple linear regression problem. Without regularization, the algorithm tries to fit a line that minimizes the sum of squared residuals. With L1 regularization, the algorithm not only tries to minimize the sum of squared residuals, but also the sum of the absolute values of the coefficients. This has the effect of pushing some of the coefficients towards zero, resulting in a sparse model. With L2 regularization, the algorithm tries to minimize the sum of squared residuals and the sum of the squares of the coefficients. This tends to distribute the coefficient values more evenly across features.

2.6 Performance Metrics for Supervised Learning Algorithms

2.6.1 Performance Metrics for Regression Models

In regression analysis, the choice of an evaluation metric hinges heavily on the problem at hand. Certain metrics offer advantages in specific contexts and bear disadvantages in others. We detail three widely used performance metrics for regression models: Mean Squared Error (MSE), Mean Absolute Error (MAE), and Coefficient of Determination (R^2).

Mean Squared Error (MSE) MSE, or Mean Squared Error, is a way to measure the difference between predicted and actual values by taking the average of the squared differences. It is also sometimes referred to as L2 Loss. Mathematically, it's expressed as:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{2.16}$$

where y_i denotes actual values, $\hat{y}_i$ refers to predicted values, and N is the sample size. A zero MSE signifies a perfect fit to the data.

Advantages:

- Easy to compute and differentiable, enabling use in optimization algorithms.
- Emphasizes larger errors over smaller ones due to squaring, useful when larger errors are particularly undesirable.

Disadvantages:

- Highly sensitive to outliers, as the squaring magnifies their effect.
- The output is not in the same unit as the input, making interpretation somewhat more challenging.

Mean Absolute Error (MAE) MAE calculates the mean absolute deviation between predicted and actual values. It's defined as:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{2.17}$$

An MAE of zero implies a perfect prediction.

Advantages:

- Less sensitive to outliers than MSE.
- Output is in the same unit as the input, enhancing interpretability.

Disadvantages:

- Non-differentiable at zero, presenting potential issues in optimization algorithms.
- May under-emphasize large errors if such errors are particularly costly.

Coefficient of Determination (R^2) R^2 measures how much of the variation in the dependent variable is explained by the independent variables in the model. Using the total sum of squares (SST) given by:

$$SST = \sum_{i=1}^{N} (y_i - \bar{y})^2 \tag{2.18}$$

where $\bar{y}$ is the mean of target values, and the residual sum of squares (SSE) given by:

$$SSE = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{2.19}$$

R^2 can be calculated as:

$$R^2 = 1 - \frac{SSE}{SST} \tag{2.20}$$

An R^2 of 1 signifies that the model perfectly predicts the target variable.

Advantages:

- Outputs a normalized score, allowing for comparison between different regression models.
- Provides a measure of how well unseen samples are likely to be predicted by the model.

Disadvantages:

- For non-linear regression models, R^2 can be artificially high as the model can overfit the data.
- R^2 is not capable of detecting bias in coefficient estimates and predictions.

2.6.2 Performance Metrics for Classification Models

For classification problems, there are numerous metrics which are employed to gauge the model's performance. The choice of the metric heavily depends on the problem at hand. We discuss five commonly used performance metrics: Confusion Matrix, Accuracy, Precision, Recall, F1 Score, and AUC-ROC.

Confusion Matrix A confusion matrix is a table that helps us evaluate how well a classification model is performing. It's also called an error matrix. The tool provides a summary of prediction results by showing the actual and predicted class labels side by side. This helps identify different types of prediction errors that a classification model can make.

A confusion matrix for binary classification has four parts: TP, TN, FP, and FN. TP and TN are correct predictions for positive and negative instances. FP (Type I error) is when negative instances are predicted as positive incorrectly, and FN (Type II error) is when positive instances are predicted as negative incorrectly.

$$\begin{bmatrix} \text{TP} & \text{FP} \\ \text{FN} & \text{TN} \end{bmatrix} \tag{2.21}$$

The confusion matrix can further be used to calculate a suite of performance metrics, such as accuracy, precision, recall, and the F1 score, each providing a different perspective on the model's performance. However, it's worth noting that in imbalanced datasets, the confusion matrix might give a misleadingly optimistic view of the model's performance, necessitating the use of other performance metrics as supplements.

Accuracy Accuracy calculates the proportion of all true predictions (both positive and negative) in the overall data. It is given by:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.22}$$

where TP, TN, FP, and FN refer to True Positives, True Negatives, False Positives, and False Negatives, respectively.

Advantages:

- Intuitive and easy to understand.
- Useful when classes are balanced.

Disadvantages:

- Not suitable for imbalanced classes.

Precision and Recall Precision is a metric that measures the accuracy of positive predictions by identifying the proportion of true positives. Recall, also known as Sensitivity, Hit Rate, or True Positive Rate, is the metric that quantifies the percentage of true positives correctly detected. They are defined as:

$$Precision = \frac{TP}{TP + FP} \tag{2.23}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.24}$$

Advantages:

- Useful in imbalanced classes.
- In machine learning, we consider Precision to be good when we want to avoid False Positive. On the other hand, Recall is good when we want to avoid False Negatives.

Disadvantages:

- If the cost of missing a positive example is high, then Precision may be low. On the other hand, if the cost of falsely identifying a negative example as positive is high, then Recall may be low.

F1 Score The F1 Score is a metric that combines Precision and Recall using the harmonic mean to evaluate the performance of a model. This gives a balanced metric to judge the performance of an ML model, particularly on imbalanced datasets:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{2.25}$$

Advantages:

- Balanced measure for binary classification.
- Suitable for imbalanced classes.

Disadvantages:

- Not suitable if you care more about Precision or Recall.

AUC-ROC The Area Under the Receiver Operating Characteristic Curve (AUC-ROC) measures the entire two-dimensional area underneath the ROC curve, which plots the True Positive Rate against the False Positive Rate. AUC-ROC helps to compare classifiers and to choose the best threshold for classification. An AUC-ROC score of 1 signifies a perfect classifier, while 0.5 implies a worthless classifier.

Advantages:

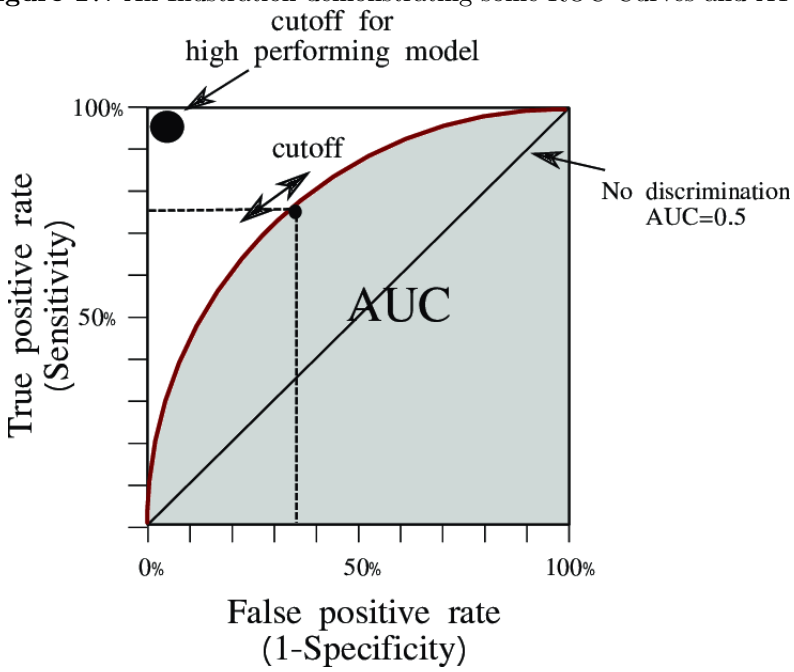
- Provides a single scalar value summarizing the classifier performance over all thresholds.
- Useful when you care equally about positive and negative classes.

Disadvantages:

- Not suitable when you have a strong preference over false positives compared to false negatives, or vice versa.

Each metric possesses its own strengths and weaknesses, and the appropriate metric should be chosen based on the problem's specific needs and constraints.

Figure 17. An Illustration demonstrating some ROC Curves and AYC.



3 Derivation of Machine Learning Methods of Interest

3.1 Logistic Regression

3.1.1 Intuitive Explanation of Logistic Regression

Logistic Regression is a statistical model used in machine learning for classification tasks. It is a simple yet powerful algorithm that can provide probabilistic predictions and interpretability of the input features.

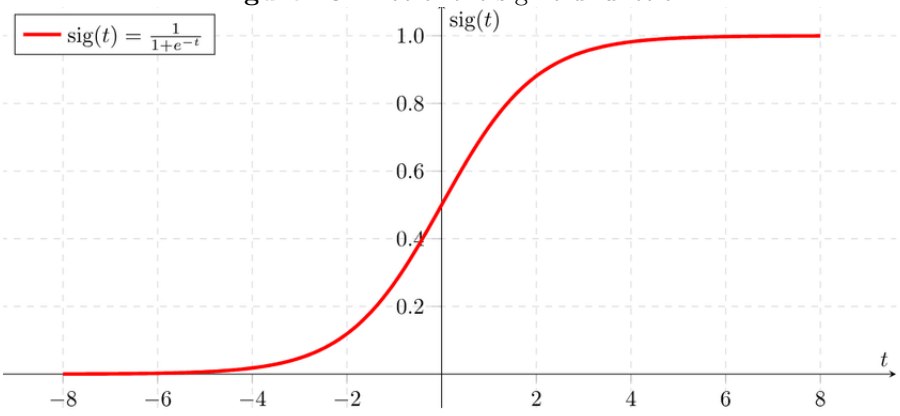
Logistic regression aims to determine the most suitable model that can explain the connection between a binary outcome variable (dependent variable) and a group of predictor variables (independent variables)..

In contrast to linear regression, where the output is a linear function of the input features, logistic regression transforms the output of the linear function using the logistic function, also known as the sigmoid function. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.1}$$

where z is the input to the function. The sigmoid function transforms any real-valued number into the range 0 to 1, which can be treated as a probability for binary classification problems.

Figure 18. Plot of the sigmoid function.



For a binary classification problem with input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ and binary output $y \in \{0, 1\}$, the logistic regression model first computes a linear combination of the input features:

$$z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n \tag{3.2}$$

where $\beta_0, \beta_1, \ldots, \beta_n$ are the parameters of the model. The model then applies the sigmoid function to this linear combination to obtain the probability that the output is 1:

$$p(y = 1|\mathbf{x}; \boldsymbol{\beta}) = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.3}$$

The decision boundary of the logistic regression model is a hyperplane defined by the equation $z = 0$. For an input $\mathbf{x}$ that lies on this hyperplane, the output of the model is 0.5. For $\mathbf{x}$ on one side of the hyperplane, the model predicts a probability greater than 0.5, and for $\mathbf{x}$ on the other side, it predicts a probability less than 0.5.

3.1.2 Mathematical Derivation of Logistic Regression for Binary Classification

The logistic regression model computes the probability that the output is 1 given the input features as follows:

$$p(y = 1|\mathbf{x}; \boldsymbol{\beta}) = \sigma(\boldsymbol{\beta}^T \mathbf{x}) \tag{3.4}$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function, $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$ is the input vector augmented with a constant 1 for the bias term, and $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2, \ldots, \beta_n]^T$ are the parameters of the model.

The probability that the output is 0 given the input features can be computed as the complement of the probability that the output is 1:

$$p(y = 0|\mathbf{x}; \boldsymbol{\beta}) = 1 - \sigma(\boldsymbol{\beta}^T \mathbf{x}) \tag{3.5}$$

We can combine these two equations into a single equation as follows:

$$p(y|\mathbf{x}; \boldsymbol{\beta}) = \sigma(\boldsymbol{\beta}^T \mathbf{x})^y (1 - \sigma(\boldsymbol{\beta}^T \mathbf{x}))^{1-y} \tag{3.6}$$

Given a training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, we can define the likelihood of the parameters as the product of the probabilities of the outputs given the input features:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\beta}) \tag{3.7}$$

The goal of the logistic regression model is to find the parameters that maximize the likelihood of the data. However, it is often more convenient to maximize the log-likelihood:

$$\ell(\boldsymbol{\beta}) = \log L(\boldsymbol{\beta}) = \sum_{i=1}^{m} y^{(i)} \log \sigma(\boldsymbol{\beta}^T \mathbf{x}^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\boldsymbol{\beta}^T \mathbf{x}^{(i)})) \tag{3.8}$$

To maximize the log-likelihood, we can use gradient ascent. The gradient of the log-likelihood with respect to the parameters is given by:

$$\nabla_{\boldsymbol{\beta}} \ell(\boldsymbol{\beta}) = \sum_{i=1}^{m} (y^{(i)} - \sigma(\boldsymbol{\beta}^T \mathbf{x}^{(i)})) \mathbf{x}^{(i)} \tag{3.9}$$

The update rule for gradient ascent is:

$$\boldsymbol{\beta}_{new} = \boldsymbol{\beta}_{old} + \alpha \nabla_{\boldsymbol{\beta}} \ell(\boldsymbol{\beta}_{old}) \tag{3.10}$$

This process is repeated until the parameters $\boldsymbol{\beta}$ converge, typically when the change in the log-likelihood is smaller than a pre-specified threshold or a maximum number of iterations is reached.

Once the model is trained, for a new input vector $\mathbf{x}$, we can predict the output as:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(\boldsymbol{\beta}^T \mathbf{x}) \geq 0.5, \\ 0 & \text{otherwise.} \end{cases} \tag{3.11}$$

This concludes the mathematical derivation of logistic regression. The model is simple and interpretable, yet can provide powerful classification performance given linearly separable classes or when augmented with kernel methods. In practice, regularization terms may be added to the log-likelihood to prevent overfitting.

3.1.3 Generalization of Logistic Regression to Multiple Classes

For binary classification tasks, logistic regression is a suitable model. However, for multi-class classification tasks, we need to generalize logistic regression. The generalized model is known as Multinomial Logistic Regression or Softmax Regression.

Softmax Regression predicts a probability for each class, rather than a single probability. The total probability of all possible outcomes is equal to one, and the class with the highest probability is considered as the predicted class.

Formally, suppose we have K classes and a feature vector $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$. The Softmax Regression model first computes a score for each class k:

$$z_k = \boldsymbol{\beta}_k^T \mathbf{x} \tag{3.12}$$

where $\boldsymbol{\beta}_k = [\beta_{k0}, \beta_{k1}, \beta_{k2}, \ldots, \beta_{kn}]^T$ are the parameters for class k.

The model then applies the softmax function to these scores to obtain the probabilities:

$$p(y = k|\mathbf{x}; \boldsymbol{\beta}) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.13}$$

The decision boundaries between any two classes are the points where their probabilities are equal.

The likelihood of the parameters given the training set $\{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$ is the product of the probabilities of the outputs given the input features:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\beta}) \tag{3.14}$$

Similar to binary logistic regression, we typically maximize the log-likelihood:

$$\ell(\boldsymbol{\beta}) = \log L(\boldsymbol{\beta}) = \sum_{i=1}^{m} \log p(y^{(i)}|\mathbf{x}^{(i)}; \boldsymbol{\beta}) \tag{3.15}$$

The gradient of the log-likelihood with respect to the parameters for class k is given by:

$$\nabla_{\boldsymbol{\beta}_k} \ell(\boldsymbol{\beta}) = \sum_{i=1}^{m} (1\{y^{(i)} = k\} - p(y = k|\mathbf{x}^{(i)}; \boldsymbol{\beta}))\mathbf{x}^{(i)} \tag{3.16}$$

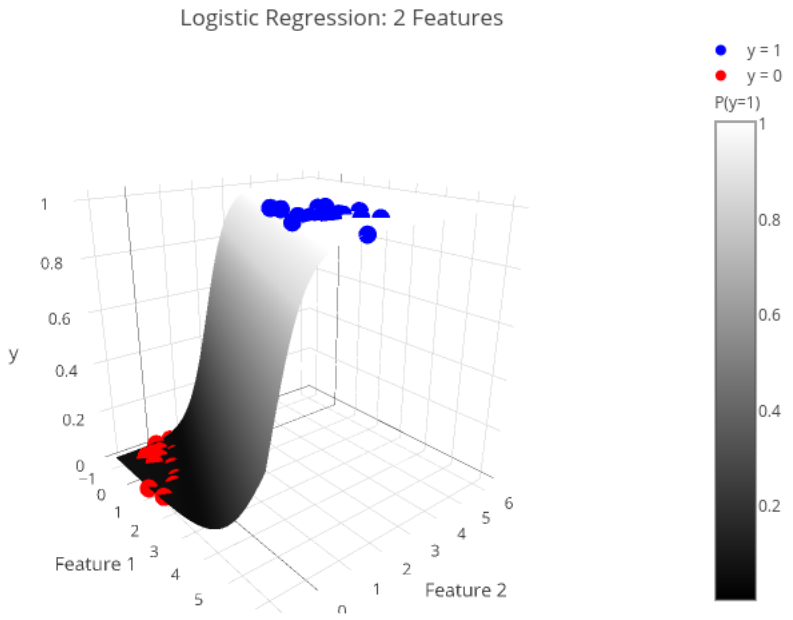
where $1\{\cdot\}$ is the indicator function, which equals 1 if the condition in braces is true and 0 otherwise.

We can use gradient ascent to maximize the log-likelihood, with the update rule:

$$\boldsymbol{\beta}_{k,new} = \boldsymbol{\beta}_{k,old} + \alpha \nabla_{\boldsymbol{\beta}_k} \ell(\boldsymbol{\beta}_{k,old}) \tag{3.17}$$

where α is the learning rate. The algorithm iterates this process until the parameters converge.

Figure 19. 3D plot of a logistic regression model fitted on 2-input categorical data.



3.2 Decision Trees

3.2.1 Intuitive Explanation of Decision Trees

Decision Trees are a Supervised Machine Learning technique that involves splitting data based on a specific parameter.

A decision tree is a flowchart that logically represents decisions and their outcomes. This is a plan with a tree structure that tests a set of attributes. Each internal node represents an attribute. The class label is represented by each leaf node, which results in a straightforward and easy-to-understand method.

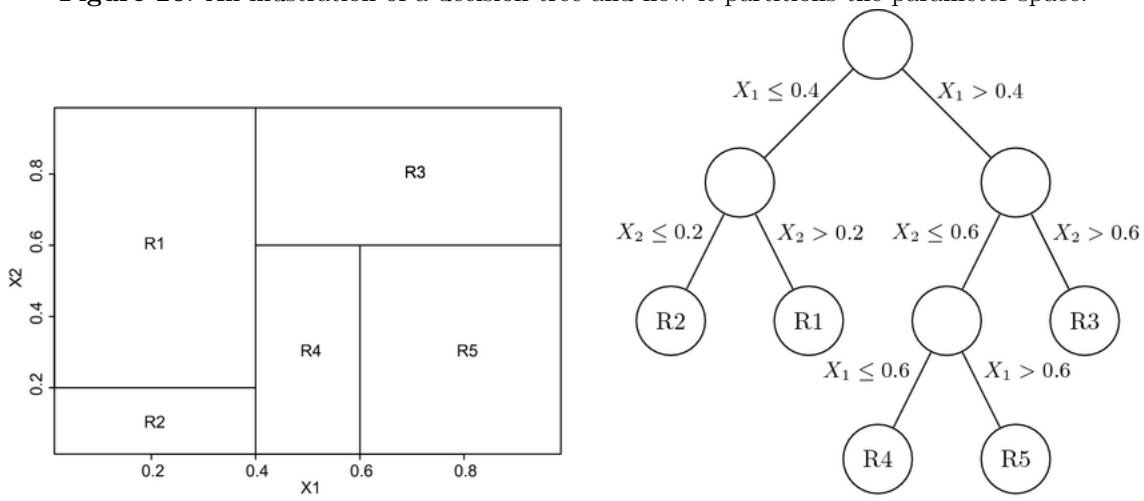
A decision tree begins with a root node that splits into various branches representing potential outcomes. Each outcome generates new nodes that can branch into other possibilities. The structure has a tree-like form, facilitating the visualization of the sequence of choices and results that culminate in the ultimate decision.

To understand why decision trees work, let's consider a simple analogy. Suppose you are playing a game of Twenty Questions. Your friend thinks of a person, and you try to guess who it is by asking yes/no questions. Good questions narrow down the possibilities significantly. For instance, asking "Is the person alive?" cuts down the possibilities almost in half. On the other hand, asking "Is the person's birthday in July?" would eliminate fewer possibilities. After asking enough well-chosen questions, you can often guess who the person is.

Decision trees work in a similar way when making predictions. They ask a series of yes/no questions about the input features until they arrive at a good guess for the target variable. At each node in the tree, the question is chosen to gain the most information based on the answers to the previous questions.

In essence, decision trees manage to split up the complex process of prediction into simpler, smaller decisions, thereby making the whole task manageable. Each decision at each node is fairly simple, but when combined together, the decision tree can accurately model complex relationships between input features and the target variable.

Figure 20. An illustration of a decision tree and how it partitions the parameter space.



3.2.2 Construction of Decision Trees for Binary Classification

Building a decision tree involves a series of steps. The main procedure of building a decision tree can be broken down into the following steps:

1. Utilize Attribute Selection Measures (ASM) to determine the optimal attribute for dividing the data.

2. Transform the attribute into a node for decision-making and divide the dataset into more manageable subsets.

3. Initiate the construction of a tree by iteratively applying this procedure to every offspring until one of the criteria is satisfied:

    - All tuples share a common attribute value.
    - All attributes have been exhausted.
    - The dataset is depleted.

The ASM assigns a score to each attribute based on its relevance to the dataset. The attribute with the highest score will be chosen as the attribute for splitting. The

prevalent evaluation metrics are Information Gain, Gain Ratio, and Gini Index.

Information Gain The concept of information gain quantifies the amount of information that a feature provides regarding the class. The underlying principle is rooted in entropy, a metric for gauging impurity, uncertainty, or disorder. Information gain is the reduction of uncertainty. Information gain measures the change in uncertainty before and after splitting a dataset based on a given attribute.

Let's denote the entropy of a dataset S relative to a binary classification with classes C_1 and C_2 as follows:

$$E(S) = -p(C_1)\log_2 p(C_1) - p(C_2)\log_2 p(C_2) \tag{3.18}$$

where $p(C_1)$ and $p(C_2)$ are the proportions of instances of classes C_1 and C_2 in S, respectively.

The information gain of an attribute A with respect to S is defined as:

$$IG(S, A) = E(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} E(S_v) \tag{3.19}$$

where Values(A) is the set of all possible values of attribute A, S_v is the subset of S for which attribute A has value v, and $|S_v|$ and $|S|$ are the numbers of instances in S_v and S, respectively.

Gini Index The Gini Index, or Gini impurity, measures the probability of a feature being misclassified when chosen randomly. A set of elements is considered pure if they are all associated with the same class.

The Gini Index of a dataset S relative to a binary classification with classes C_1 and C_2 is defined as follows:

$$G(S) = 1 - p(C_1)^2 - p(C_2)^2 \tag{3.20}$$

The Gini Index of an attribute A with respect to S is defined as:

$$G(S, A) = \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} G(S_v) \tag{3.21}$$

The attribute with the smallest Gini Index is chosen to split the data at each node.

Once the tree is constructed, it is necessary to prune it to prevent overfitting. Overfitting happens when the decision tree model is excessively intricate and captures irrelevant information in the data. Pruning reduces decision tree size and improves generalization to new data.

Various techniques exist for pruning, including reduced error pruning and cost complexity pruning. All these methods share the common goal of eliminating low-power branches for instance classification.

Reduced error pruning involves replacing each node with its most popular class, starting from the leaves. If the alteration does not impact some performance metrics of the model, it is retained.

In cost complexity pruning, we define a measure of the complexity of a tree as the number of its leaves, and we denote it by $|T|$. The cost complexity of a tree is defined as:

$$R_\alpha(T) = E(T) + \alpha|T| \tag{3.22}$$

where $E(T)$ is the misclassification error rate of the tree T and α is a parameter that controls the trade-off between the tree's accuracy and complexity.

The idea of cost complexity pruning is to find the tree that minimizes $R_\alpha(T)$.

After building and pruning the tree, we can use it to make predictions on new instances. Starting from the root node, for each attribute in the instance, we follow the corresponding branch until reaching a leaf node. The predicted class is the class label associated with the leaf node.

3.2.3 Generalizing Decision Trees to Multiple Classes

In the binary classification scenario, we measure the impurity of a node in the decision tree using entropy or Gini index, which are defined with respect to a binary classification. However, both entropy and Gini index can be generalized to handle multiple classes.

Let's denote $C_1, C_2, \ldots, C_K$ as the K classes in the multi-class classification problem. The proportions of instances of classes $C_1, C_2, \ldots, C_K$ in a subset S of the training set are denoted as $p(C_1), p(C_2), \ldots, p(C_K)$, respectively.

Entropy The entropy of the subset S is defined as:

$$E(S) = -\sum_{k=1}^{K} p(C_k) \log_2 p(C_k) \tag{3.23}$$

The information gain of an attribute A with respect to S is defined as before:

$$IG(S, A) = E(S) - \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} E(S_v) \tag{3.24}$$

The attribute with the highest information gain is chosen to split the data at each node.

Gini Index The Gini index of the subset S is defined as:

$$G(S) = 1 - \sum_{k=1}^{K} p(C_k)^2 \tag{3.25}$$

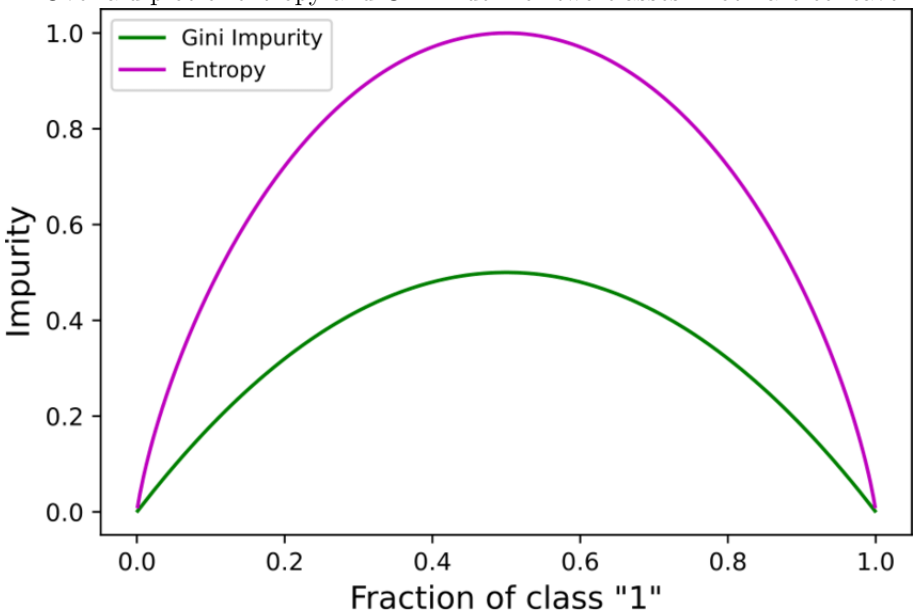
The Gini index of an attribute A with respect to S is defined as before:

$$G(S, A) = \sum_{v \in \text{Values}(A)} \frac{|S_v|}{|S|} G(S_v) \tag{3.26}$$

The attribute with the smallest Gini index is chosen to split the data at each node.

As we can see, the generalization of decision trees to multiple classes does not require a change in the overall structure or logic of the decision tree algorithm. It merely involves replacing the binary-class impurity measures with their multi-class counterparts.

Figure 21. Overlaid plot of entropy and Gini-index for two classes. Both are concave functions.



3.2.4 Construction of Decision Trees for Regression

A decision tree for regression is very similar to that for classification, the major difference being in the way the responses are modeled. While in a classification setting the tree predicts a class label, in a regression setting, the tree predicts a continuous value.

In the context of a decision tree for regression, we wish to make predictions based on a set of decision rules derived from the data. The tree is constructed by binary recursive partitioning of the feature space, such that at each node of the tree, the feature and the split point are chosen to minimize a certain cost function. For regression trees, this cost function is typically the residual sum of squares (RSS):

$$\text{RSS} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{3.27}$$

where y_i is the observed response, $\hat{y}_i$ is the predicted response, and N is the number of observations.

The algorithm starts at the top of the tree. To find the best binary split at this top node, for every feature j and every value s, we partition the data into two subsets:

$$R_1(j, s) = \{X | X_j \leq s\} \text{ and } R_2(j, s) = \{X | X_j > s\} \tag{3.28}$$

We then compute the pair (j, s) that minimizes the cost function:

$$\min_{j,s} \left[\min_{c_1} \sum_{x_i \in R_1(j,s)} (y_i - c_1)^2 + \min_{c_2} \sum_{x_i \in R_2(j,s)} (y_i - c_2)^2 \right] \tag{3.29}$$

where the inner minimizations are solved by $\hat{c}_1 = \text{avg}(y_i | x_i \in R_1(j, s))$ and $\hat{c}_2 = \text{avg}(y_i | x_i \in R_2(j, s))$, i.e., the mean response of the observations in each region.

After finding the best feature and split point at the top node, we repeat this process in a recursive manner for each of the two resulting regions, and then for each of the four resulting regions after that, and so on. This is continued until some stopping criterion is met, such as a maximum tree depth or a minimum node size.

In regression problems, predictions for a new instance are made by traversing the decision tree based on the instance's feature values, and then returning the mean response of the training observations in the corresponding leaf node.

Finally, just as with classification trees, the decision tree can be pruned to achieve a compromise between its complexity and its fit to the training data, using techniques such

as cost-complexity pruning (also known as weakest link pruning).

3.3 Neural Networks

3.3.1 Intuitive Explanation of Neural Networks

Neural networks, or artificial neural networks, are computer systems modeled after the neural networks found in animal brains. Machine learning systems acquire task-solving abilities by analyzing examples, typically without the need for programming tailored to the specific task.

A neural network receives inputs, which are processed in hidden layers using adjustable weights during training. The model generates a prediction as its output.

To comprehend the functioning of neural networks, it is essential to comprehend the basic computational unit of a neural network, which is the neuron or node.

A perceptron is a simplified model of a biological neuron. The process involves taking numerical inputs, multiplying them with assigned weights, summing them up, and then applying an activation function to generate an output.

This model is a simplification of a biological neuron because neurons in the brain exhibit binary firing behavior. The activation function of our model neuron emulates this characteristic by producing a value that sharply rises when the total input surpasses a specific threshold.

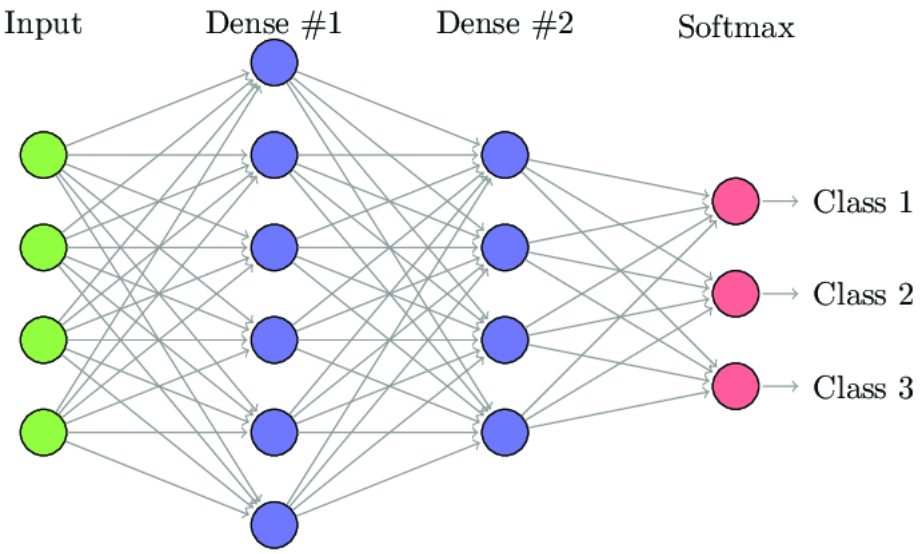
An artificial neural network comprises layers of nodes or neurons. The input layer is responsible for receiving data into the neural network. Each node in the input layer corresponds to a dimension of the data. The output layer is responsible for generating the prediction or result of the neural network. The neural network has hidden layers, which perform intermediate computations between the input and output layers.

The efficacy of neural networks is rooted in the training phase, where the neuron weights are modified to enhance predictive accuracy. The process of minimizing prediction error in a neural network is typically achieved through backpropagation, an optimization algorithm that propagates error backwards through the network and adjusts weights to minimize the error.

A key intuition for understanding why neural networks work is their ability to represent and learn hierarchies of features. In the lower layers of the network, neurons learn to represent simple features of the input data. In the subsequent layers, neurons combine the features learned by the previous layer into more complex features. This hierarchical learning process allows neural networks to learn complex patterns and make accurate predictions.

In essence, neural networks are function approximators. They can learn to approximate an unknown function given enough training data and compute resources. This makes

Figure 22. A simplified diagram of a neural network used for classification.



them incredibly powerful tools for tasks where the exact rules are difficult to pin down with traditional programming, such as image recognition, natural language processing, and of course many problems in physics.

3.3.2 Derivation of Neural Networks for Regression

The Structure of Neural Network Typically, a multilayer neural network comprises an input layer, one or more hidden layers, and an output layer. Each layer consists of numerous nodes, or neurons, and each node in one layer is connected to each node in the following layer. Weights are associated with these connections.

Let's denote the number of nodes in layer l as $d^{[l]}$, the input to node i in layer $l+1$ as $z_i^{[l+1]}$, the output of node i in layer l as $a_i^{[l]}$, and the weight of the connection from node j in layer l to node i in layer $l+1$ as $w_{ij}^{[l]}$. The bias of node i in layer $l+1$ is denoted as $b_i^{[l+1]}$.

Forward Propagation The forward propagation process starts from the input layer and ends at the output layer. The input to each node is a weighted sum of the outputs of all nodes in the previous layer, plus a bias term. The output of each node is the result of applying an activation function to its input.

Mathematically, this can be described as:

$$z_i^{[l+1]} = \sum_{j=1}^{d^{[l]}} w_{ij}^{[l]} a_j^{[l]} + b_i^{[l+1]} \tag{3.30}$$

$$a_i^{[l+1]} = g^{[l+1]}(z_i^{[l+1]}) \tag{3.31}$$

where $g^{[l+1]}$ is the activation function for layer $l+1$. Common choices of activation functions include the sigmoid function, the hyperbolic tangent function, and the rectified linear unit (ReLU) function.

The activation function of the output layer is determined by the type of regression task. In regression tasks with continuous output, the activation function is usually the identity function. When dealing with tasks that have exclusively positive outputs, one option for the activation function is the exponential function.

Loss Function In a regression task, the typical loss function is the mean squared error (MSE) between the predicted output and the true output. If we denote the true output as y, and the predicted output as $\hat{y}$, then the MSE is defined as:

$$J(w, b) = \frac{1}{2}(y - \hat{y})^2 \tag{3.32}$$

Here, the factor of $1/2$ is introduced for mathematical convenience as it simplifies the derivative of the function.

Backpropagation Backpropagation is an algorithm to update the weights and biases in order to minimize the loss function. It involves two main steps:

1. Compute the gradient of the loss function with respect to the weights and biases. 2. Update the weights and biases in the direction of the negative gradient.

The gradient of the loss function with respect to the weights and biases can be computed using the chain rule of calculus. Let's denote the gradient of the loss function with respect to $z^{[l]}$ as $\delta_i^{[l]}$. Using the chain rule, we have:

$$\delta_i^{[l]} = \frac{\partial J(w, b)}{\partial z_i^{[l]}} = \frac{\partial J(w, b)}{\partial a_i^{[l]}} \cdot \frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} \tag{3.33}$$

The term $\frac{\partial J(w,b)}{\partial a_i^{[l]}}$ is the derivative of the loss function with respect to $a_i^{[l]}$, and the term $\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}}$ is the derivative of the activation function with respect to its input.

The derivative of the loss function with respect to $a_i^{[l]}$ depends on the gradients in the next layer:

$$\frac{\partial J(w,b)}{\partial a_i^{[l]}} = \sum_{k=1}^{d^{[l+1]}} \frac{\partial J(w,b)}{\partial z_k^{[l+1]}} \cdot \frac{\partial z_k^{[l+1]}}{\partial a_i^{[l]}} = \sum_{k=1}^{d^{[l+1]}} \delta_k^{[l+1]} w_{ki}^{[l]} \tag{3.34}$$

The derivative of the activation function with respect to its input is just the derivative of the activation function:

$$\frac{\partial a_i^{[l]}}{\partial z_i^{[l]}} = g^{[l]\prime}(z_i^{[l]}) \tag{3.35}$$

Now we have everything we need to compute the gradients:

$$\delta_i^{[l]} = \left(\sum_{k=1}^{d^{[l+1]}} \delta_k^{[l+1]} w_{ki}^{[l]} \right) g^{[l]\prime}(z_i^{[l]}) \tag{3.36}$$

The gradients of the loss function with respect to the weights and biases are then given by:

$$\frac{\partial J(w,b)}{\partial w_{ij}^{[l]}} = \delta_i^{[l+1]} a_j^{[l]} \tag{3.37}$$

$$\frac{\partial J(w,b)}{\partial b_i^{[l+1]}} = \delta_i^{[l+1]} \tag{3.38}$$

After computing the gradients, we update the weights and biases using gradient descent:

$$w_{ij}^{[l]} = w_{ij}^{[l]} - \alpha \frac{\partial J(w,b)}{\partial w_{ij}^{[l]}} \tag{3.39}$$

$$b_i^{[l+1]} = b_i^{[l+1]} - \alpha \frac{\partial J(w,b)}{\partial b_i^{[l+1]}} \tag{3.40}$$

Here, α is the learning rate, a hyperparameter that determines the step size in the direction of the negative gradient. This completes the description of how a multi-layer neural network for regression is derived.

3.3.3 Derivation of Neural Networks for Classification

The Structure of Neural Network Similar to regression, a multilayer neural network for classification comprises an input layer, one or more hidden layers, and an output layer. The output layer's neuron count usually matches the classification task's class count.

Forward Propagation The forward propagation process remains the same as in the regression case.

Activation Function in the Output Layer The softmax activation function is utilized in the output layer for multi-class classification. The function generates a probability distribution for the classes, indicating the network's level of certainty regarding the input's classification.

The softmax function is defined as:

$$a_i^{[L]} = \frac{e^{z_i^{[L]}}}{\sum_{j=1}^{K} e^{z_j^{[L]}}} \tag{3.41}$$

where L denotes the output layer, and K is the number of classes.

Loss Function In classification, we often use the cross-entropy loss function to calculate the difference between the actual distribution of the data and the predicted distribution. The cross-entropy loss is a mathematical function that measures the difference between two probability distributions. It is defined as:

$$J(w, b) = -\sum_{i=1}^{K} y_i \log \hat{y}_i \tag{3.42}$$

Backpropagation Backpropagation in a classification neural network follows a similar principle to the one in a regression network. The difference lies in the computation of the error term $\delta_i^{[l]}$ in the output layer:

For the output layer:

$$\delta_i^{[L]} = a_i^{[L]} - y_i \tag{3.43}$$

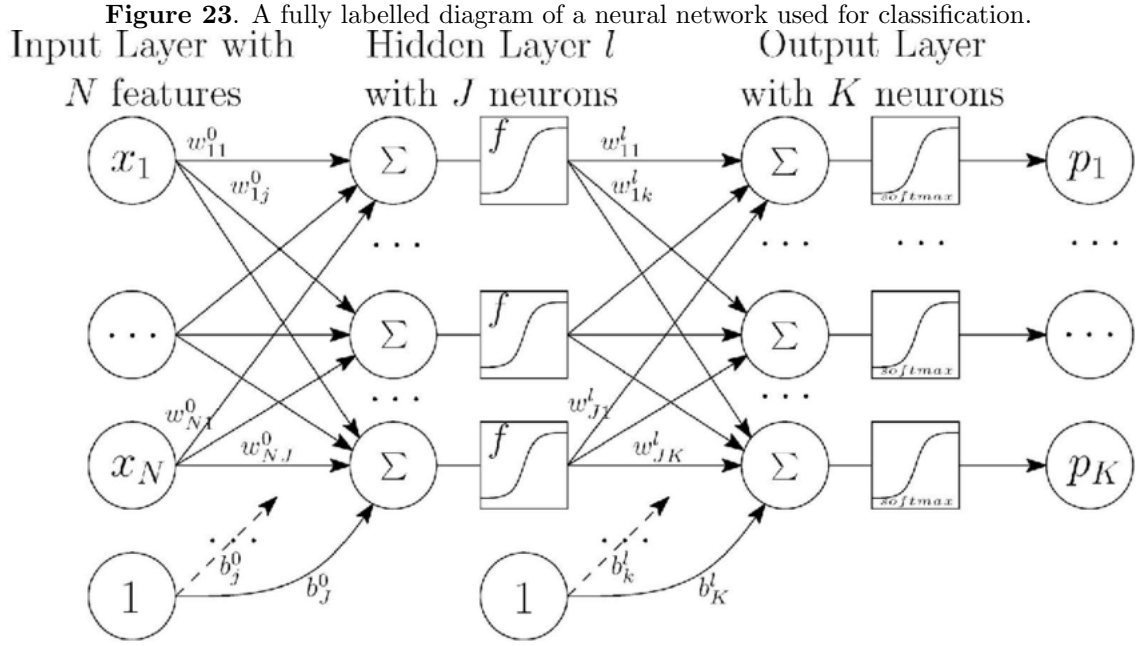
For hidden layers, the computation remains the same:

$$\delta_i^{[l]} = \left(\sum_{k=1}^{d^{[l+1]}} \delta_k^{[l+1]} w_{ki}^{[l]} \right) g^{[l]\prime}(z_i^{[l]}) \tag{3.44}$$

Weights and biases are updated using the computed gradients just like in the regression case.

This concludes the derivation of a multi-layer neural network for classification. As we see, the main difference from the regression case lies in the choice of the activation function in the output layer and the loss function, which are chosen to reflect the probabilistic nature of the classification task.

Figure 23. A fully labelled diagram of a neural network used for classification.



3.3.4 Activation Functions in Neural Networks

In artificial neural networks, activation functions are mathematical equations that determine the output of a neuron or node. They are a crucial part of the learning process as they introduce non-linearity into the model, allowing it to learn from the error, and consequently improve the prediction or classification results.

- **Sigmoid:** The sigmoid activation function is defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.45}$$

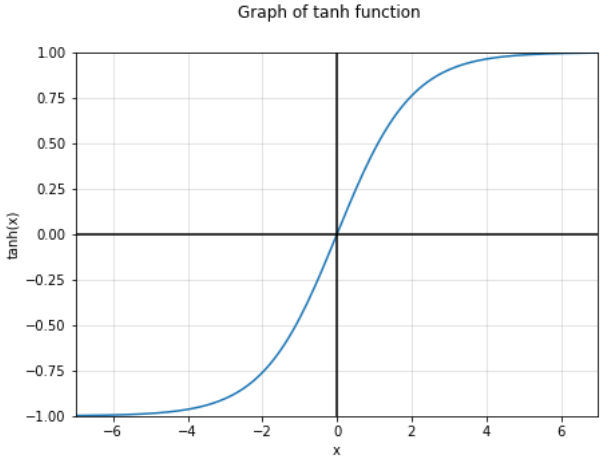
  It outputs a value between 0 and 1, thus it is used mainly in the output layer of binary classification problems. Its main advantage is that it's able to convert numbers to probabilities. However, it suffers from the vanishing gradient problem, where gradients are very close to zero. This results in slow convergence, thus slowing down the learning speed.

- **Tanh:** The tanh (hyperbolic tangent) activation function is defined as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.46}$$

The output range is limited to -1 and 1. The tanh function has the property of strongly mapping negative inputs to negative values and mapping inputs close to zero to values near zero on its graph. Similar to the sigmoid function, this function also encounters the issue of the gradient vanishing.

Figure 24. A plot of the *tanh* function.



- **ReLU:** The Rectified Linear Unit (ReLU) activation function is defined as:

$$f(x) = max(0, x) \tag{3.47}$$

The function implements a thresholding operation, where any negative input is mapped to 0 and any non-negative input is preserved. ReLU is a computationally efficient activation function that addresses the vanishing gradient problem. The model encounters the issue of "dying ReLU" when a substantial gradient leads to weight updates that prevent a neuron from activating on any datapoint in the future.

- **Leaky ReLU:** This function is an improved version of the ReLU function, and is defined as:

$$f(x) = max(0.01x, x) \tag{3.48}$$

It solves the "dying ReLU" problem as it allows small negative values when the input is less than zero.

- **Softmax:** Softmax is a function frequently applied in the output layer of multi-classification tasks. The function is an extension of the sigmoid function. It maps

Figure 25. A plot of the *ReLU* function.

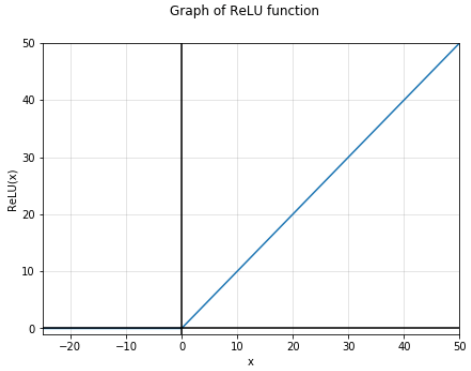
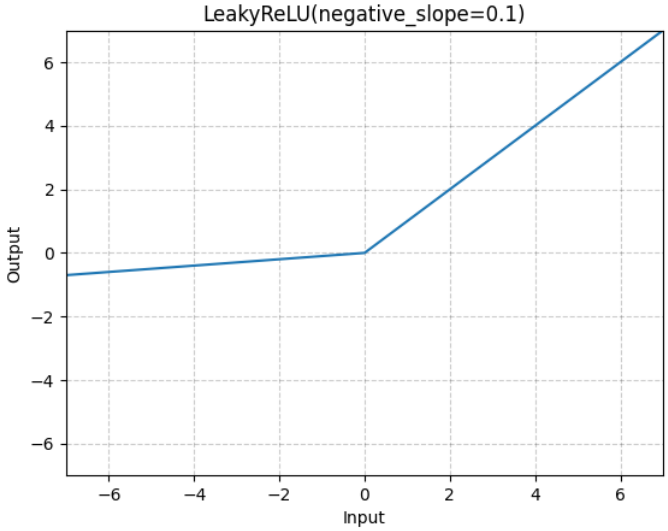


Figure 26. A plot of the Leaky *ReLU* function.



a K-dimensional vector of real numbers to a K-dimensional vector of real numbers. The output vector's entries are between 0 and 1, and their sum is 1.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{i=1}^{K} e^{x_i}} \text{ for } j = 1, \ldots, K \tag{3.49}$$

Choosing the right activation function depends on the specific requirements of the model and the nature of the data. It's a crucial decision that significantly influences the performance of a neural network model.

4 Classification of air showers measured with the MAGIC telescope using logistic regression

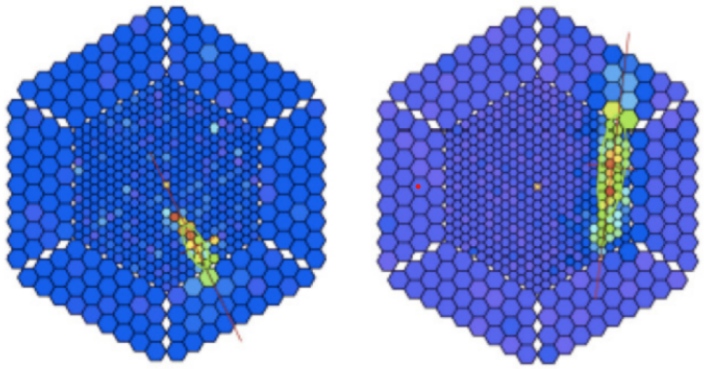
4.1 Background of Air Showers and the MAGIC Telescope

Air showers based on Cherenkov radiation are a significant research topic in the field of high-energy astrophysics. When energetic cosmic or gamma rays reach Earth's atmosphere, they interact with air particles and create a chain reaction of secondary particles, which is known as an "air shower". Secondary particles exceeding the speed of light in air produce Cherenkov radiation, a brief light emission lasting a few nanoseconds. Studying air showers helps us understand primary cosmic rays and gain knowledge about the universe's most powerful phenomena.

The Magic Telescope is a crucial tool for conducting these observations. The MAGIC telescope, stationed at Roque de los Muchachos Observatory in La Palma, Canary Islands, specializes in the detection of gamma rays. The system has two IACTs with a diameter of 17 meters each. These IACTs gather Cherenkov light produced by air showers caused by gamma rays.

A significant obstacle in air shower analysis with IACTs, such as MAGIC, is the differentiation between gamma-ray-induced showers and those generated by cosmic rays. Cosmic rays, which are mostly charged particles like protons and atomic nuclei, get redirected by magnetic fields in space before reaching Earth from various angles. Gamma rays are uncharged and exhibit source directionality. Detecting gamma-ray-induced air showers is crucial for astrophysics research. This task is challenging because there are many more air showers caused by cosmic rays than by gamma rays.

Figure 27. Detector plot for the MAGIC telescope for Gamma and Hadronic showers.



Gamma shower Hadronic shower

A potential solution to this challenge lies in machine learning, specifically logistic regression. The classification task of distinguishing between gamma-ray and cosmic-ray-

induced air showers can be treated as a binary classification problem, suitable for a logistic regression approach. The input features for this problem can be derived from various properties of the air showers as measured by the MAGIC telescope, such as the arrival direction, shower size, and shape of the shower image.

4.2 Traditional Methods for Air Shower Classification and their Limitations

Historically, several techniques have been used to discriminate between gamma-ray and cosmic-ray-induced air showers. These methods generally rely on distinct properties or patterns observed in the air shower data, and often involve detailed and intricate statistical analysis.

1. **Hillas Parameters Analysis**: Named after A. M. Hillas, Hillas parameters are geometrical and statistical properties of the shower image captured by the Cherenkov telescope. These include parameters like width, length, size, and orientation of the shower image. Hillas deduced that gamma-ray showers tend to be more symmetric and compact, while cosmic ray showers are more diffuse and irregular. A cut-based analysis can be performed on these parameters to distinguish between gamma-ray and cosmic-ray showers.

   The Hillas parameters can be described mathematically as follows. Given a shower image $I(x, y)$ where (x, y) are the coordinates of the pixels in the image, the image centroid $(\bar{x}, \bar{y})$, length L, width W, and orientation ϕ are given by:

$$
\begin{aligned}
\bar{x} &= \frac{\sum xI(x,y)}{\sum I(x,y)}, \\
\bar{y} &= \frac{\sum yI(x,y)}{\sum I(x,y)}, \\
L &= \sqrt{2((\sigma_x^2 - \sigma_y^2) + 4\sigma_{xy}^2)}, \\
W &= \sqrt{2((\sigma_x^2 - \sigma_y^2) - 4\sigma_{xy}^2)}, \\
\phi &= \frac{1}{2} \arctan\left(\frac{2\sigma_{xy}}{\sigma_x^2 - \sigma_y^2}\right),
\end{aligned}
$$

   where $\sigma_x^2 = \frac{\sum (x-\bar{x})^2 I(x,y)}{\sum I(x,y)}$, $\sigma_y^2 = \frac{\sum (y-\bar{y})^2 I(x,y)}{\sum I(x,y)}$, and $\sigma_{xy} = \frac{\sum (x-\bar{x})(y-\bar{y}) I(x,y)}{\sum I(x,y)}$.

   However, the cut-based approach has limitations as it does not fully exploit the correlations between different parameters, and the cut thresholds need to be manually defined and adjusted, which can be suboptimal and time-consuming.

2. **Disp Parameter Analysis**: Another method involves the use of the DISP param-

eter, which measures the displacement of the shower image centroid from the source direction. While this method improves upon the Hillas analysis by considering additional image features, it still suffers from similar limitations as the Hillas method, primarily in its inability to efficiently capture the correlations between parameters and the need for manually defining cut thresholds.

Given these limitations, an automated, data-driven approach like logistic regression, which can simultaneously consider all parameters and their correlations, offers a promising alternative. Furthermore, logistic regression can automatically learn the optimal decision boundary for classification, alleviating the need for manual threshold adjustment.

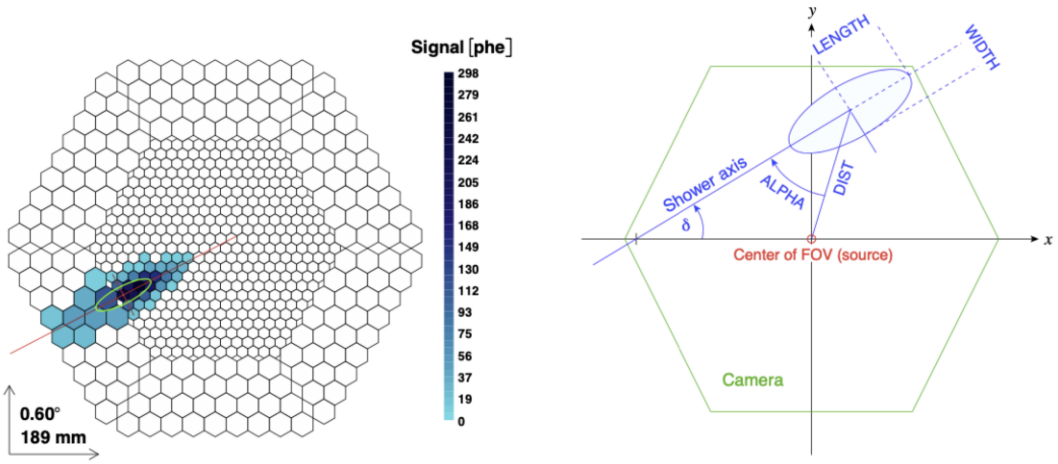
4.3 Applying Logistic Regression to Classify Air Showers

To overcome the limitations of the traditional methods for classifying air showers, we turn to logistic regression, a type of supervised learning method that has been widely used for binary classification problems. This method can leverage all the available features simultaneously and can automatically learn the optimal decision boundary without the need for manual threshold adjustment.

We use the MAGIC Gamma Telescope dataset hosted by the UCI Machine Learning Repository, which consists of 19020 instances of simulated air shower events (12332 gamma events and 6688 hadron events). The dataset includes ten features extracted from images of air showers, which are described in the list below.

1. **fLength**: Major axis of ellipse [mm]
2. **fWidth**: Minor axis of ellipse [mm]
3. **fSize**: 10-log of sum of content of all pixels [in #phot]
4. **fConc**: Ratio of sum of two highest pixels over fSize [ratio]
5. **fConc1**: Ratio of highest pixel over fSize [ratio]
6. **fAsym**: Distance from highest pixel to center, projected onto major axis [mm]
7. **fM3Long**: 3rd root of third moment along major axis [mm]
8. **fM3Trans**: 3rd root of third moment along minor axis [mm]
9. **fAlpha**: Angle of major axis with vector to origin [deg]
10. **fDist**: Distance from origin to center of ellipse [mm]

Figure 28. An illustration showing how some features are calculated from the detector.



Firstly, the data is loaded and the class labels ('g' for gamma and 'h' for hadron) are mapped to binary values (1 and 0, respectively) for the purpose of binary classification.

```
import pandas as pd

filename = "https://www.physi.uni-heidelberg.de/~reygers/lectures/2021/ml/
    data/magic04_data.txt"
df = pd.read_csv(filename, engine='python')

# use categories 1 and 0 insted of "g" and "h"
df['class'] = df['class'].map({'g': 1, 'h': 0})
```

We then create plots of each feature for gamma and hadron events to visualize their distributions and differences.

```
import matplotlib.pyplot as plt

df0 = df[df['class'] == 0] # hadron data set
df1 = df[df['class'] == 1] # gamma data set

for col in df0.columns:
    plt.figure()
    plt.hist(df0.loc[:, col], alpha = 0.5, label = 'Hadron')
    plt.hist(df1.loc[:, col], alpha = 0.5, label = 'Gamma')
    plt.xlabel(col)
    plt.ylabel('frequency')
    plt.legend()
```

We then split our data into a training set and a test set, with the test set constituting 50% of the total data.

```
from sklearn.model_selection import train_test_split

y = df['class'].values
X = df[[col for col in df.columns if col!="class"]]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
    random_state=42)
```

We define a logistic regression model and fit it to our training data.

```
from sklearn import linear_model

logreg = linear_model.LogisticRegression(max_iter = 1000)

logreg.fit(X_train, y_train)
```

We then evaluate the accuracy and AUC score of our logistic regression model.

```
from sklearn.metrics import accuracy_score, roc_auc_score

y_pred = logreg.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
auc_score = roc_auc_score(y_test, y_pred)

print("Accuracy: ", accuracy)
print("AUC score: ", auc_score)
```

Finally, we plot the ROC curve, which provides a comprehensive view of the model's performance across all classification thresholds.

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
%matplotlib inline

y_pred_prob = logreg.predict_proba(X_test)[:, 1] # predicted probabilities
    for class 1
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)

plt.figure()
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve')
plt.show()
```

By applying logistic regression to this task, we are able to achieve efficient, automated classification of air showers, outperforming traditional methods in terms of both accuracy and convenience. This initial model achieves 78.9% accuracy and a AUC value of 0.742, which are decent for a simple logistic model with no improvements.

Also, notice how simple it is to train and apply a logistic regression model. Although we discussed the mathematical detail of a logistic regression model, we do not need

to implement it from scratch since extremely efficient implementations already exist in the form of python libraries. A physicist just needs to clean the physical data in order to apply a logistic regression model or any machine learning model for that matter, to solve his problem.

5 Predicting critical temperature for superconductivty using Decision Trees

5.1 Background: Predicting Critical Temperature for Superconductivity

Superconductivity is a phenomenon characterized by the perfect conduction of electricity - zero electrical resistance - along with the expulsion of magnetic fields. It occurs in certain materials below a particular temperature known as the critical temperature (T_c). Superconductivity, first discovered in 1911, is a profoundly quantum mechanical effect and a vital topic in condensed matter physics.

Superconducting materials can be categorized into two types, Type I and Type II, each exhibiting distinct magnetic properties. Type I superconductors completely expel magnetic fields up to the critical magnetic field, while Type II superconductors allow magnetic fields to penetrate through vortex lines even below the critical temperature.

The underlying mechanism for superconductivity is the formation of Cooper pairs, named after physicist Leon Cooper. These are pairs of electrons moving through a lattice that manage to overcome their natural repulsion and form a bound state. This pairing mechanism is mediated by phonons, quantized vibrations of the lattice, and happens below the critical temperature.

Predicting the critical temperature of superconductivity (T_c) is a challenging problem. Theoretical calculations based on BCS (Bardeen–Cooper–Schrieffer) theory often fail to predict T_c accurately for high-temperature superconductors, where other quantum effects become significant. Also, experimental methods to determine T_c are time-consuming, resource-intensive, and don't provide an a priori prediction for material design.

Machine learning offers a compelling tool to tackle this problem. Given enough data on superconductors with known features and corresponding T_c, a machine learning algorithm can learn the mapping from the feature space to the T_c and be used for prediction. A regression model, in this case, implemented through a decision tree algorithm, could be trained to predict the T_c for a given set of input features, hence significantly aiding the search for new superconducting materials.

5.2 Traditional Methods for Predicting Critical Temperature and their Limitations

The prediction of the critical temperature (T_c) in superconductors is a challenging problem that involves complex interactions at the quantum level. The traditional approach, based on Bardeen-Cooper-Schrieffer (BCS) theory, provided significant insight into this behavior.

BCS theory explains superconductivity as a macroscopic quantum state that results from the formation of Cooper pairs, pairs of electrons with opposite momenta and spin. These pairs form a condensate, which behaves as a single quantum entity and is responsible for the phenomenon of superconductivity.

The critical temperature in BCS theory is given by the equation:

$$T_c = \frac{2\hbar\omega_D}{k_B} \exp\left(-\frac{1}{N(0)V}\right) \tag{5.1}$$

where ω_D is the Debye frequency related to lattice vibrations (phonons), k_B is the Boltzmann constant, $N(0)$ is the density of states at the Fermi level, and V is the effective potential or the attraction strength between electrons.

Determining V and $N(0)$ accurately is not trivial. The effective interaction V arises from an electron exchanging a virtual phonon with another electron, creating an attractive interaction that binds them into a Cooper pair. Accurately calculating this interaction requires a detailed understanding of the electron-phonon interaction, which in turn depends on the electronic band structure and lattice dynamics, both computationally challenging to calculate.

Furthermore, $N(0)$, the density of states, is highly dependent on the electronic band structure, requiring detailed knowledge of the material's electronic properties.

Additionally, BCS theory assumes a weak electron-phonon interaction, valid in conventional low-temperature superconductors. However, this assumption fails in unconventional high-temperature superconductors where other interactions (like spin fluctuations) may play a crucial role.

Thus, the task of predicting T_c is challenging and computationally intensive using the traditional theoretical approaches. Experimental determination of T_c is resource-intensive and doesn't provide a predictive tool for new materials. In light of these challenges, an alternative approach using machine learning, such as decision tree regression, can provide a novel and efficient method to predict T_c.

5.3 Applying Decision Tree Regression to Predict Critical Temperature for Superconductivity

In light of the limitations of traditional methods, a decision tree regressor can provide an alternative approach to predict the critical temperature of superconductors. This model was trained on a comprehensive dataset containing 81 different attributes, including atomic mass, electron affinity, thermal conductivity, and valence, among others. The data set is hosted by the UC Irvine Machine Learning Repository.

```
1 import pandas as pd
```

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
```

This block of code imports necessary libraries. Pandas and numpy are used for data handling, matplotlib for plotting, and sklearn for machine learning tasks and evaluation metrics.

```
filename = "https://www.physi.uni-heidelberg.de/~reygers/lectures/2021/ml/
    data/train_critical_temp.csv"
df = pd.read_csv(filename, engine='python')
```

The data is loaded into a pandas dataframe from the given URL.

```
y = df['critical_temp'].values
X = df[[col for col in df.columns if col!="critical_temp"]]
```

The target variable, i.e., the critical temperature, is isolated from the feature set.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
    shuffle=True)
```

The data is then split into a training set and a test set, with 70% of the data used for training and 30% used for testing.

```
from sklearn.tree import DecisionTreeRegressor
import time

DTreg = DecisionTreeRegressor()

start_time = time.time()
DTreg.fit(X_train, y_train)
run_time = time.time() - start_time

print(run_time)
```

A decision tree regressor is instantiated and fitted to the training data, while also timing the training process for efficiency evaluation.

```
y_pred = DTreg.predict(X_test)
```

The trained regressor is used to predict the critical temperature for the test set.

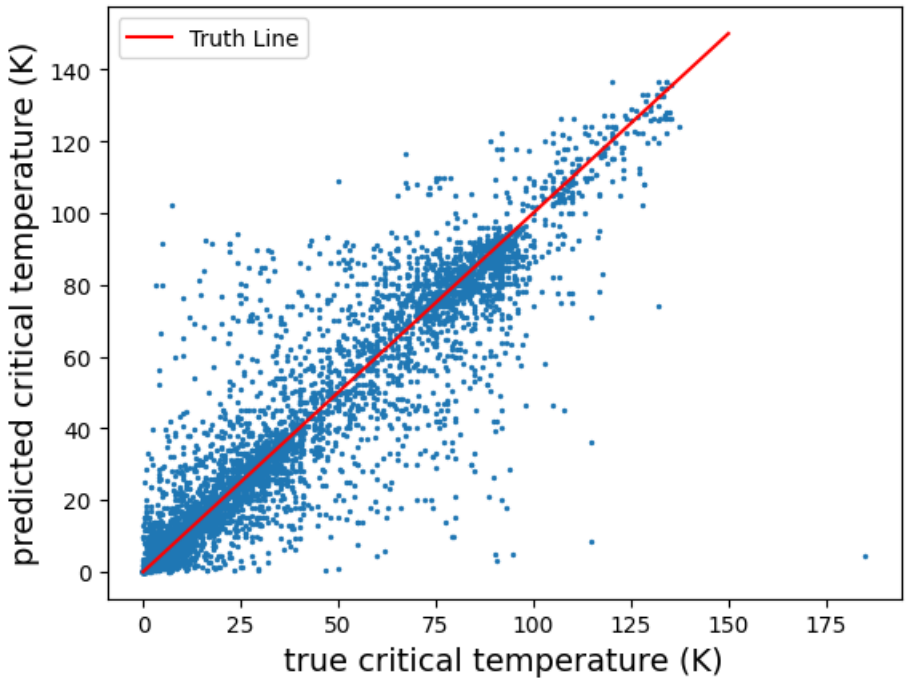
```
plt.scatter(y_test, y_pred, s=2)
plt.xlabel("true critical temperature (K)", fontsize=14)
plt.ylabel("predicted critical temperature (K)", fontsize=14)
plt.savefig("critical_temperature.pdf")
```

The predicted and actual critical temperatures are then plotted against each other, showing how well the model has learned to predict the critical temperature.

```
rms = np.sqrt(mean_squared_error(y_test, y_pred))
print(f"root mean square error {rms:.2f}")
r2 = r2_score(y_test, y_pred)
print(f"R2 score {r2:.3f}")
```

Finally, the root mean square error and R^2 score are calculated to quantitatively assess the performance of the model. They come out to be 12.12 and 0.874. In this context, a high R^2 score indicates that the features we selected are good predictors of the critical temperature.

Figure 29. An plot of the predicted values of the critical temperature against the true critical temperature.



This approach presents a substantial improvement over the traditional methods, as it doesn't need to compute the complicated interactions explicitly. Instead, it learns from the data. However, care should be taken when interpreting the results, as the model lacks the physical intuition embedded in traditional theories.

6 Applying Neural Networks for Separation of Signal and Background in an Exotic Higgs Scenario

6.1 Background of Neural Networks for Separation of Signal and Background in an Exotic Higgs Scenario

The field of particle physics aims to identify and study the fundamental constituents of matter and their interactions. The Standard Model (SM) of particle physics, a well-established theoretical framework, provides an incredibly successful description of the electromagnetic, weak, and strong interactions. The discovery of the Higgs boson at the Large Hadron Collider (LHC) at CERN in 2012, a key component of the SM, confirmed the mechanism of electroweak symmetry breaking, thereby giving masses to elementary particles through the Higgs field's non-zero vacuum expectation value.

However, the SM doesn't provide answers to several key questions, such as the nature of dark matter, the hierarchy problem, or the apparent absence of antimatter in the observable universe. Supersymmetry (SUSY) is one of the most promising extensions of the SM that can address some of these issues. SUSY introduces a symmetry between fermions (matter particles) and bosons (force-carrying particles), effectively doubling the particle spectrum.

In the context of the Minimal Supersymmetric Standard Model (MSSM), the simplest SUSY model, the Higgs sector is expanded to include two complex SU(2) doublets, which after electroweak symmetry breaking, lead to five physical Higgs bosons: two neutral CP-even (h and H), one neutral CP-odd (A), and a pair of charged bosons (H^+ and H^-).

One interesting aspect of the MSSM is the presence of a heavy neutral Higgs boson H^0 that can decay into two W bosons and a SM-like Higgs boson h, denoted as $H^0 \rightarrow W^+W^-h$. The W bosons can subsequently decay leptonically into a lepton and neutrino ($W \rightarrow l\nu$), while the Higgs boson can decay into a pair of b-quarks ($h \rightarrow b\bar{b}$). This process is intriguing due to its multi-lepton and b-jet final state.

The problem at hand is to distinguish this exotic Higgs decay, which is our signal, from the overwhelming SM background processes that can mimic the same final state. This background predominantly comes from top quark pair production ($t\bar{t}$), diboson production (WW, WZ, ZZ), and associated production of a W boson with a pair of heavy quarks.

Due to the complexity of this task, traditional analysis methods that rely on a series of hard cuts on physical observables have limitations. Machine Learning (ML), and in particular Deep Neural Networks (DNNs), offer a novel avenue to exploit the correlations between the various physical observables, potentially improving the discrimination power and enhancing the significance of the signal.

6.2 Traditional Methods and Their Limitations for Separation of Signal and Background

Particle physicists traditionally separate signals from backgrounds using a series of hard cuts on observable quantities, based on their understanding of the physics of the events of interest. This 'cut-based' approach involves creating a selection criteria which attempts to maximize the signal and minimize the background.

Consider our signal of interest, the decay chain $H^0 \rightarrow W^+W^-h$, where the W bosons decay leptonically ($W \rightarrow l\nu$), and the Higgs boson decays into a pair of b-quarks ($h \rightarrow b\bar{b}$). Given the multi-lepton and b-jet final state, traditional approaches would attempt to isolate events by placing requirements such as:

1. The event must contain at least two leptons (electrons or muons) of opposite charge to account for the leptonic W boson decays.

2. The event should also contain at least two b-tagged jets to account for the Higgs boson decaying into a pair of b-quarks.

3. Various kinematic cuts might be applied, for instance on the transverse momentum (p_T) and pseudo-rapidity (η) of the leptons and jets, to further refine the selection.

4. Additionally, global event variables such as the missing transverse energy (MET), which indicates the presence of neutrinos in the event, might be used.

5. Finally, invariant mass cuts can be used to select events where the invariant mass of the lepton pair or b-jet pair are close to the known masses of the W boson and the Higgs boson, respectively.

The choice of these cuts is typically based on maximizing the significance of the signal, defined as $S/\sqrt{B}$, where S is the number of signal events and B is the number of background events passing the selection criteria.

The main limitation of this approach is that it doesn't account for correlations between different observables. Each cut is applied independently, which can lead to a significant loss of signal events. Moreover, these hard cuts can be too strict or too lenient, which means that they could exclude potential signal events or include too many background events.

Additionally, the choice of cuts is somewhat arbitrary and heavily dependent on the experimenter's understanding and intuition of the physics involved. Two researchers might choose slightly different cuts, which could lead to different results.

Furthermore, these methods do not scale well with the increasing complexity of the data and the signal of interest. As more observables need to be considered, the cut-based approach quickly becomes impractical. The advent of machine learning techniques,

such as deep neural networks, provides an alternative, more scalable, and potentially more powerful method to tackle this problem.

6.3 Applying Neural Network Classification for Signal-Background Separation

For the problem of signal-background separation in the Higgs decay scenario, deep neural networks (DNNs) offer an elegant and effective solution. We will illustrate how DNNs can be used to tackle this challenge by leveraging the Python library 'Keras'.

We consider a dataset hosted by the UC Irvine Machine Learning Repository. The dataset includes both low-level quantities (var1 - var21) and high-level quantities (var22 - var28), and a binary classification label (1 = signal, 0 = background). The low-level features correspond to basic detector-level quantities while high-level features are derived from combinations of these, designed by physicists to provide better discrimination.

The first step involves loading the necessary libraries, and then reading the data:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import keras
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.optimizers import SGD
from sklearn.metrics import roc_auc_score

filename = "https://www.physi.uni-heidelberg.de/~reygers/lectures/2021/ml/
    data/HIGGS_100k.csv"
df = pd.read_csv(filename, engine='python', header = None).to_numpy()
```

We separate the data into features and labels and further split the data into training, validation and test sets.

```
y = df[:, 0].astype(int)
X = df[:, 22:]

from sklearn.model_selection import train_test_split

X_train_valid, X_test, y_train_valid, y_test = train_test_split(X, y,
    test_size=0.3)
X_train, X_valid, y_train, y_valid = train_test_split(X_train_valid,
    y_train_valid, test_size=0.1)
```

Next, we define the architecture of our deep neural network. This involves specifying the number of layers, the number of neurons in each layer, the activation function for each layer, and other specifications such as dropout rate:

```
model = Sequential()
model.add(Dense(100, input_dim=X.shape[1], kernel_initializer='uniform'))
```

```
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(100, kernel_initializer='normal'))
model.add(Activation('relu'))
model.add(Dropout(0.10))
model.add(Dense(1, kernel_initializer='normal'))
model.add(Activation('sigmoid'))
```

We then compile our model, specifying the loss function, the optimizer, and the metrics we want to track:

```
sgd = SGD(learning_rate=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='binary_crossentropy', metrics=['accuracy'], optimizer=
    sgd)
```

Next, we train the model for a specified number of epochs:

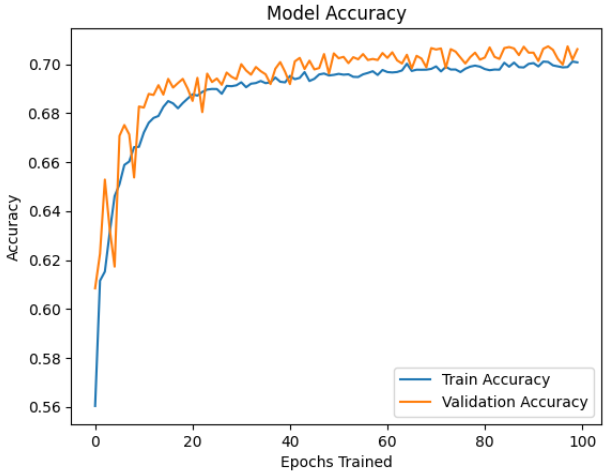
```
model.fit(X_train, y_train, epochs = 50, batch_size= 1000, validation_data
    =(X_valid, y_valid), shuffle=True)
```

Finally, we evaluate the model's performance using the ROC-AUC score:

```
model_ROC = roc_auc_score(y_test, model.predict(X_test))
print(model_ROC)
```

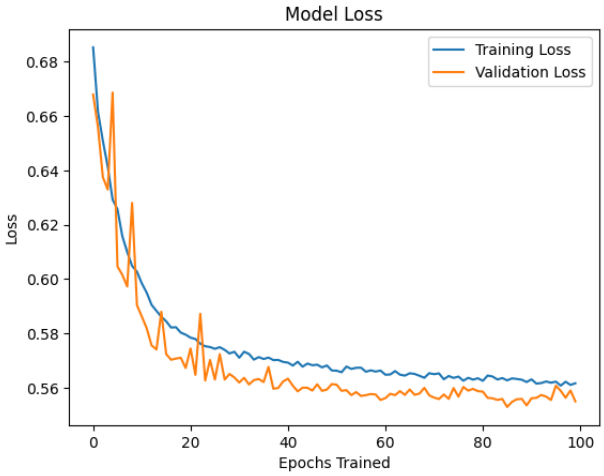
This neural network achieves an accuracy of 70% with an AUC value of 0.801. Following are some plots of the neural network as it trained.

Figure 30. A plot of the training and validation accuracies against epochs trained.



This model achieves an accuracy of 70% and and AUC value of 0.801, proving that deep neural network are capable of learning complex, non-linear relationships in the data and thereby provides an alternative to the traditional cut-based approach, potentially improving the discrimination between signal and background events. It also naturally handles correlations between different variables, which is a significant advantage over the

Figure 31. A plot of the training and validation losses against epochs trained.



traditional cut-based approach. The trained model can then be used to make predictions on new, unseen data, providing a valuable tool in the search for exotic Higgs scenarios.

7 Conclusion

This paper has endeavored to introduce some fundamental machine learning (ML) techniques and their applications in physics. By utilizing logistic regression, decision trees, and neural networks, we tackled several physics problems and achieved promising results.

Our study started with logistic regression, where we successfully classified air showers measured with the MAGIC telescope. The model was able to distinguish between gamma and hadron initiated showers with an accuracy of 78.9% and an area under the ROC curve of 0.742. In the next step, we utilized decision trees to predict the critical temperature for superconductivity and achieved a root mean square error of 12.12, with an R^2 value of 0.874. Finally, we applied neural networks to separate signal from background in an exotic Higgs scenario. This application resulted in a model with a 70% accuracy and an area under the ROC curve of 0.801.

These applications demonstrate that ML provides a robust set of tools that can significantly contribute to the analysis and interpretation of complex physical phenomena. Moreover, they highlight the potential of ML to become an integral part of modern physics research. However, this paper only scratches the surface of this vast and rapidly evolving field.

Advanced topics in ML, which are beyond the scope of this paper, can further enhance model performance and provide novel perspectives for approaching complex physics problems. For instance, ensemble methods such as Random Forests and Gradient Boosting Machines can create more powerful models by aggregating multiple weaker ones.

Convolutional Neural Networks (CNNs) have proven remarkably successful in analyzing image data and could be applied to problems in physics where spatial relationships are essential, like the detection of patterns in particle collisions. Recurrent Neural Networks (RNNs) have shown extraordinary results in time-series predictions, opening the door to applications in numerous physics domains involving temporal data.

Hyperparameter tuning methods like grid search and Bayesian optimization could potentially improve model performance. Advanced regularization techniques like Lasso and Ridge Regression, or Dropout and Batch Normalization for neural networks, can help prevent overfitting and improve model generalization.

In conclusion, ML techniques offer powerful methodologies for understanding and solving complex problems in physics. These results and the vast range of available ML tools serve as a strong motivation for undergraduate physics students to delve deeper into ML. As we continue to explore the intricacies of the universe, ML will undoubtedly play an increasingly vital role in shaping our understanding and presenting new avenues of exploration.

References

[1] François Chollet. *Deep Learning with Python*. Manning, November 2017.

[2] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to machine learning for physicists. *Physics Reports*, 810:1–124, may 2019.

[3] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová . Machine learning and the physical sciences. *Reviews of Modern Physics*, 91(4), dec 2019.

[4] R.K. Bock, A. Chilingarian, M. Gaug, F. Hakl, T. Hengstebeck, M. Jiřina, J. Klaschka, E. Kotrč, P. Savický, S. Towers, A. Vaiciulis, and W. Wittek. Methods for multidimensional event classification: a case study using images from a cherenkov gamma-ray telescope. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 516(2):511–528, 2004.

[5] Kam Hamidieh. A data-driven statistical model for predicting the critical temperature of a superconductor. *Computational Materials Science*, 154:346–354, 2018.

[6] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5(1), jul 2014.