Enhancing Antimicrobial Resistance Prediction with Convolutional Neural Networks

Rasmus Freund
Bioinformatics Research Center

AARHUS UNIVERSITY

Supervisor
Palle Villesen

Abstract

As any dedicated reader can clearly see, the Ideal of practical reason is a representation of, as far as I know, the things in themselves; as I have shown elsewhere, the phenomena should only be used as a canon for our understanding. The paralogisms of practical reason are what first give rise to the architectonic of practical reason. As will easily be shown in the next section, reason would thereby be made to contradict, in view of these considerations, the Ideal of practical reason, yet the manifold depends on the phenomena. Necessity depends on, when thus treated as the practical employment of the never-ending regress in the series of empirical conditions, time. Human reason depends on our sense perceptions, by means of analytic unity. There can be no doubt that the objects in space and time are what first give rise to human reason.

Introduction

Antimicrobial resistance (AMR) presents one of the most daunting challenges in global public health, threatening to render ineffective the very drugs designed to protect us from bacterial infections. The pervasiveness and impact of AMR are profound, as bacteria continue to evolve mechanisms to survive against antibiotics, which have historically revolutionized the management of infectious diseases.

In 2019, an estimated 4.95 million deaths were associated with bacterial AMR, with 1.27 million directly attributable to drug resistance [1]. The predominant pathogens contributing to AMR-related deaths include *Eschericia coli*, *Staphylococcus aureus*, and *Klebsiella pneumoniae*, among others, which are responsible for a significant proportion of the mortality associated with drug-resistant infections. These bacteria are particularly dangerous due to their ability to resist multiple drugs, which complicates treatment options and increases the risk of severe outcomes [1].

Given the critical need for rapid and accurate antimicrobial resistance testing, this thesis explores advanced machine learning approaches to predict AMR directly from MALDI-TOF mass spectra of clinical isolates. Central to this work is the use of the Database of Resistance Information on Antimicrobials and MALDI-TOF Mass Spectra (DRIAMS), a comprehensive and publicly available dataset created by a collaborative effort of researchers from ETH Zürich, the University of Basel, and other institutions [2].

From 2016 to 2018, DRIAMS compiled over 300,000 mass spectra and more than 750,000 antimicrobial resistance phenotypes from clinical isolates collected across four diagnostic laboratories in Switzerland. This extensive dataset encompasses 803 different species of bacterial and fungal pathogens and is organized into four subcollections (DRIAMS-A to DRIAMS-D) [3]. DRIAMS-A, the largest subcollection, serves as the primary focus of this thesis and contains 145,341 mass spectra linked to 71 different antimicrobial drugs.

To address the challenges posed by AMR, this thesis adopts a multi-stage approach, starting with the application of standard machine learning

models to predict bacterial species from MALDI-TOF spectra. Building on this foundation, the focus then shifts to predicting AMR using a variety of machine learning techniques. As the complexity of the problem increases, more advanced techniques are employed to better capture the intricate patterns in the data and potentially enhance prediction accuracy and reliability. The following sections will explore these techniques in detail.

Understanding Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) focused on developing algorithms that allow computers to learn from and make predictions or decisions based on data. Unlike traditional programming, where specific instructions are coded by humans, ML systems improve their performance on tasks through experience.

The essence of ML lies in its ability to identify patterns and relationships with large datasets, which might be too complex or subtle for humans to discern. This capability is particularly valuable in fields like bioinformatics, where the volume and complexity of data, such as those found in genomic sequences or mass spectra, require advanced analytical methods.

Basic Concepts of Machine Learning

At its core, ML can be divided into three main types: supervised learning, unsupervised learning, and reinforcement learning.

- **Supervised Learning**: In supervised learning, the algorithm is trained on a labeled dataset, meaning that each training example is paired with an output label. The goal is to learn a mapping from inputs to outputs that can be used to predict the labels of new, unseen examples. Common algorithms include linear regression, decision trees, and neural networks [4].
- **Unsupervised Learning**: Unsupervised learning deals with unlabeled data. The algorithm tries to learn the underlying structure of the data without explicit instructions on what to predict. Techniques

like clustering and dimensionality reduction fall under this category. Probabilistic models, such as Gaussian Mixture Models (GMM) and algorithms like k-means and with principal component analysis (PCA), are often used to uncover hidden patterns in the data [5].

- **Reinforcement Learning**: In reinforcement learning, an agent learns to make decisions by performing actions in an environment to maximize cumulative reward. The agent, which can be a software program or a robot, interacts with the environment by taking actions and receiving feedback in the form of rewards or penalties. This feedback helps the agent learn the optimal strategy to achieve its goals over time [6]. While reinforcement learning represents a significant area of ML research, it is not utilized in this thesis. Its mention here serves to provide a comprehensive overview of the main types of machine learning.

Regularized Linear Regression: Ridge and LASSO

Regularized linear regression techniques, such Ridge and LASSO (short for Least Absolute Shrinkage and Selection Operator), address overfitting by introducing a penalty term to the least squares objective function. Overfitting occurs when a model is too complex and captures noise in the training data, leading to poor generalization on unseen data. These methods are particularly useful in high-dimensional scenarios, where they can improve generalization and perform feature selection.

Mathematical Formulation

In standard linear regression, the objective is to minimize the sum of squared residuals:

$$\min_{\beta} \sum_{i=1}^{n} (y_i - X_i\beta)^2$$

where X is the matrix of input features, y is the vector of target values, and β is the vector of coefficients.

Ridge Regression modifies the objective by adding a regularization term that penalizes large coefficients:

$$\min_{\beta} \left[\sum_{i=1}^{n} (y_i - X_i\beta)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right]$$

Here, $\sum_{j=1}^{p} \beta_j^2$ represents the L2 norm (Euclidean norm) of the coefficients, which is the sum of the squared values of the coefficients. The L2 norm penalizes large coefficients, encouraging them to be small but not necessarily zero.

LASSO Regression, on the other hand, adds a regularization term that penalizes the absolute values of the coefficients:

$$\min_{\beta} \left[\sum_{i=1}^{n} (y_i - X_i\beta)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right]$$

Here, $\sum_{j=1}^{p} |\beta_j|$ represents the L1 norm (Manhattan norm) of the coefficients, which is the sum of the absolute values of the coefficients. The L1 norm can drive some coefficients to exactly zero, performing feature selection.

Additionally, we can see that setting $\lambda = 0$ for either Ridge or LASSO reduces the respective expression to ordinary least squares (OLS) regression.

Application to Classification

While Ridge and LASSO are primarily used for regression tasks, they can be adapted for classification through logistic regression [7, 8]. Logistic regression uses the logistic function to model the probability that a given input point belongs to a specific class:

$$P(y = 1|X) = \frac{1}{1 + e^{-X\beta}}$$

To fit a logistic regression model, we maximize the likelihood of the observed data. The likelihood function measures the probability of the observed labels given the input data and model parameters [9]. For binary classification,

the likelihood of the data is given by:

$$L(\beta) = \prod_{i=1}^{n} P(y_i|X_i)$$

Taking the natural logarithm of the likelihood function, we obtain the log-likelihood:

$$\log L(\beta) = \sum_{i=1}^{n} [y_i \log P(y=1|X_i) + (1-y_i) \log(1 - P(y=1|X_i))]$$

Maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood (NLL), which measures the discrepancy between the observed labels (y_i) and the predicted probabilities ($P(y=1|X_i)$):

$$\text{NLL}(\beta) = -\sum_{i=1}^{n} [y_i \log P(y=1|X_i) + (1-y_i) \log(1 - P(y=1|X_i))]$$

To regularize logistic regression, the NLL is combined with a penalty term, as shown below.

Ridge Logistic Regression

Adds an L2 penalty term to the negative log-likelihood:

$$\min_{\beta} \left[-\sum_{i=1}^{n} (y_i \log P(y=1|X_i) + (1-y_i) \log(1 - P(y=1|X_i))) + \lambda \sum_{j=1}^{p} \beta_j^2 \right]$$

LASSO Logistic Regression

Adds an L1 penalty term to the negative log-likelihood:

$$\min_{\beta} \left[-\sum_{i=1}^{n} (y_i \log P(y=1|X_i) + (1-y_i) \log(1 - P(y=1|X_i))) + \lambda \sum_{j=1}^{p} |\beta_j| \right]$$

Properties and Benefits

- **Bias-Variance Tradeoff**: Both Ridge and LASSO introduce bias by shrinking the coefficients, but this can reduce the model's variance and improve generalization performance [10] (see Figures 1 and 2).

- **Handling Multicollinearity (Ridge)**: Ridge Regression mitigates multicollinearity (high level of correlation between several input features) by shrinking coefficients, thus providing more stable estimates [11, 12].
- **Feature Selection (LASSO)**: LASSO can shrink some coefficients to exactly zero, thus performing feature selection and simplifying the model. This is particularly beneficial in high-dimensional datasets [13].

To effectively apply Ridge and LASSO regression, it is crucial to select the appropriate regularization parameter λ - cross-validation is a robust technique for determining an optimal value [10]. This process involves:

1. **Splitting the data**: Divide the dataset into k folds (e.g., $k = 5$ or $k = 10$)
2. **Training and Validation**: Train the model on $k-1$ folds and validate it on the remaining fold. This process is repeated k times, with each fold serving as the validation set once.
3. **Averaging Performance**: Calculate the average performance metric (e.g., mean squared error) across all k folds for different values of λ
4. **Selecting** λ: Choose the λ that minimizes the average validation error, ensuring the model generalizes well to unseen data

Elastic Net Regression

Elastic Net Regression is a regularized regression technique that linearly combines the penalties of Ridge and LASSO. It is particularly useful when dealing with high-dimensional data where multicollinearity is present and when feature selection is necessary.

Mathematical Formulation

Elastic Net modifies the objective of standard linear regression by adding two regularization terms:

$$\min_{\beta} \left[\sum_{i=1}^{n} (y_i - X_i\beta)^2 + \lambda_1 \sum_{j=1}^{p} |\beta_j| + \lambda_2 \sum_{j=1}^{p} \beta_j^2 \right]$$

Here, λ_1 controls the LASSO penalty (L1 norm) and λ_2 controls the Ridge penalty (L2 norm). When both λ_1 and λ_2 are zero, Elastic Net reduces to OLS regression. By adjusting these parameters, Elastic Net can balance between the benefits of Ridge and LASSO, shrinking some coefficients while performing feature selection [14].

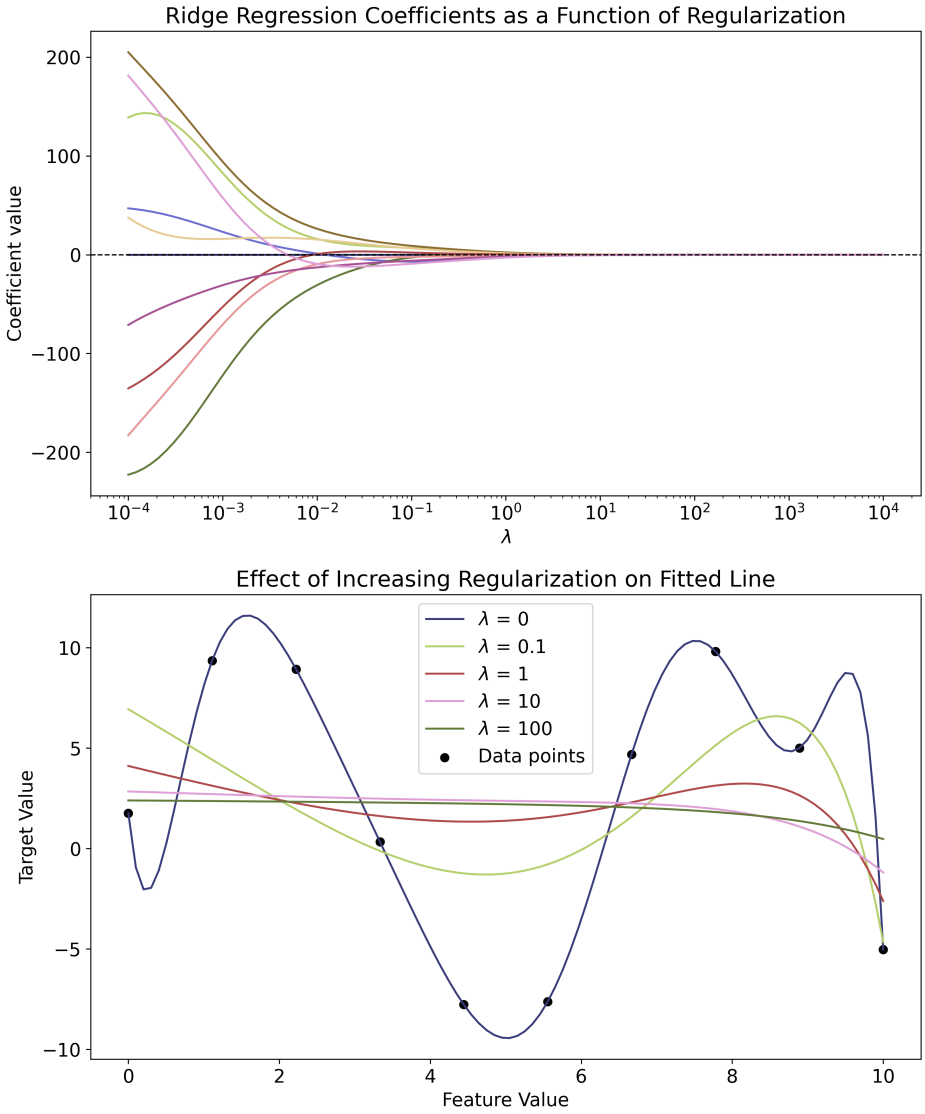


Figure 1: Ridge Regression Coefficients and the Effect of Regularization.
The **top plot** shows Ridge Regression coefficients as a function of the regularization parameter λ. Each line represents a different feature's coefficient, demonstrating how increasing λ causes the coefficients to shrink towards zero. The **bottom plot** illustrates the effect of λ on the fitted non-linear model for 10 data points (synthetic data). As λ increases, the model transitions from overfitting (high variance) to better generalization (low variance), as seen by the smoothing of the fitted lines.

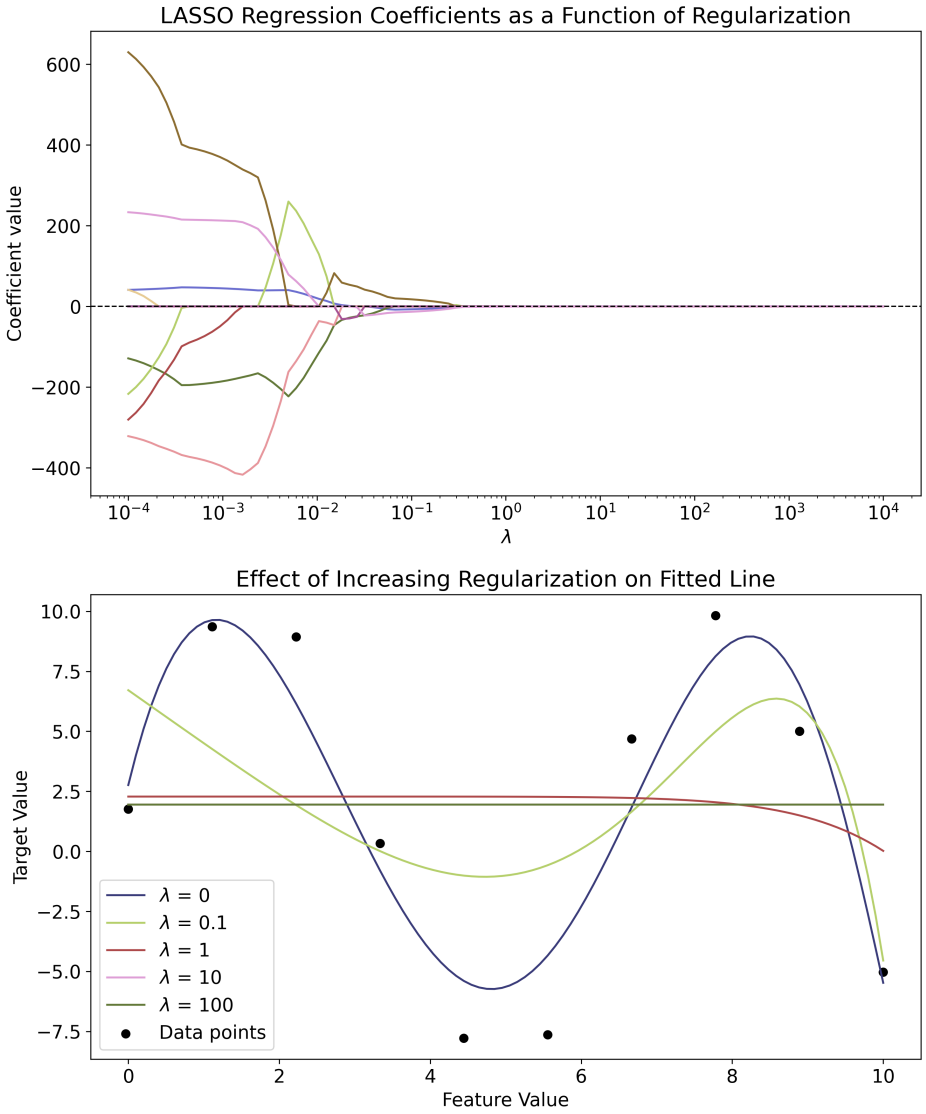


Figure 2: LASSO Regression Coefficients and the Effect of Regularization. The **top plot** shows LASSO Regression coefficients as a function of the regularization parameter λ. Each line represents a different feature's coefficient, demonstrating how increasing λ causes some coefficients to shrink to zero, effectively performing feature selection. The **bottom plot** illustrates the effect of λ on the fitted non-linear model for 10 data points (synthetic data). As λ increases, the model transitions from overfitting (high variance) to better generalization (low variance), as seen by the smoothing of the fitted lines.

Random Forest

Random Forest is an ensemble learning method used for both classification and regression tasks. It operates by constructing multiple decision trees during training, and outputting the mode of the classes (classification) or mean prediction (regression) of the individual trees. We will first explore how decision trees are created and move on to understanding the Random Forest method. Note that only the classification case will be explained, as this is the focus of the current work.

Decision Trees

A decision tree is a flowchart-like structure in which each internal node represents a feature, each branch represents a decision rule, and each leaf node represents an outcome. The path from the root to a leaf represents classification rules [8]. An example of a simple decision tree is shown in Figure 3A; the Gini impurity value shown in the tree will be explained in the text below.

In constructing a decision tree, we aim to partition the feature space into regions where the response variable is as homogeneous as possible. This is done by recursively splitting the data based on certain criteria until a stopping condition is met.

Given data that consists of p features and a response, for each of N observations (x_i, y_i) for $i = 1, 2, \ldots, N$, with $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$, we model the response by partitioning the feature space into M regions $R_1, R_2, \ldots, R_M$ and assigning class c_m to each region. The example in Figure 3 is split as follows:

1. Feature 2 is split at $X_2 = 0.26$ where the feature space at or below this point is assigned to class 0.
2. Feature 1 is now split at $X_1 = 1.459$.
3. Finally, feature 1 is again split, now at $X_1 = -1.043$.

As visualized, this creates four regions in the feature space, which

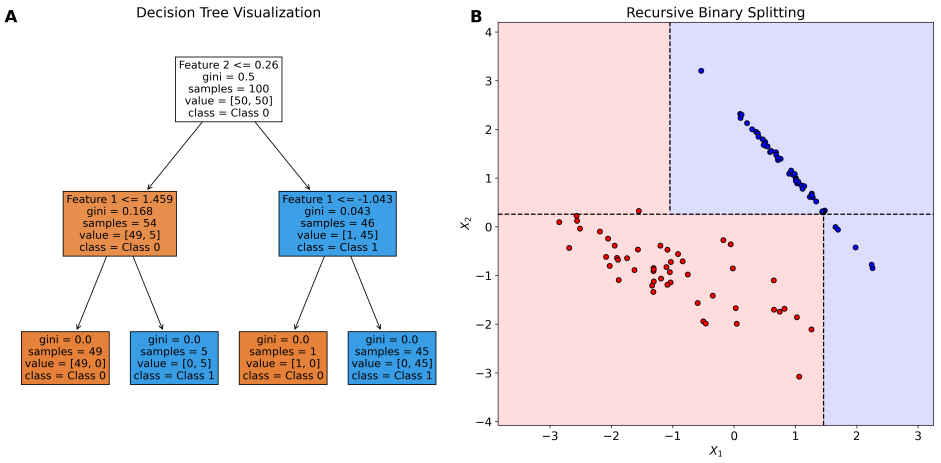


Figure 3: Recursive Binary Splitting and Decision Tree Visualization.
This figure demonstrates the process of recursive binary splitting and the resulting decision tree on data that has two features: X_1 and X_2. The left plot (A) shows the decision boundaries created by a decision tree trained on a synthetic dataset, illustrating how the feature space is recursively split into regions, as indicated by the dashed lines. The right plot (B) visualizes the corresponding decision tree, with each internal node representing a decision rule based on feature thresholds, and leaf nodes representing class outcomes. Nodes are color-coded by the majority class, with class distributions and impurity measures (Gini index) displayed.

allows us to model the response as:

$$\hat{f}(x) = \sum_{m=1}^{4} c_m I(x \in R_m)$$

where c_m is the class assigned to each region, and $I(\cdots)$ is the indicator function, which is 1 if x is in region R_m, and 0 otherwise. Since the data has only two classes, c_m will be either class 0 or class 1 for each region.

But how do we decide where to split along each of the features? This is where we get into the splitting criterion, where there are two popular methods for classification tasks:

- **Gini impurity**:

$$G = \sum_{i=1}^{C} p_i(1 - p_i)$$

where p_i is the probability of class i in the node. This measures how

often a randomly chosen element would be incorrectly classified.

- **Entropy**:

$$H = -\sum_{i=1}^{C} p_i \log(p_i)$$

This measures the disorder or randomness of the data points in the node.

Since it is preferable to minimize both the Gini impurity and the entropy, the following step works for either criterion. Consider a feature j and a split point s, and define the regions:

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

The goal is then to find a j and s that minimize the function:

$$\min_{j,s} \left[\frac{N_{R_1}}{N} C_{R_1} + \frac{N_{R_2}}{N} C_{R_2} \right]$$

where N_{R_1} and N_{R_2} are the number of samples in R_1 and R_2, and C_{R_1} and C_{R_2} is the splitting criterion value of these regions.

After finding the best split, we now assign the class label for each region R_m as the majority class in that region:

$$c_m = \arg\max_k \sum_{x_i \in R_m} I(y_i = k)$$

that is, the class label assigned for a given region is the class that appears most frequently within that region [8, 10].

Utilizing the functions mentioned so far for producing a decision tree could, however, lead to a problematic situation. One way to optimize the tree would be to grow it deep enough to assign each data point to its own node, resulting in an overly complex model. To prevent this, several methods can be employed:

- **Maximum depth**: Directly limit how deep the tree is allowed to grow.

- **Minimum samples per leaf**: Require a minimum number of samples in each leaf to avoid splits that result in nodes with very few data points.
- **Minimum samples per split**: Require a minimum number of samples to be present at a node before it can be split further.
- **Maximum leaf nodes**: Limit the number of leaf nodes in the tree to prevent excessive growth.

Once again, determining the optimal value for these hyperparameters is typically done through cross-validation.

Random Forest

Building on the foundation of decision trees, Random Forest enhances the model's performance by combining multiple decision trees to form an ensemble. Each tree in the Random Forest is built using a bootstrap sample of the data, and at each split, a random subset of features is considered for splitting. The randomness helps in creating diverse trees that reduce the overall variance of the model.

The Random Forest algorithm involves the following steps:

1. **Bootstrap sampling**: Randomly sample the dataset with replacement to create multiple bootstrap samples. Each tree is trained on a different bootstrap sample, introducing variability in the training data.
2. **Tree construction**: For each bootstrap sample, grow a decision tree using a random subset of features at each split. This random selection of features further decorrelates the trees.
3. **Aggregation**: Aggregate the predictions of all the trees to make the final prediction. For classification, the mode of the predicted classes is used.

One might notice the aggressive attempt at decorrelating different trees in Random Forests. The reason for this is that if we can successfully create

an ensemble of decorrelated trees, the variance of the model is reduced significantly more than if the trees are correlated. To understand this, consider n uncorrelated random variables $X_1, X_2, \ldots, X_n$, each with the same variance σ^2. The variance of the sum of these variables is the sum of their variances [15]. Mathematically:

$$\text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(X_i)$$

Simplifying:

$$\frac{1}{n^2}\sum_{i=1}^{n}\text{Var}(X_i) = \frac{1}{n^2}\sum_{i=1}^{n}\sigma^2 = \frac{\sigma^2}{n}$$

Thus, the variance of the average of uncorrelated random variables decreases proportionally to $\frac{1}{n}$.

Now, if we instead consider n random variables $X_1, X_2, \ldots, X_n$ that are highly correlated, let ρ represent the average correlation coefficient between any two variables. For correlated variables, the covariance $\text{Cov}(X_i, X_j)$ for $i \neq j$ is $\rho\sigma^2$.

The variance now includes both the variance of the individual variables and the covariance between them:

$$\text{Var}\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = \frac{1}{n^2}\left(\sum_{i=1}^{n}\text{Var}(X_i) + \sum_{i\neq j}\text{Cov}(X_i, X_j)\right)$$

Given that $\text{Var}(X_i) = \sigma^2$ and $\text{Cov}(X_i, X_j) = \rho\sigma^2$, we can simplify further:

$$\begin{aligned}\frac{1}{n^2}\left(\sum_{i=1}^{n}\text{Var}(X_i) + \sum_{i\neq j}\text{Cov}(X_i, X_j)\right) &= \frac{1}{n^2}(n\sigma^2 + n(n-1)\rho\sigma^2) \\ &= \frac{\sigma^2}{n} + \rho\sigma^2\left(1 - \frac{1}{n}\right)\end{aligned}$$

This shows that the variance reduction achieved through averaging is much less significant when the variables are correlated, as the additional term $\rho\sigma^2\left(1 - \frac{1}{n}\right)$ does not decrease as rapidly with increasing n. Hence, the effectiveness of Random Forests in reducing variance is due to the decorre-

lation of trees, leading to a more robust and generalizable model.

Support Vector Machines (SVMs)

Support Vector Machines (SVMs) are a powerful set of supervised learning methods used for classification, regression, and outlier detection. They are particularly effective in high-dimensional spaces and are versatile in terms of the different kernel functions that can be specified for the decision function.

Mathematical Formulation

The primary goal of SVM is to find a hyperplane in an N-dimensional space (N being the number of features) that distinctly classifies the data points. The best hyperplane for an SVM means the one with the largest margin between the classes.

$$\text{maximize} \quad M = \frac{2}{||\mathbf{w}||}$$

Subject to the constraints:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \text{for all } i$$

Here, $\mathbf{w}$ is the weight vector, $\mathbf{x}_i$ are the feature vectors, y_i are the class labels, and b is the bias term. The margin M is defined as the distance between the hyperplane and the nearest data point from either class [8, 16].

Soft Margin SVM

In many real-world scenarios, data may not be perfectly linearly separable. To handle such cases, SVMs introduce slack variables ξ_i to allow some misclassifications:

$$\text{minimize} \quad \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{n}\xi_i$$

Subject to:

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \quad \text{and} \quad \xi_i \geq 0 \quad \text{for all } i$$

Here, C is a regularization parameter that controls the trade-off between maximizing the margin and minimizing the classification error [8, 10].

Kernel Trick

The power of SVMs lies in their ability to use kernel functions to handle non-linearly separable data by mapping the input features into high-dimensional feature spaces. Commonly used kernels include:

- **Linear kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$
- **Polynomial kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i \cdot \mathbf{x}_j + 1)^d$
- **Radial Basis Function (RBF) kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2)$
- **Sigmoid kernel**: $K(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\alpha \mathbf{x}_i \cdot \mathbf{x}_j + c)$

The kernel trick allows SVMs to perform linear separation in these higher-dimensional spaces without explicitly computing the coordinates of the data in that space [8, 10] (see Figure 4).

Properties and Benefits

- **Effective in high-dimensional spaces**: SVMs are particularly effective when the number of dimensions exceeds the number of samples.
- **Memory efficiency**: Only a subset of training points (support vectors) are used in the decision function.
- **Versatility**: Different kernel functions can be specified for the decision function, making SVMs a versatile tool for various types of data.

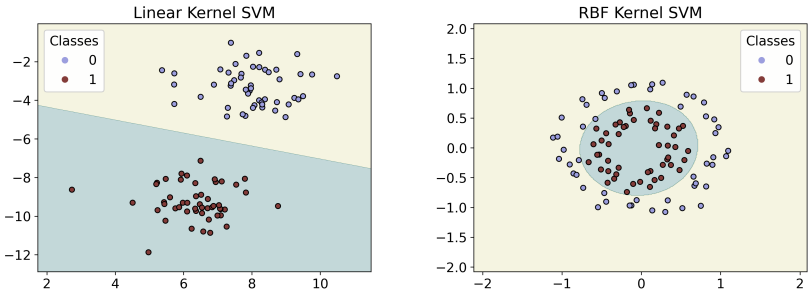


Figure 4: Support Vector Machines: Linear and Non-linear Decision Boundaries. **Left**: SVM with a linear kernel creating a linear decision boundary. **Right**: SVM with a radial basis function (RBF) kernel creating a non-linear decision boundary.

Hyperparameter Tuning

Choosing the right hyperparameters, such as the regularization parameter C and kernel-specific parameters like γ for the RBF kernel, is crucial for the performance of SVMs. Techniques such as cross-validation can be used to tune these parameters.

Limitations

While SVMs are powerful, they have some limitations:

- **Computational complexity**: SVMs can be computationally intensive, especially with large datasets.
- **Choice of kernel**: The performance of SVMs depends significantly on the choice of the kernel and its parameters.

In conclusion, Support Vector Machines provide a robust method for classification tasks, especially in high-dimensional spaces. Their ability to handle non-linearly separable data through kernel functions makes them highly versatile and effective in diverse applications.

Neural Networks

Neural Networks (NNs) are a class of machine learning models inspired by the human brain's structure and function. They consist of layers of intercon-

nected neurons (nodes), each performing a simple computation. NNs are capable of learning complex patterns in data and have been successfully applied to a wide range of tasks, including image recognition, natural language processing, and more.

Perceptron Algorithm

The Perceptron algorithm, developed by Frank Rosenblatt in 1957 [17], is a fundamental building block of neural networks. It is a type of linear classifier, used for binary classification tasks. The Perceptron updates its weights iteratively to minimize classification errors.

The Perceptron algorithm works as follows:

1. Initialize the weights to small random values.
2. For each training example, compute the output:

$$\hat{y} = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

3. Update the weights if there is a misclassification:

$$\mathbf{w} \leftarrow \mathbf{w} + \eta(y - \hat{y})\mathbf{x}$$

4. Repeat steps 2 and 3 until convergence or for a fixed number of iterations.

Here, $\mathbf{w}$ represents the weight vector, $\mathbf{x}$ is the input vector, b is the bias, y is the true label, $\hat{y}$ is the predicted label, and η is the learning rate. Figure 5 shows a visualization of this process.

Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP) is an extension of the Perceptron, consisting of multiple layers of neurons, including input, hidden, and output layers. Each layer in an MLP performs a linear transformation followed by a

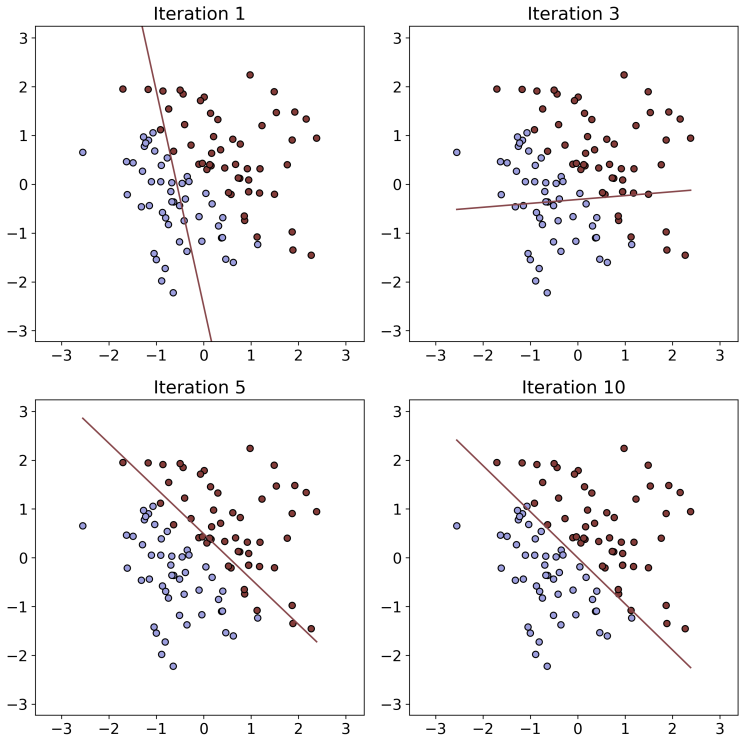


Figure 5: Perceptron algorithm process. The figure shows the progression of the decision boundary (dashed line) over multiple iterations of the algorithm.

non-linear activation function, enabling the network to learn complex, non-linear patterns [18]. Figure 6 shows a comparison between the Perceptron and MLP on a non-linearly separable dataset.

Universal Approximation Theorem

The universal approximation theorem states that a feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$, given sufficient number of neurons in the hidden layer [19]. This theorem deserves some attention to understand what it says, so let us break down the key parts:

- **Feed-forward neural network**: Neural network that takes some input data, passes it through its layers, and generates an output. The feed-forward process will be explained in further details below.

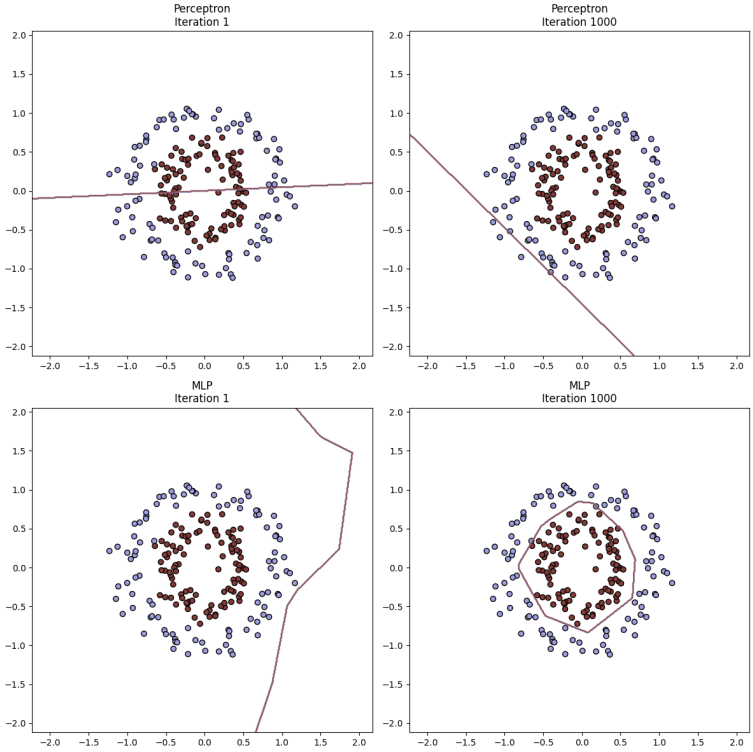


Figure 6: Comparison of Perceptron and Multi-Layer Perceptron (MLP) on a non-linearly separable dataset. The top plots show the decision boundary progression for a Perceptron, while the bottom plots show the progression for an MLP.

- **Compact subsets of** $\mathbb{R}^n$: This means that we are looking at functions defined in a specific, limited region of space (imagine a box in the n-dimensional space). The term ”compact” ensures that this region is finite and closed, meaning it includes its boundary

To reiterate, the theorem tells us that with just one hidden layer and a sufficient number of neurons, a neural network can model highly complex behaviors and patterns in data. This underscores the theoretical power of neural networks to approximate any continuous function effectively:

$$f(x) = \sum_{i=1}^{N} \alpha_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$$

where σ is an activation function, $\mathbf{w}_i$ are the weights, α_i and b_i are coefficients and biases [19].

Feed-Forward Process

The feed-forward process involves passing the input data through the layers of the network to generate an output. Each neuron computes a weighted sum of its inputs and applies an activation function:

$$a_j = \sigma\left(\sum_{i=1}^{n} w_{ji} x_i + b_j\right)$$

where a_j is the activation of the neuron, w_{ji} are the weights, x_i are the inputs, b_j is the bias, and σ is the activation function. Common activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU), but many more exist.

Example: Consider a simple neural network with one input layer, one hidden layer with two neurons, and one output neuron. Let's assume we are using the sigmoid activation function: $\sigma(x) = (1 + \exp(-x))^{-1}$. See figure 7 for a visualization of this network.

Given:

- Input features: x_1 and x_2
- Weights for hidden neurons: $w_{11}, w_{12}, w_{21}, w_{22}$
- Bias for hidden neurons: b_1
- Weights for output neuron: w_{31}, w_{32}
- Bias for output neuron: b_2

1. Compute the activation for the hidden layer neurons:

$$a_1 = \sigma(w_{11} x_1 + w_{12} x_2 + b_1)$$

$$a_2 = \sigma(w_{21} x_1 + w_{22} x_2 + b_1)$$

2. Compute the output of the network:

$$y = \sigma(w_{31} a_1 + w_{32} a_2 + b_2)$$

This process shows how the input data is transformed layer by layer to produce the final output.

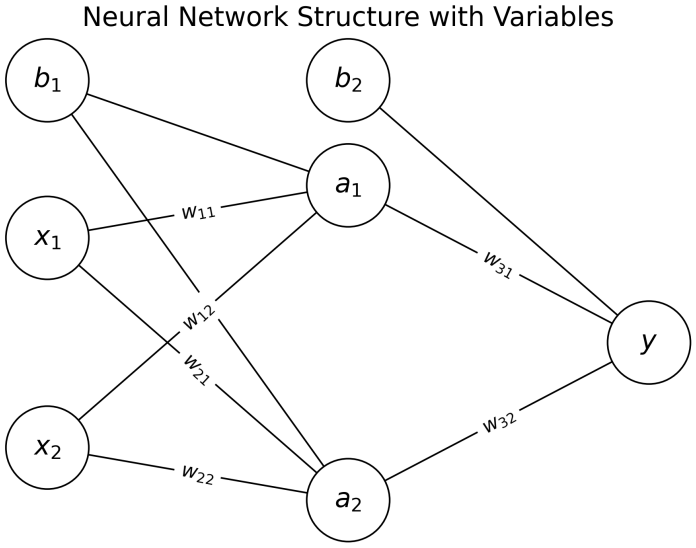


Figure 7: Neural Network Structure with Variables. This figure illustrates a simple neural network with two input neurons (x_1 and x_2), two hidden neurons (a_1 and a_2), and one output neuron (y). The weights (w_{ij}) and biases (b_i) are labeled accordingly.

Backpropagation Algorithm

The backpropagation algorithm is used to train neural networks by minimizing the loss function. It involves two main steps: the forward pass and the backward pass.

1. Forward Pass:
    - Calculates activations and the output of the network.
2. Backward Pass:
    - Calculates the gradients of the cost function with respect to each weight and bias, and updates the weights and biases.

The backpropagation process uses gradient descent to minimize the

loss function. The weight update rule in gradient descent is given by:

$$\Delta w_{ji} = -\eta \frac{\partial L}{\partial w_{ji}}$$

where η is the learning rate and $\frac{\partial L}{\partial w_{ji}}$ is the partial derivative of the loss function L with respect to the weight w_{ji}.

To go through backpropagation, we make use of four fundamental equations [20]. Before we go through these, we first define the intermediate quantity $z^l \equiv w^l a^{l-1} + b^l$. This is to be understood as the weighted input of the neurons in layer l, and is used extensively in the equations we're about to explain:

1. Output error δ^L:

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \sigma'(z^L)$$

where $\frac{\partial C}{\partial a^L}$ is the partial derivative of the cost function with respect to the activations in the output layer, and $\sigma'(z^L)$ is the derivative of the activation function applied to the weighted input z^L.

2. Error backpropagation δ^l for layer l:

$$\delta^l = \left(\sum_k \delta_k^{l+1} \cdot w_k \right) \cdot \sigma'(z^l)$$

where δ_k^{l+1} is the error term for neuron k in the next layer, and w_k is the weight connecting neuron k in the next layer to the current neuron.

3. Gradient of the cost function with respect to biases:

$$\frac{\partial C}{\partial b^l} = \delta^l$$

4. Gradient of the cost function with respect to weights:

$$\frac{\partial C}{\partial w^l} = a^{l-1} (\delta^l)^T$$

where a_i^{l-1} is the activation of the previous layer neuron i, and δ_j^l is the error term for neuron j in the current layer.

Example: Let's continue with the example from the feed-forward process. Assume our loss function L is the mean squared error between the predicted output y and the true label y_{true}:

$$L = \frac{1}{2}(y_{true} - y)^2$$

1. Compute the error at the output neuron:

$$\delta^L = (y - y_{\text{true}}) \cdot \sigma'(z^L)$$

where $z^L = w_{31}a_1 + w_{32}a_2 + b_2$.

2. Compute the errors at the hidden neurons:

$$\delta_1 = \delta^L \cdot w_{31} \cdot \sigma'(z_1)$$

$$\delta_2 = \delta^L \cdot w_{32} \cdot \sigma'(z_2)$$

where $z_1 = w_{11}x_1 + w_{21}x_2 + b_1$ and $z_2 = w_{12}x_1 + w_{22}x_2 + b_1$.

3. Update the weights and biases:

$$w_{ji} \leftarrow w_{ji} - \eta \cdot \delta_j \cdot a_i$$

$$b_j \leftarrow b_j - \eta \cdot \delta_j$$

E.g., to update the weight w_{31} and the bias b_2:

$$w_{31} \leftarrow w_{31} - \eta \cdot \delta^L \cdot a_1$$

$$b_2 \leftarrow b_2 - \eta \cdot \delta^L$$

By repeating these steps for many iterations, the neural network learns to minimize the loss function and accurately map inputs to outputs.

Single vs. Multiple Output Neurons

Neural networks can be designed with different output structures depending on the task:

- **Single Output Neuron:** Used for binary classification tasks. Typically employs a sigmoid activation function at the output layer to produce probabilities.
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$
- **Multiple Output Neurons:** Used for multi-class classification tasks. Employs the softmax activation function at the output layer to produce a probability distribution over classes.
$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

In summary, neural networks are a flexible and powerful model for learning complex patterns in data. Understanding the theoretical underpinnings such as the universal approximation theorem, and practical aspects like feed-forward processing and backpropagation, is essential for leveraging their full potential in various applications. However, traditional neural networks, such as Multi-Layer Perceptrons, face challenges when dealing with high-dimensional data, particularly images and spectral data, due to their lack of inherent spatial structure. Convolutional Neural Networks (CNNs), which we will dive into as the next topic, address these limitations by utilizing convolutional layers that can efficiently capture spatial hierarchies and patterns within the data, making them highly effective for image recognition, signal processing, and analysis of spectral data such as MALDI-TOF MS.

Convolutional Neural Networks

It is important to note that while my work focuses on one-dimensional data from MALDI-TOF MS, this section will primarily use images (two-dimensional data) to explain the concepts of Convolutional Neural Networks (CNNs). The reason for this approach is that I believe images provide a more intuitive understanding of the subject due to their visual and spatial properties. The theory behind CNNs is directly applicable to one-dimensional data as well, allowing for a seamless transition of these concepts to MALDI-TOF MS data. Therefore, understanding CNNs in the context of image data will

facilitate a clearer comprehension of their application to spectral and signal data.

Feature Extraction with Convolutions

Convolutions are the fundamental building blocks of CNNs, allowing the network to extract meaningful features from the input data. By applying a set of filters (kernels) across the input, convolutions help identify patterns such as edges, textures, and more complex features in later layers [21].

Example: Consider a simple 3x3 filter applied to an image. The filter slides over the image, performing element-wise multiplication and summing the results to produce a feature map. This process helps detect specific features, such as vertical or horizontal edges. This effect is shown in Figure 8 where the input image (left) is convolved with two different kernels that detect vertical edges (middle) and horizontal edges (right).

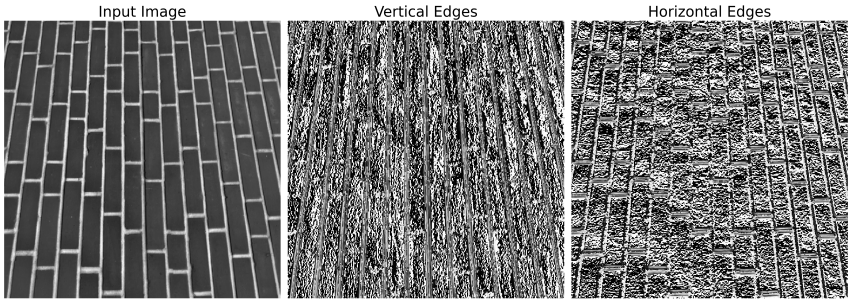


Figure 8: Illustration of feature extraction using convolutions. The input image (left) is convolved with a vertical edge detection kernel (middle) and a horizontal edge detection kernel (right). The resulting feature maps highlight vertical and horizontal edges, respectively.

Sparse Interactions

In a CNN, each neuron in a layer is connected to a small region of the input known as the receptive field, rather than the entire input. This sparse connectivity ensures that the network focuses on local patterns, making the model more efficient and reducing the number of parameters compared to fully connected layers.

Sparse interactions mean that a single convolutional filter (kernel) is applied locally, which allows the network to learn spatial hierarchies. For instance, in an image, lower layers might learn to detect simple features such as edges or textures, while higher layers might combine these features to detect more complex patterns like shapes or objects. By limiting the connections to a local region, CNNs can efficiently capture spatial dependencies.

Benefits of Sparse Interactions

- **Efficiency**: Sparse interactions significantly reduce the number of parameters in the model. For a typical fully connected layer with n input units and m output units, there are $n \times m$ parameters. In contrast, a convolutional layer with a filter size of $k \times k$ applied to an $n \times n$ input has k^2 parameters per filter. Since CNNs typically use multiple filters, the number of parameters is $k^2 \times f$, where f is the number of filters. This is still much smaller and depends only on the filter size and the number of filters, not the input size.
- **Parameter Sharing**: The same filter (set of parameters) is applied across different regions of the input, which allows the network to detect the same feature in different parts of the input. This parameter sharing leads to better generalization and reduces the risk of overfitting.
- **Local Receptive Fields**: Each neuron only focuses on a small region of the input, which enables the network to learn local features effectively. This local focus is crucial for tasks like image recognition, where spatial hierarchies are essential.

Example: Consider an input image of size 32x32 pixels. A convolutional layer with a 5x5 filter scans across the image, creating a feature map. Each neuron in this feature map is connected to a 5x5 region of the input image, resulting in a sparse interaction. If the filter moves one pixel at a time (stride of 1), the feature map will have a size of 28x28 (since the filter cannot go beyond the boundaries of the image without padding). This localized interaction helps the network learn relevant features while keeping

the number of parameters low.

By leveraging sparse interactions, CNNs are able to efficiently process high-dimensional data such as images, learning hierarchical features that are essential for complex pattern recognition tasks.

Receptive Field

The receptive field refers to the region of the input that a particular neuron is responsive to. In a convolutional neural network (CNN), neurons in a layer are connected only to a small, localized region of the input, known as the receptive field. As the network goes deeper, the receptive field increases, allowing the network to capture more complex and abstract features [21].

For example, in the first convolutional layer, each neuron might be connected to a 3x3 region of the input image (as shown in Figure 9). As we add more convolutional layers, the receptive field of neurons in higher layers becomes larger. This means that each neuron in these deeper layers is influenced by a larger portion of the original input image. Consequently, the network can learn hierarchical features, starting with simple edges and textures in the early layers and moving to more complex patterns and objects in the deeper layers.

Pooling Layers

Pooling layers are a critical component of convolutional neural networks (CNNs). They perform downsampling operations that reduce the spatial dimensions (width and height) of the input feature maps while retaining important features [22]. Pooling serves several key purposes in CNNs:

- **Dimensionality Reduction:** By reducing the spatial dimensions of the feature maps, pooling layers decrease the number of parameters and computational complexity of the model. This makes the network more efficient and faster to train.
- **Translation Invariance:** Pooling helps the network to become more robust to translations and distortions in the input data. By

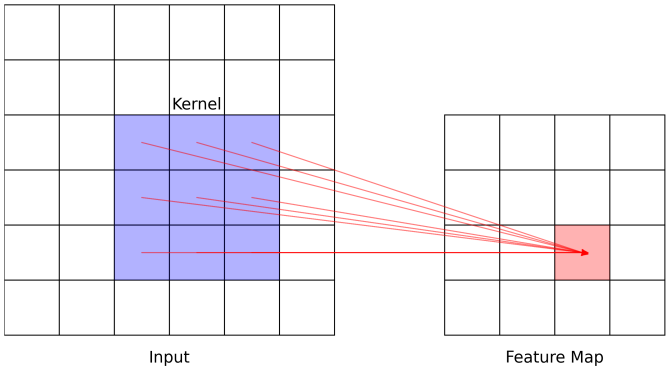


Figure 9: Illustration of the receptive field in a convolutional neural network. The input image is convolved with a 3x3 kernel (blue region) to produce a single value in the feature map (red region). This process demonstrates how each value in the feature map is influenced by a specific region of the input image, which is the receptive field of the neuron.

summarizing local regions, it ensures that small shifts or changes in the input do not significantly affect the output.

- **Overfitting Reduction:** By reducing the number of parameters, pooling layers help to prevent overfitting, especially when working with smaller datasets.

Common Pooling Methods Several pooling methods are commonly used in CNNs:

- **Max Pooling:** Max pooling selects the maximum value within a local patch of the feature map. It effectively retains the most prominent features in the local region.

$$y_{i,j} = \max\left(x_{i+k,j+l}\right), \quad \forall\, 0 \leq k, l < f$$

where f is the filter size, and x is the input feature map.

- **Average Pooling:** Average pooling computes the average value within a local patch of the feature map. It smooths the feature map and reduces noise by averaging the values.

$$y_{i,j} = \frac{1}{f^2} \sum_{k=0}^{f-1} \sum_{l=0}^{f-1} x_{i+k,j+l}$$

Example: Consider a 4x4 feature map that we apply 2x2 max pooling to with a stride of 2. The resulting 2x2 feature map will contain the maximum values from each 2x2 patch of the input feature map. This process reduces the spatial dimensions by a factor of 2 while preserving the most significant features.

In summary, pooling layers play a crucial role in reducing the spatial dimensions of feature maps, enhancing computational efficiency, and promoting translation invariance in convolutional neural networks. The most commonly used pooling methods are max pooling and average pooling, each with its own advantages depending on the application.

Fully Connected Layers in CNNs

After several convolutional and pooling layers, CNNs typically include fully connected layers. These layers integrate the features learned by the convolutional layers to make the final prediction, and perform duties similar to layers in standard NNs [21].

AlexNet

AlexNet, introduced by Alex Krizhevsky and colleagues in 2012 [23], significantly advanced the field of deep learning. It has a deeper architecture than LeNet-5, with five convolutional layers, followed by three fully connected layers. AlexNet's success in the ImageNet competition highlighted the potential of deep CNNs for large-scale image classification.

Methods

Different methodologies were employed for predicting species and resistances, and these will be described in their respective sections. However, common practices were followed for both methodologies. Specifically, Python (version 3.11) [24] was used as the programming language. All computations were performed on the GenomeDK cluster (`genome.au.dk`) at Aarhus University.

Species classification

For species classification, the data used in this thesis was sourced directly from the dataset described in Weis et al. (2022) [2, 3]. Specifically, the mass spectra were binned into 6000 fixed bins of 3 Da each, ranging from 2,000 Da to 20,000 Da, to create a consistent 6000-dimensional feature vector for each sample . This binning process ensures that each spectrum is uniformly represented, enabling the effective application of machine learning algorithms.

The preprocessing steps involved in creating the binned data include the following:

- **Mass-to-Charge Ratio Partitioning**: The m/z axis is divided into equal-sized bins of 3 Da.
- **Intensity Summation**: The intensity of all measurements falling within each bin is summed to produce the feature vector.

Prior to creating any models, the data was filtered to exclude data points that included the string ”MIX!” in their species label, as it was unclear how this label should be interpreted. Additionally, species containing fewer than five samples were excluded to ensure more reliable modeling. The data was then split into training (80%) and validation (20%) sets using a stratified split method based on species to ensure that all species were represented in both sets.

All machine learning models for species classification, including Elastic Net, Ridge, Random Forest, and SVM, were implemented using scikit-learn [25].

Resistance classification

For resistance classification, the mass spectra underwent several preprocessing steps to ensure the data was suitable for model training and prediction. These preprocessing steps included log-transformation, smoothing, normalization, and rescaling, as detailed below. These steps were originally implemented in R by PhD student Johan Kjeldbjerg Lassen.

Preprocessing steps

1. **Log-transformation**: The intensity values of the spectra were log-transformed to stabilize variance and make the data more normally distributed:
$$int_{log} = log(int + 1)$$

2. **Smoothing with LOWESS**: A two-step Locally Weighted Scatterplot Smoothing (LOWESS) procedure was applied to the log-transformed intensities:
   - **First LOWESS**: Applied with a fraction (frac) of 0.0008 and a regularization parameter (λ) of 40 to smooth out the data.
   - **Second LOWESS**: Applied with a fraction (frac) of 0.3 and a regularization parameter (λ) of 40 to further smooth and normalize the data.

3. **Rescaling**: The normalized intensities were rescaled to a range of $[-1, 1]$:
$$int_{rescaled} = 2 \left(\frac{int_{normalized} - int_{min}}{int_{max} - int_{min}} \right) - 1 \quad (1)$$

4. **Mapping to new scale**: The mass-to-charge ratio (m/z) values were mapped to a new scale ranging from 2000 to 20000 with 0.5 Da

increments using a final LOWESS smoothing with a fraction (frac) of 0.0003 and a regularization parameter (λ) of 40.

These preprocessing steps ensured that the spectra were uniformly represented and smoothed, reducing noise and enhancing signal clarity. A visualization of these steps is shown in Figure 10.

Data filtering and splitting

In order to standardize and focus the analysis on the most prevalent resistances, I identified the top 25 most frequent resistances from each dataset in the DRIAMS-A collection using a custom Python script. The script filters out entries labeled as 'MIX!' in the species column, counts the occurrences of each resistance phenotype, and selects the most common resistances. Subsequently, the script consolidates these resistances across all datasets, retaining only those resistances in the filtered datasets, thus ensuring that the final datasets include a consistent set of the most frequent resistance phenotypes for analysis.

The filtered datasets were initially divided into training/validation (90%) and testing (10%) sets. Due to the multilabel nature of the resistance classification problem, the split was performed using scikit-multilearn [26], which ensures an even distribution of label combinations across the splits (as demonstrated in Supplementary 1). The training/validation set was further divided into training (80%) and validation (20%) subsets using the same method.

Model implementations

Ridge and Random Forest classification models were implemented using sci-kit learn [25]. Convolutional Neural Network models were implemented using PyTorch [27], PyTorch Lightning [28], and Weights and Biases [29].

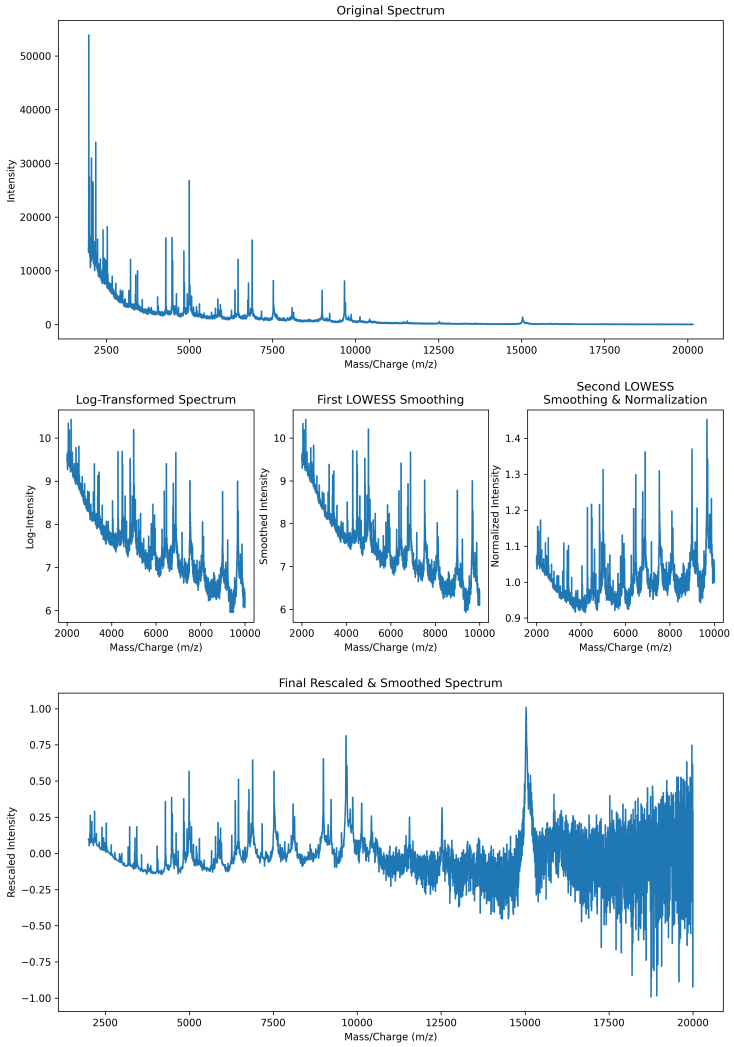


Figure 10: Illustration of the preprocessing steps applied to an original MALDI-TOF MS spectrum. The top plot shows the original spectrum. The middle row includes three subplots displaying the intermediate steps: the log-transformed spectrum (left), the first LOWESS smoothing (middle), and the second LOWESS smoothing with normalization (right). The bottom plot presents the final rescaled and smoothed spectrum. This step-by-step preprocessing enhances the data quality by reducing noise and normalizing intensity values, facilitating more effective downstream analysis.

Results

Discussion

References

[1] Christopher J L Murray et al. “Global Burden of Bacterial Antimicrobial Resistance in 2019: A Systematic Analysis”. In: *The Lancet* 399.10325 (Feb. 2022), pp. 629–655. ISSN: 01406736. DOI: 10.1016/S0140-6736(21)02724-0. URL: https://linkinghub.elsevier.com/retrieve/pii/S0140673621027240 (visited on 05/13/2024).

[2] Caroline V. Weis et al. *Database of Resistance Information on Antimicrobials and MALDI-TOF Mass Spectra (DRIAMS)*. Dryad, 2021. DOI: 10.5061/dryad.bzkh1899q. URL: https://datadryad.org/stash/dataset/doi:10.5061/dryad.bzkh1899q.

[3] Caroline Weis et al. “Direct Antimicrobial Resistance Prediction from Clinical MALDI-TOF Mass Spectra Using Machine Learning”. In: *Nature Medicine* 28.1 (Jan. 2022), pp. 164–174. ISSN: 1078-8956, 1546-170X. DOI: 10.1038/s41591-021-01619-9. URL: https://www.nature.com/articles/s41591-021-01619-9 (visited on 12/11/2023).

[4] M. I. Jordan and T. M. Mitchell. “Machine Learning: Trends, Perspectives, and Prospects”. In: *Science* 349.6245 (July 17, 2015), pp. 255–260. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aaa8415. URL: https://www.science.org/doi/10.1126/science.aaa8415 (visited on 05/15/2024).

[5] Zoubin Ghahramani. “Probabilistic Machine Learning and Artificial Intelligence”. In: *Nature* 521.7553 (May 28, 2015), pp. 452–459. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14541. URL: https://www.nature.com/articles/nature14541 (visited on 05/15/2024).

[6] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. 2nd ed. Cambridge, Massachusetts: MIT press, 2018. ISBN: 978-0-262-03924-6. URL: http://incompleteideas.net/book/the-book-2nd.html.

[7] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. "Regularization Paths for Generalized Linear Models via Coordinate Descent". In: *Journal of Statistical Software* 33.1 (2010). ISSN: 1548-7660. DOI: 10.18637/jss.v033.i01. URL: http://www.jstatsoft.org/v33/i01/ (visited on 05/18/2024).

[8] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY: Springer New York, 2009. ISBN: 978-0-387-84857-0. DOI: 10.1007/978-0-387-84858-7. URL: http://link.springer.com/10.1007/978-0-387-84858-7 (visited on 05/15/2024).

[9] Michael C. Whitlock and Dolph Schluter. *The Analysis of Biological Data*. Second edition. New York, NY: Macmillan Education, 2015. 818 pp. ISBN: 978-1-319-15421-9.

[10] Fariha Sohil, Muhammad Umair Sohali, and Javid Shabbir. "An Introduction to Statistical Learning with Applications in R: By Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, New York, Springer Science and Business Media, 2013, $41.98, eISBN: 978-1-4614-7137-7". In: *Statistical Theory and Related Fields* 6.1 (Jan. 2, 2022), pp. 87–87. ISSN: 2475-4269, 2475-4277. DOI: 10.1080/24754269.2021.1980261. URL: https://www.tandfonline.com/doi/full/10.1080/24754269.2021.1980261 (visited on 05/16/2024).

[11] Arthur E Hoerl and Robert W Kennard. "Ridge Regression: Biased Estimation for Nonorthogonal Problems". In: *Technometrics* 12.1 (1970), pp. 55–67. DOI: 10.2307/1267351.

[12] Donald W Marquardt and Ronald D Snee. "Ridge Regression in Practice". In: *The American Statistician* 29.1 (1975), pp. 3–20. DOI: 10.1080/00031305.1975.10479105.

[13] Robert Tibshirani. "Regression Shrinkage and Selection Via the Lasso". In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 58.1 (Jan. 1, 1996), pp. 267–288. ISSN: 1369-7412, 1467-9868. DOI: 10.1111/j.2517-6161.1996.tb02080.x. URL: https://

academic.oup.com/jrsssb/article/58/1/267/7027929 (visited on 05/17/2024).

[14] Hui Zou and Trevor Hastie. "Regularization and Variable Selection Via the Elastic Net". In: *Journal of the Royal Statistical Society Series B: Statistical Methodology* 67.2 (Apr. 1, 2005), pp. 301–320. ISSN: 1369-7412, 1467-9868. DOI: 10.1111/j.1467-9868.2005.00503.x. URL: https://academic.oup.com/jrsssb/article/67/2/301/7109482 (visited on 05/17/2024).

[15] Alexander McFarlane Mood, Franklin A. Graybill, and Duane C. Boes. *Introduction to the Theory of Statistics*. 3d ed. McGraw-Hill Series in Probability and Statistics. New York: McGraw-Hill, 1973. 564 pp. ISBN: 978-0-07-042864-5.

[16] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00994018. URL: http://link.springer.com/10.1007/BF00994018 (visited on 05/15/2024).

[17] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain." In: *Psychological Review* 65.6 (1958), pp. 386–408. ISSN: 1939-1471, 0033-295X. DOI: 10.1037/h0042519. URL: https://doi.apa.org/doi/10.1037/h0042519 (visited on 05/21/2024).

[18] Geoffrey E. Hinton. "Connectionist Learning Procedures". In: *Artificial Intelligence* 40.1-3 (Sept. 1989), pp. 185–234. ISSN: 00043702. DOI: 10.1016/0004-3702(89)90049-0. URL: https://linkinghub.elsevier.com/retrieve/pii/0004370289900490 (visited on 05/23/2024).

[19] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer Feedforward Networks Are Universal Approximators". In: *Neural Networks* 2.5 (Jan. 1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8. URL: https://linkinghub.elsevier.com/retrieve/pii/0893608089900208 (visited on 05/21/2024).

[20] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html (visited on 05/23/2024).

[21] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. Dec. 2, 2015. arXiv: 1511.08458 [cs]. URL: http://arxiv.org/abs/1511.08458 (visited on 05/25/2024). preprint.

[22] Rajendran Nirthika et al. "Pooling in Convolutional Neural Networks for Medical Image Analysis: A Survey and an Empirical Study". In: *Neural Computing and Applications* 34.7 (Apr. 2022), pp. 5321–5347. ISSN: 0941-0643, 1433-3058. DOI: 10.1007/s00521-022-06953-8. URL: https://link.springer.com/10.1007/s00521-022-06953-8 (visited on 05/25/2024).

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Communications of the ACM* 60.6 (May 24, 2017), pp. 84–90. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386. URL: https://dl.acm.org/doi/10.1145/3065386 (visited on 05/21/2024).

[24] Python Software Foundation. *Python Language Reference, version 3.11.0*. 2022. URL: https://www.python.org/.

[25] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

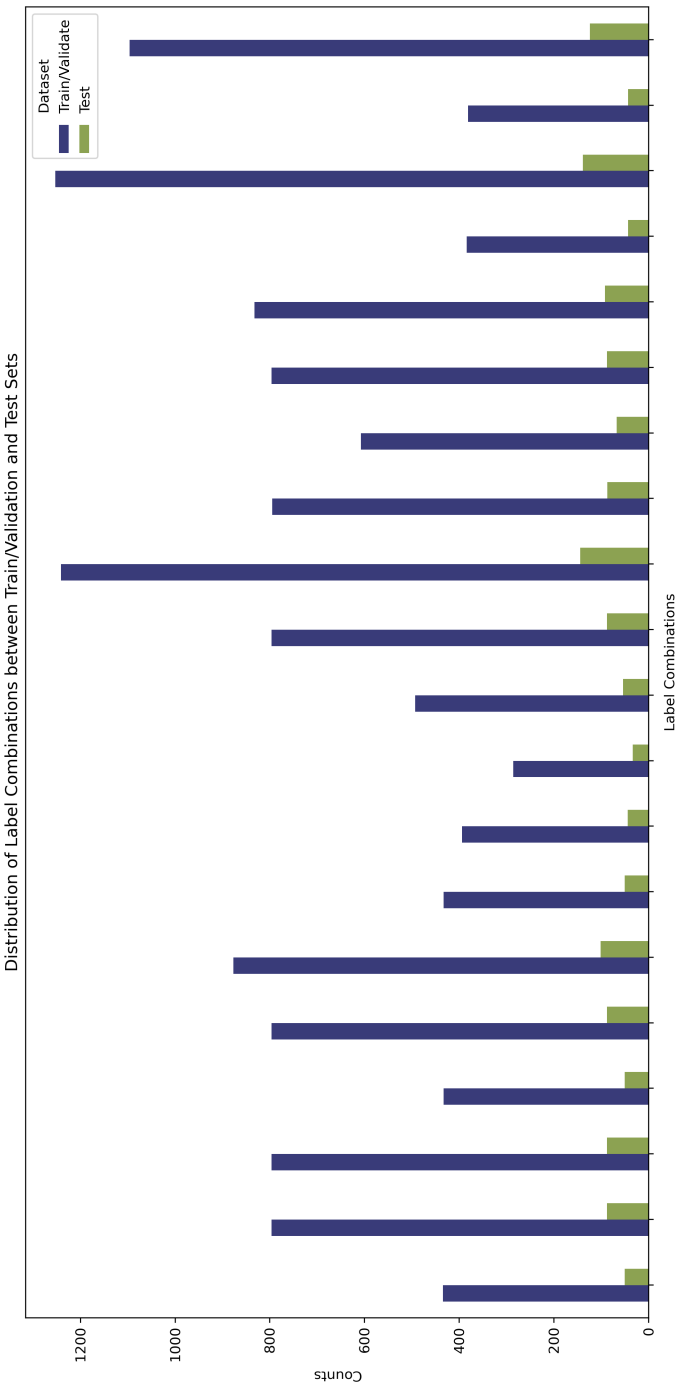
[26] P. Szymański and T. Kajdanowicz. "A scikit-based Python environment for performing multi-label classification". In: *ArXiv e-prints* (Feb. 2017). arXiv: 1702.01460 [cs.LG].

[27] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[28] William Falcon and The PyTorch Lightning team. *PyTorch Lightning*. Version 2.2.5. Apache-2.0 License. Mar. 2019. DOI: 10.5281/zenodo.3828935. URL: https://www.pytorchlightning.ai.

[29] A. Lavin and WandB Team. *Weights and Biases*. https://github.com/wandb. 2020.

Supplementary Materials



Supplementary 1: Distribution of Label Combinations between Train/Validation and Test Sets. The bar plot shows the frequency of some of the label combinations in the training/validation set (blue) and the test set (green). The x-axis represents the different label combinations, while the y-axis indicates the count of each combination in the respective datasets. This visualization demonstrates that the stratified split maintains the distribution of label combinations across the train/validation and test sets, ensuring that the datasets are representative of each other.