Development and Evaluation of a Neural Network for Food Product Classification Based on Chemical Properties

Introduction

The goal of this project was to develop a neural network classifier capable of distinguishing food products manufactured by three different companies based on their chemical properties. This task encompassed the full machine learning pipeline, including data preparation, model construction, training, and evaluation, with a focus on understanding which chemical properties influence manufacturer classification.

This journey brought together theory and real-world challenges. Creating neural networks from scratch showed me how math theory works in practice. Minor changes in parameters made big differences in results. Artificial intelligence mixes science with intuition. This project significantly enhanced my comprehension of artificial intelligence, offering me substantial growth and learning.

Code Overview and Execution

The project's code is encapsulated in a single Python file, built sequentially to align with the four key tasks: data preparation, model building, model training, and evaluation. Each task is clearly labeled in the code, making it easy to navigate and understand the logical flow. The organization reflects the various stages of the project, ensuring coherent progression from initial data processing to final analysis.

Code Structure

- **Data Preparation**: Initial functions import, process, and partition the dataset, establishing a foundation for model inputs.
- **Model Construction**: The neural network's architecture is crafted through object-oriented programming, comprising four key classes: `BasicLayer`, `ReLULayer`, `SoftmaxCrossEntropyLoss`, and `FullyConnectedANN`. `BasicLayer` linearly transforms inputs, `ReLULayer` introduces non-linearity with the ReLU function, `SoftmaxCrossEntropyLoss` calculates loss, and

`FullyConnectedANN` integrates these components, facilitating both forward and backward propagation and enabling gradient descent for training and validation processes.

- **Model Training and Validation**: This task focuses on training the neural network on a training data subset and validating its performance using a separate data subset. The model undergoes optimization through mini-batch gradient descent, with parameter adjustments aimed at enhancing accuracy. Regularization methods, built into the network's structure in Task 2, counteract overfitting, thereby boosting the model's generalization capabilities to new data.
- **Evaluation**: Leveraging the tools and data prepared in the preceding tasks, the model is trained, and its performance outcomes are visually depicted. This final step synthesizes the project's efforts, showcasing the trained neural network's capabilities and the effectiveness of the training and validation approach.

Execution

My development environment is Python3.9.7. The program does not have strict requirements for the Python environment. It only requires the following libraries to be installed: pandas, numpy, matplotlib, seaborn and sci-learn.

Please run the python script directly. The program will automatically read the data, perform data preprocessing, establish a fully connected neural network, train the network, and finally present the charts required for analysis.

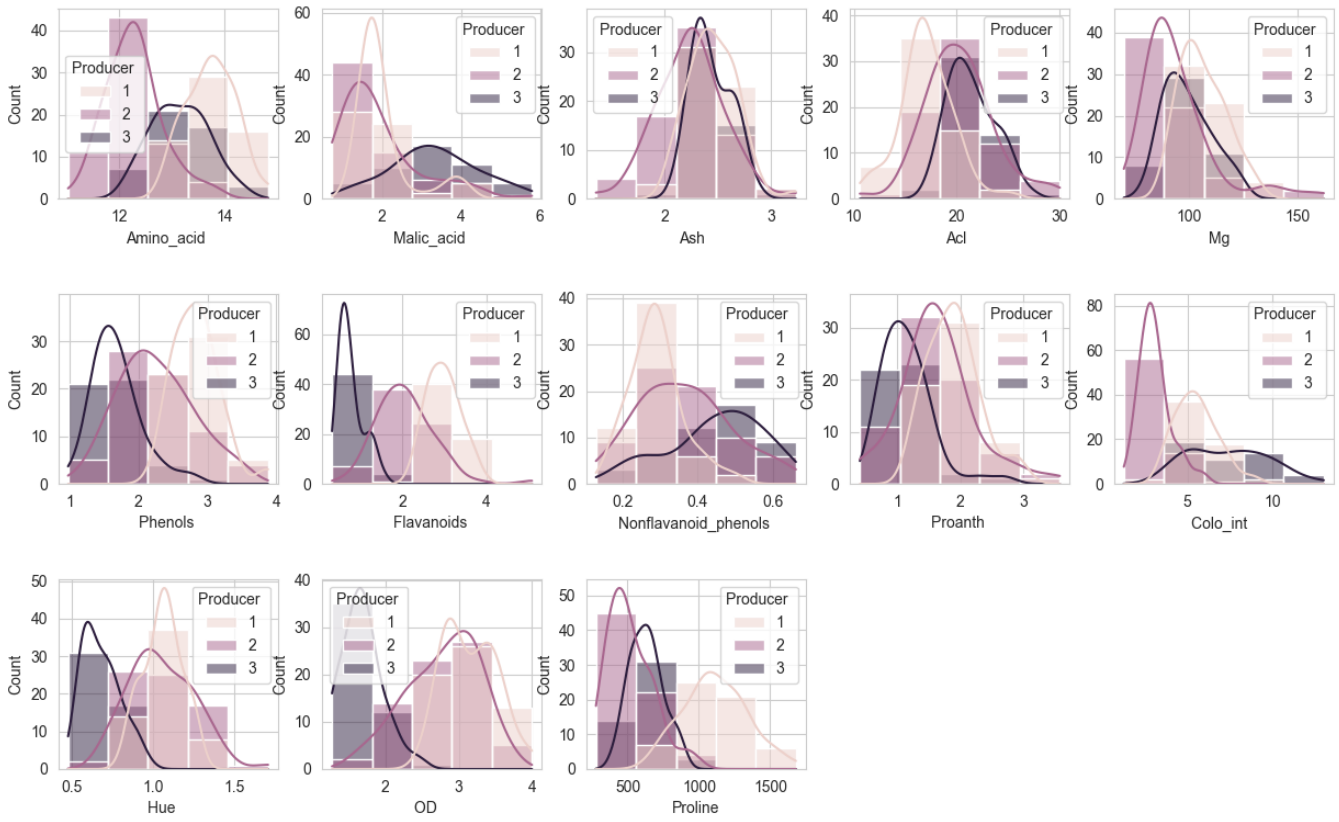
Task 1 Data Preparation

Import and verify data

The `preprocess_data` function loads the data file, preprocesses it, and returns four numpy arrays for neural network training and validation. It also provides the complete pandas dataframe and feature names for subsequent analysis.

Data files are loaded using pandas. Given the possibility of erroneous data in text files, it is critical to verify the accuracy of the data after loading. I initially checked the data type of each column in the DataFrame. If the text file contains invalid data, pandas cannot perform data conversion and the columns will be of object type. The missing object type column indicates clean data.

The function `showdata` uses seaborn to visualize the data. The data graph is like this. Everything is fine, the next step is to perform data conversion.



Dividing Data into Training and Validation Set

After the data file is loaded by pandas, `preprocess_data` splits the data into features (X) and labels (y). It uses `train_test_split` to split the data into training and validation sets with a ratio of 3:1. Setting "random_state" to 0 helps with debugging by making the splits remain the same every time.

```
df_X_train, df_X_test, df_y_train, df_y_test = train_test_split(X, y, test_size=0.25, random_st
```

Data dividing is critical to the project. The training set is used to train the model and discover patterns in the data, while the validation set is used to evaluate its performance on unseen data. This approach helps identify and mitigate overfitting, ensuring that the model generalizes well to new data.

Normalization of Feature Data

I found data features have vastly different ranges, like 'Mg' in the tens and 'Nonflavanoid_phenols' under one, normalization is critical. It ensures all features equally influence the model by scaling them to a common range, enhancing training accuracy and fairness. Furthermore, normalization improves the loss function's shape, facilitating smoother and faster convergence during optimization. Without it, skewed loss functions can lead to inefficient training and suboptimal model performance, as dominant

features disproportionately affect gradient updates. Thus, normalization is key to stable, efficient model training and optimal generalization.

Normalization parameters are established solely from training data to prevent data leakage and enhance the model's generalization to unseen data. Applying these parameters to both training and validation data ensures consistency and preserves the integrity of the evaluation process.

I use the np.linalg.norm method for normalization.

```
# Calculate the L2 norm of the training features along each feature axis
norms = np.linalg.norm(X_train, axis=0)
# Normalize training and test feature data by the calculated norms
X_train, X_test = X_train/norms, X_test/norms
```

Converting Labels to One-Hot Encoding

For this project, training a model for category recognition necessitates converting labels to one-hot encoding because it enables the model to treat each category as a separate entity, improving accuracy in classification tasks by providing a distinct binary vector for each label, thereby eliminating any implied order among categories.

```
y_train, y_test = to_one_hot(y_train), to_one_hot(y_test)
```

This project often needs to change numbers to one-hot codes and back again. So, two simple functions `to_one_hot` and `from_one_hot` were made to handle these conversions, making it easier to work with the data for training and making predictions.

Task 2 Model Construction

Loss Function

The loss function is crucial in machine learning because it tells us how wrong the model’s predictions are compared to the truth. It acts like a compass for model training, showing where the model needs improvement. By looking at the loss, we can adjust the model's internal settings, pushing it to make better guesses. It is a key tool in helping models learn from their mistakes and get better over time. Without this function, we have no clear way to fix the model's errors and help it learn.

The project involves classifying data into three categories based on the chemical properties of food products from different manufacturers. For this, I'm going to implement and utilize the softmax cross-

entropy loss function.

Uunderstanding of softmax cross-entropy loss function

- **Softmax: Transforming Network Inputs into Probabilities**
  Consider a set of logits, which are the raw output values from the network for each class. The softmax function for a specific logit z_i is defined as:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$$

  This equation takes the exponential of the logit z_i to ensure it's positive, then normalizes this value by dividing it by the sum of exponentials for all K logits. The result is a probability distribution where all the probabilities add up to one.
- **Cross-Entropy: Measuring the Difference Between Prediction and Reality**
  The cross-entropy loss function quantifies the difference between the predicted probability distribution and the actual distribution. Given a true class label (in one-hot encoding) and the predicted probability distribution, the cross-entropy loss L is calculated as:

$$L = -\sum_{i=1}^{K} y_i \cdot \log(p_i)$$

  In this formula, the sum iterates over all K classes, where y_i is 1 for the correct class and 0 for the others, and p_i is the predicted probability for class i. The loss increases when the predicted probability for the correct class is low, effectively penalizing incorrect predictions, and decreases when the predicted probability is high, reflecting more accurate predictions.
  Cross-entropy provides a mechanism for quantitatively assessing the accuracy of predictions, guiding the network to improve by adjusting the model to increase the probability assigned to the correct class. This function is fundamental in training neural networks for classification, enhancing the model's ability to classify data correctly by learning from its errors.
- **Simplified Gradient Calculation**
  During backpropagation, the method for updating the model's weights becomes surprisingly straightforward when combining softmax and cross-entropy. For any specific output, the required adjustment (the gradient) simplifies to:

$$\frac{\partial L}{\partial z_i} = p_i - y_i$$

  In multi-class scenarios, where $\mathbf{p}$ represents the model's predicted probabilities and $\mathbf{y}$ the true labels, the gradient across all outputs is merely the difference between these vectors:

$$\mathbf{d_z} = \mathbf{p} - \mathbf{y}$$

This computational simplicity stems from the logarithmic and exponential mathematical properties inherent in the softmax and cross-entropy formulas. The gradient calculation can be completed with one subtraction, which is the attraction of the softmax cross-entropy loss function.

Code implementation of softmax cross-entropy loss function

Only a few simple Python statements can implement this seemingly complex function.The calculation of the loss function and gradient calculation are completed in the forward and backward methods of the `SoftmaxCrossEntropyLoss` class.

- **Compute softmax cross-entropy loss**

```
def forward(self, logits, y_true):
    # Compute softmax output and cross-entropy loss
    exps = np.exp(logits - np.max(logits, axis=1, keepdims=True))
    self.softmax_output = exps / np.sum(exps, axis=1, keepdims=True)

    self.y_true = y_true
    epsilon = 1e-15   # Prevent log(0)
    clipped_output = np.clip(self.softmax_output, epsilon, 1 - epsilon)
    loss = -np.sum(y_true * np.log(clipped_output)) / logits.shape[0]
    return loss

def backward(self):
    # Compute gradient of loss w.r.t. softmax input (simplified due to softmax-crossentropy
    d_logits = self.softmax_output - self.y_true
    return d_logits
```

Network Design

Overview

A fully connected neural network is made up of an input layer, hidden layers, and an output layer, with activation functions linking each layer together.

Using an object-oriented programming approach, I have implemented the functionality of a fully connected neural network from scratch.

There are four main classes involved in building neural networks: `BasicLayer`, `ReLULayer`, `SoftmaxCrossEntropyLoss`, and `FullyConnectedANN`. `BasicLayer` performs a linear transformation on the input, making it general enough to be used as an input, hidden, or output layer in a network. On

the other hand, `ReLULayer` introduces nonlinearity through the ReLU function to enhance the network's ability to capture complex patterns. `SoftmaxCrossEntropyLoss` is tasked with calculating the loss, which is crucial for evaluating network performance. Finally, `FullyConnectedANN` integrates these components, supporting forward and backward passes as well as gradient descent, thereby facilitating network training and validation.

Matrix and ReLU Gradients in Neural Networks

In neural networks, understanding how to update model parameters requires grasping the gradients of both matrices (weights W and biases b) and the ReLU activation function. For a layer's output defined as $Z = XW + b$, where X represents the input matrix, W the weight matrix, and b the bias vector, the gradients are crucial for backpropagation. The gradient of the loss L with respect to the weights W is calculated as:

$$\frac{\partial L}{\partial W} = X^T \frac{\partial L}{\partial Z}$$

Similarly, the gradient with respect to the bias b is:

$$\frac{\partial L}{\partial b} = \sum \frac{\partial L}{\partial Z}$$

These equations show how adjustments to W and b are guided by the input X and the loss gradient $\frac{\partial L}{\partial Z}$.

For the ReLU (Rectified Linear Unit) function, widely used for its computational efficiency and ability to mitigate the vanishing gradient problem, the derivative is defined as:

$$\frac{\partial \text{ReLU}}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The conclusion drawn from the above derivation formula is that whether it is a matrix or ReLU, its gradient is related to its own input and the gradient of the next layer. As long as you know the gradient of the next layer and its own input, you can easily calculate its own gradient.

Forward and Backward

All four main classes implement forward and backward method. In a fully connected neural network, the "forward" process involves passing input data from the input layer through all the hidden layers to the output layer, calculating the network's prediction using the current set of weights and biases.

The "backward" process, or backpropagation, follows the forward pass. It calculates the gradient of the loss function (which measures the error of the prediction) with respect to each weight and bias in the

network, based on how much they contributed to the error. This gradient information is then used in a separate step to adjust the weights and biases, aiming to minimize the error and improve the network's predictions in future forward passes.

In the forward pass of a neural network, besides performing the necessary computations and passing the results to the next layer, it's also important to keep track of its own input. This is essential for efficiently calculating gradients during the backward pass.The following is the forward implementation of BasicLayer and ReLU

```
def forward(self, input_data):
    # Forward pass: compute weighted sum of inputs + bias
    self.input = input_data
    return np.dot(input_data, self.W) + self.b
```

```
def forward(self, input_data):
    # Apply ReLU activation: max(0, x)
    self.input = input_data
    return np.maximum(0, input_data)
```

During the backward pass, a layer uses the gradient from the next layer and its stored input to compute its own gradient, which is then utilized to adjust the model's parameters through gradient descent.The following is the backward implementation of BasicLayer and ReLU

```
def backward(self, output_gradient):
    # Backward pass: compute gradients of loss w.r.t weights and biases
    self.gradient_W = np.dot(self.input.T, output_gradient)
    self.gradient_b = np.sum(output_gradient, axis=0)
    return np.dot(output_gradient, self.W.T) # Return gradient with respect to inputs for the ne
```

```
def backward(self, output_gradient):
    # Gradient of ReLU: pass gradient where input > 0
    return output_gradient * (self.input > 0)
```

Gradient Descent

The loss function guides the direction of model training, aiming to gradually reduce its value. Mathematical derivation shows that adjusting model parameters according to the gradient can reduce the loss. A round of training starts with a forward pass, where the input is processed to produce

predictions. These predictions are compared with the actual values in the loss function to calculate the current loss. During the backward pass, the gradients of each component of the network are calculated and stored. Next, the gradient update step adjusts the parameters. This cycle is repeated, resulting in reduced losses at each round and bringing the model closer to the desired results after a few iterations.

The learning rate is a key factor in gradient descent. The new parameters of the model are calculated by subtracting the product of the learning rate and the gradient from the old values.

If the learning rate is too low, gradient descent progresses slowly, requiring many rounds for the model to reach the minimum point of the loss function. Conversely, if the learning rate is too high, the model might overshoot the optimal point. Typically, it's advisable to start with a larger learning rate and adjust it during training based on the current gradient and learning rate. Due to time constraints, I was unable to optimize the learning rate in this project and had to use a fixed learning rate for gradient descent.

The gradient descent method is implemented in the `FullyConnectedANN` class. It iterates through the model, updating parameters by subtracting the product of the stored gradients and the learning rate from the current parameters to obtain new ones.

```
def gradient_descent(self):
    # Iterate through each layer in the neural network
    for layer in self.layers:
        # Check if the layer has weights ('W'), indicating it's a layer whose parameters can be
        if hasattr(layer, 'W'):
            # Update the layer's weights ('W') by subtracting the product of the learning rate a
            layer.W -= self.learning_rate * layer.gradient_W
            # Update the layer's biases ('b') by subtracting the product of the learning rate ar
            layer.b -= self.learning_rate * layer.gradient_b
```

Module Regularisation

Module regularisation involves applying techniques like L1 or L2 shrinkage to prevent overfitting by penalizing large weights. This helps in creating simpler models that generalize better to unseen data.

The Theory of Regularisation

Overfitting occurs when a model learns not just the underlying patterns but also the noise in the training data, which can degrade its performance on new data. By adding a penalty to the loss function, regularisation discourages the model from becoming overly complex.

There are two primary types of regularisation: L1 (Lasso) and L2 (Ridge). L1 regularisation adds a penalty equivalent to the absolute value of the coefficients ($L1 : \lambda \sum |w_i|$), which can drive some coefficients to zero, leading to sparser models by excluding less important features. On the other hand, L2 regularisation adds a penalty equal to the square of the coefficients ($L2 : \lambda \sum w_i^2$), which tends to distribute weights more evenly and discourages large weights but does not eliminate them entirely, maintaining all features but with smaller coefficients.

The balance between model complexity and the ability to generalize is controlled by the regularization strength, denoted as λ, which adjusts the weight of the penalty in the overall loss function. A higher λ means a stronger push towards simplicity, potentially at the cost of increased bias but with the benefit of reduced variance.

Regularisation essentially introduces a bias-variance trade-off, promoting model simplicity to combat overfitting without significantly increasing underfitting.

Adjusting Gradients with L1 and L2 Penalties

L1 regularisation adds a penalty equal to the absolute value of the magnitude of coefficients to the loss function. The L1 penalty term is given by:

$$L1(\mathbf{w}) = \lambda \sum_i |w_i|$$

The gradient of the L1 term with respect to the weight w_i is:

$$\frac{\partial L1(\mathbf{w})}{\partial w_i} = \lambda \cdot \text{sign}(w_i)$$

where $\text{sign}(w_i)$ is a function that returns 1 if $w_i > 0$, -1 if $w_i < 0$, and is undefined (or zero in some implementations) if $w_i = 0$.

L2 regularisation adds a penalty equal to the square of the magnitude of coefficients to the loss function. The L2 penalty term is:

$$L2(\mathbf{w}) = \lambda \sum_i w_i^2$$

The gradient of the L2 term with respect to the weight w_i is:

$$\frac{\partial L2(\mathbf{w})}{\partial w_i} = 2\lambda w_i$$

Code to implentation L1 and L2

I implemented L1 and L2 regularization in the `FullyConnectedANN` by modifying its constructor, loss function, and backward method.

```
class FullyConnectedANN:
  def __init__(self, layers, loss_function=SoftmaxCrossEntropyLoss(), learning_rate=5e-3, coeff=
      self.layers = layers    # List of layers in the neural network
      self.loss_function = loss_function    # Loss function to use
      self.learning_rate = learning_rate    # Learning rate for gradient descent
      self.coeff = coeff    # Regularization coefficient
      self.reg_type = reg_type   # Type of regularization: None, L1, or L2


  def compute_loss(self, y_pred, y_true):
      # Compute the base loss using the loss function
      basic_loss = self.loss_function.forward(y_pred, y_true)

      # Initialize regularization loss
      reg_loss = 0
      # Compute regularization loss if applicable
      if self.reg_type == 'L2':
          for layer in self.layers:
              if hasattr(layer, 'W'):
                  reg_loss += 0.5 * self.coeff * np.sum(layer.W ** 2)
      elif self.reg_type == 'L1':
          for layer in self.layers:
              if hasattr(layer, 'W'):
                  reg_loss += self.coeff * np.sum(np.abs(layer.W))

      # Total loss = base loss + regularization loss
      total_loss = basic_loss + reg_loss
      return total_loss
```

```
    def backward(self):
        # Perform the backward pass to compute gradients
        loss_list = []   # Store gradients for analysis
        d_loss = self.loss_function.backward()   # Start with gradient from the loss function
        loss_list.append(d_loss)
        for layer in reversed(self.layers):     # Iterate backwards through layers
            d_loss = layer.backward(d_loss)
            loss_list.append(d_loss)

        # Apply gradient updates for regularization
        if self.reg_type == 'L2':
            for layer in self.layers:
                if hasattr(layer, 'W'):
                    layer.gradient_W += self.coeff * layer.W
        elif self.reg_type == 'L1':
            for layer in self.layers:
                if hasattr(layer, 'W'):
                    layer.gradient_W += self.coeff * np.sign(layer.W)

        dX = loss_list[-2].dot(self.layers[0].W.T)
        return dX
```

Interface for Using the Model: `Producer_model`

The function `Producer_model` provides an interface for using fully connected networks. This function passes in five parameters to build different neural network models.The specific parameters are as follows:

- layersizes: A list of tuples, where each tuple represents the input size and output size of a layer in the network.
- learning_rate: The learning rate for the gradient descent optimization algorithm.
- coeff: Coefficient for regularization.
- reg_type: Type of regularization to apply. Options are 'None', 'L1', or 'L2'. Default is 'None'.

Task 3 Model Training

Training and Validation code

Four key functions are designed to support model training and validation. They track loss and accuracy metrics for every epoch and batch throughout the training and validation phases:

- `load_mini_batches` : This function splits the dataset into mini-batches for training. It creates mini-batches using an iterator, which is useful for handling large datasets by breaking them down into smaller chunks. This improves memory efficiency and the speed of training iterations.
- `train_producer` : Performs training for one epoch over the training data. It carries out the forward pass, calculates the loss, and then performs the backward pass and gradient descent to update the model parameters.
- `validate` : Uses the model to validate on a separate dataset, assessing the model's performance on unseen data.
- `run_producer_training` : Utilizes `train_producer` and `validate` to conduct the model's training and validation cycles. It also generates analytical charts using the collected data to visualize the training and validation process.

Choosing Training Parameters

Selecting training parameters can be a daunting task for someone new to AI. It's often challenging to determine whether unsatisfactory results stem from poor parameter choices or issues with the neural network code implementation.

After testing, I've settled on the following parameters:

- A learning rate between 5e-3 and 1e-2.
- For L2 regularization, a λ value ranging from 0.03 to 0.15.
- For L1 regularization, a λ value best kept below 0.01.

In selecting these parameters, I sometimes feel like an alchemist from the medieval ages.

Conducting Training and Validation: Data Collection for Task 4 Analysis

I conducted three training and validation sessions using the code below, gathering data for analysis in Task 4.

```
lr = 1e-2
batch_size = 10
num_epochs = 1000
coeff = 0
run_producer_training(num_epochs, lr, batch_size, layersizes, reg_type="None", coeff=coeff)
coeff = 0.08
run_producer_training(num_epochs, lr, batch_size, layersizes, reg_type="L2", coeff=coeff)
coeff = 0.008
run_producer_training(num_epochs, lr, batch_size, layers
```

The first model is a standard fully connected network, the second model incorporates L2 regularisation, and the third model utilizes L1 regularisation. The principles and implementations of regularisation are detailed in the Task 2 section, and will not be repeated here.

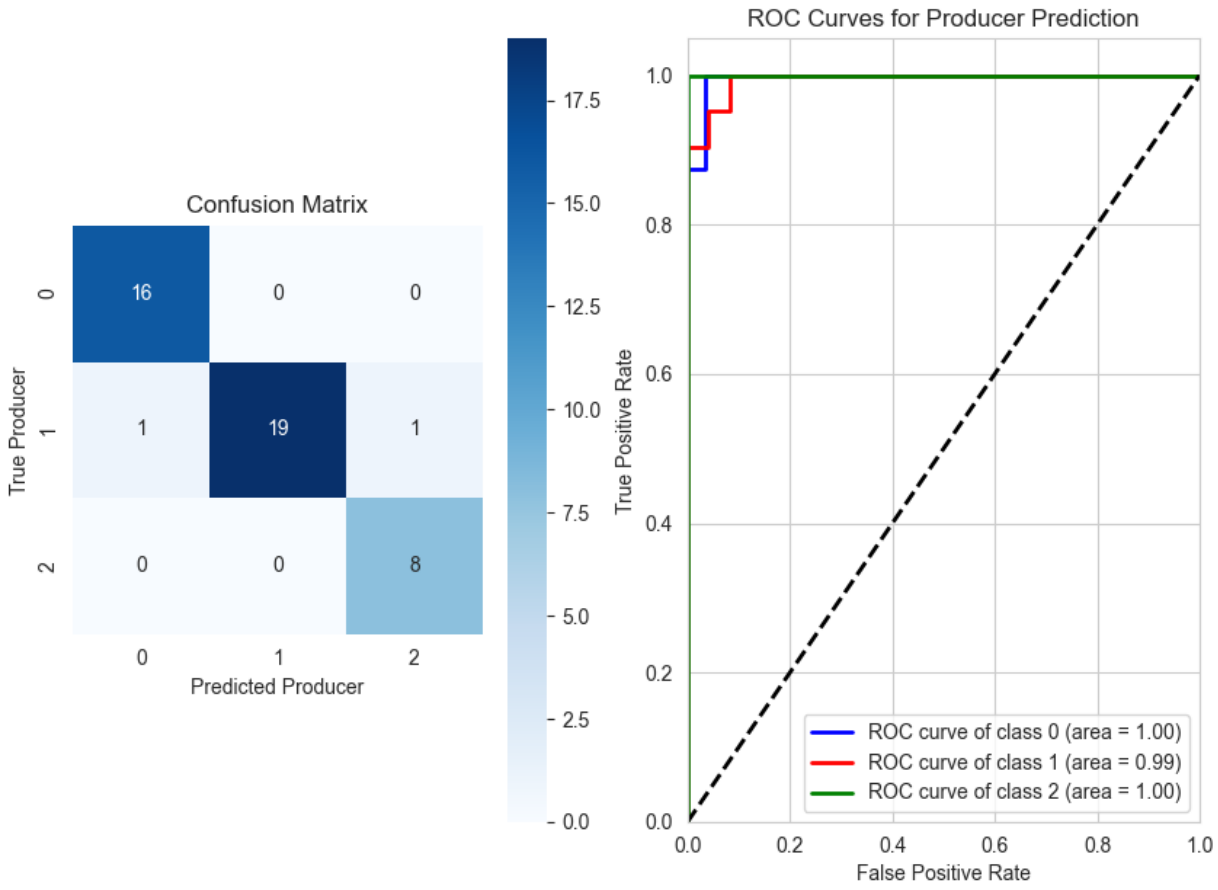
Task 4 Evaluation

Present Result

In Task 3, I trained three models with the same structure: an input layer (13, 128), a hidden layer (128, 128), and an output layer (128, 3). The first model(Model 1) was a standard fully connected network. The second model(Model 2) incorporated L2 regularization, and the third model(Model 3) utilized L1 regularization. After training, the model was evaluated using validation data and the following report and graphs were produced.The accuracy rates on the validation data for the first, second, and third models were 96%, 93%, and 96%, respectively.

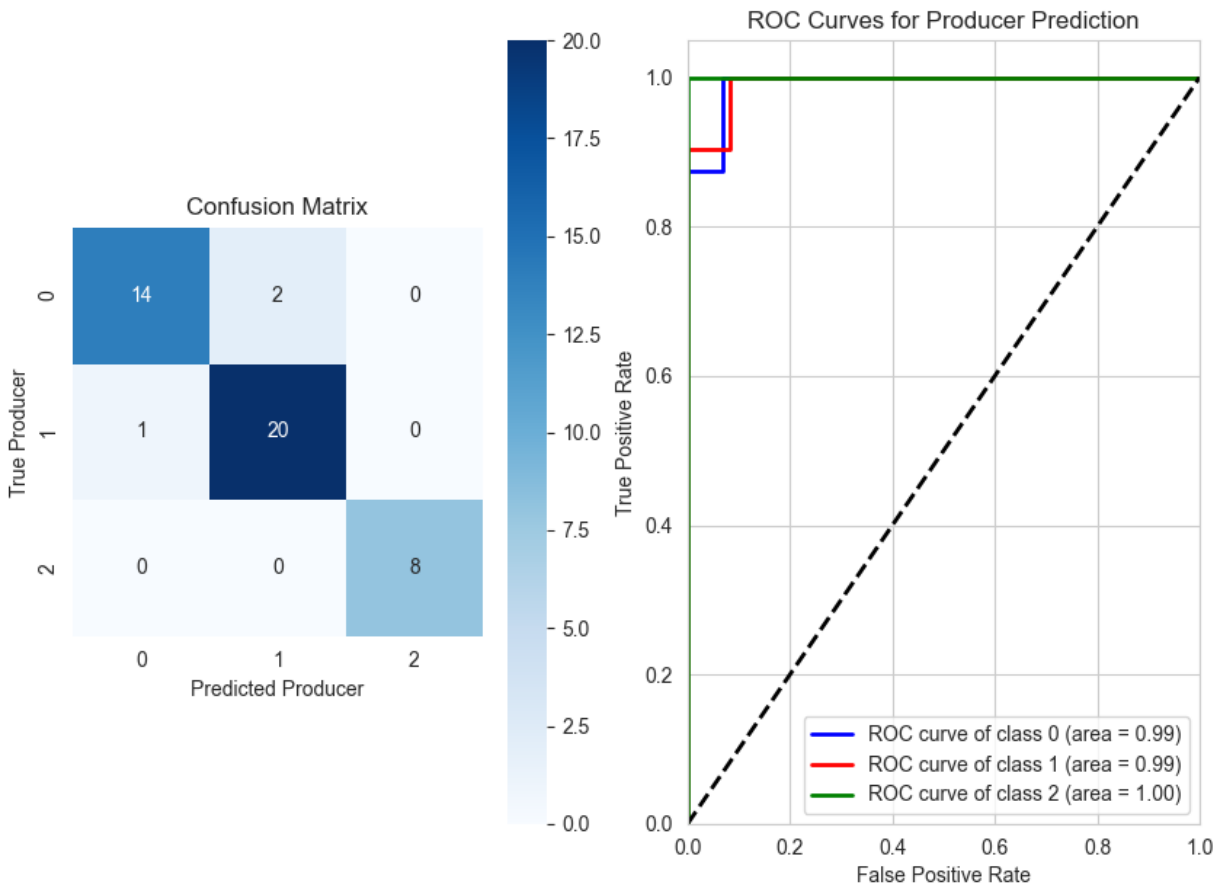
- Model 1 result

	Precision	**Recall**	**F1-score**	**Support**
1	0.94	1.00	0.97	16
2	1.00	0.90	0.95	21
3	0.89	1.00	0.94	8
Accuracy			0.96	45
Macro avg	0.94	0.97	0.95	45
Weighted avg	0.96	0.96	0.96	45



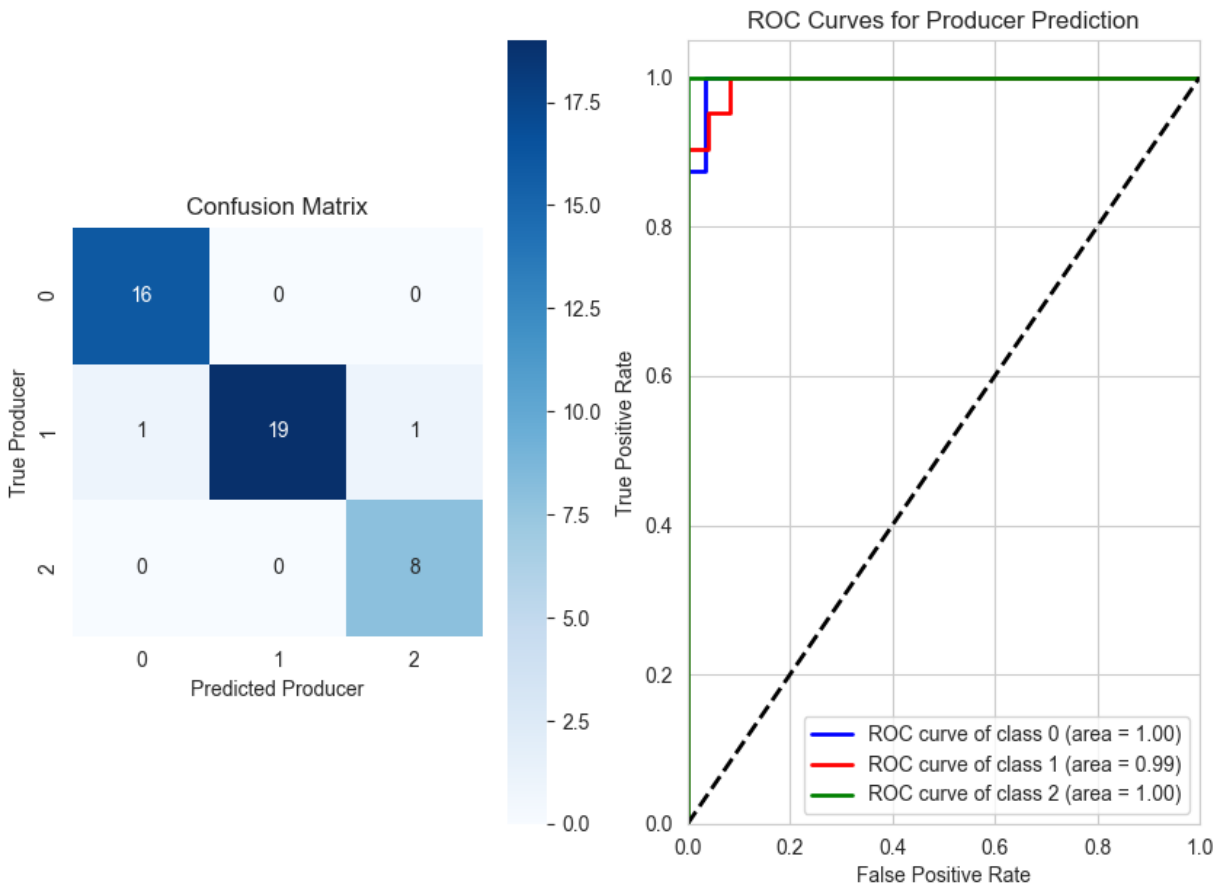
- Model 2 result

	Precision	**Recall**	**F1-score**	**Support**
1	0.93	0.88	0.90	16
2	0.91	0.95	0.93	21
3	1.00	1.00	1.00	8
Accuracy			0.93	45
Macro avg	0.95	0.94	0.94	45
Weighted avg	0.93	0.93	0.93	45



- Model 3 result

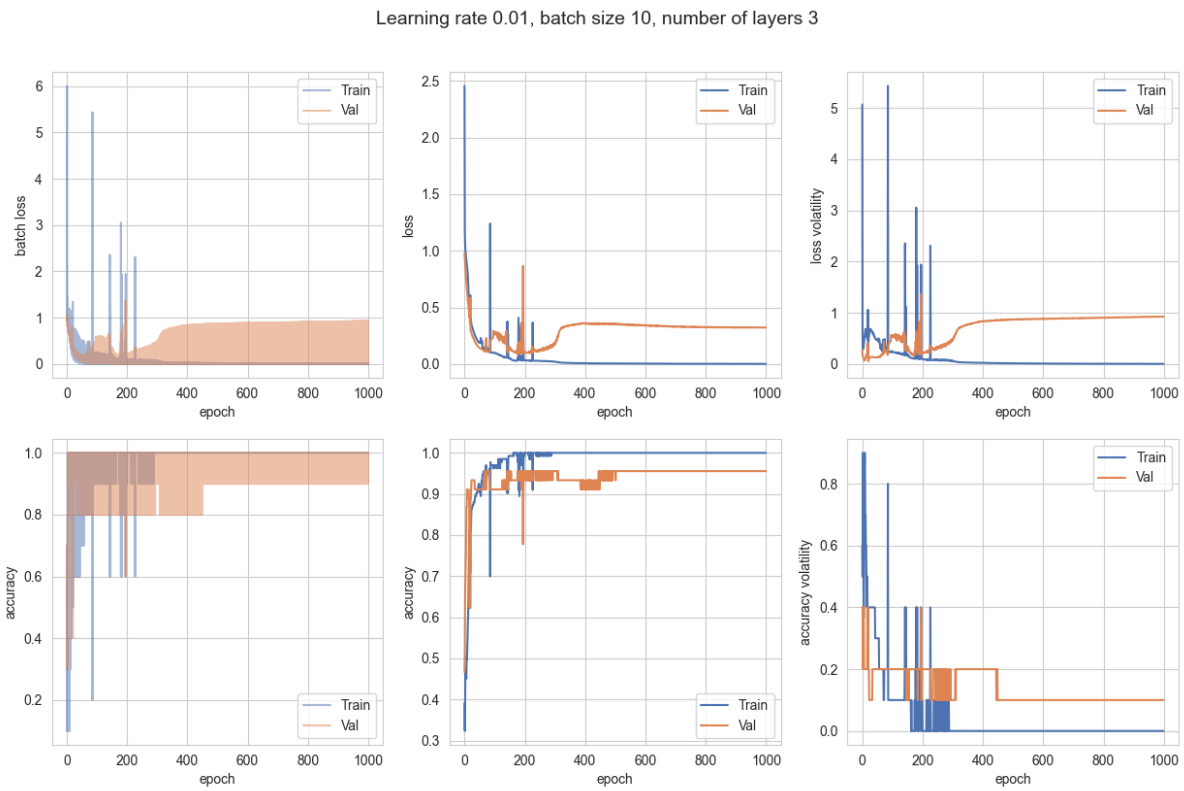
	Precision	**Recall**	**F1-score**	**Support**
1	0.94	1.00	0.97	16
2	1.00	0.90	0.95	21
3	0.89	1.00	0.94	8
Accuracy			0.96	45
Macro avg	0.94	0.97	0.95	45
Weighted avg	0.96	0.96	0.96	45



Plot Result

During each training session, the program records the loss and accuracy for every batch. It also tracks the fluctuations in these metrics, referred to as 'loss volatility' and 'accuracy volatility', which represent the differences between the highest and lowest values within each epoch for both the training and validation phases. Post-training, the results display loss, accuracy, and volatility metrics, visualized per epoch and per batch for comprehensive analysis.Each graph includes annotations of the training parameters for clarity and reference.

Analysis of Unregularized Model Graphs

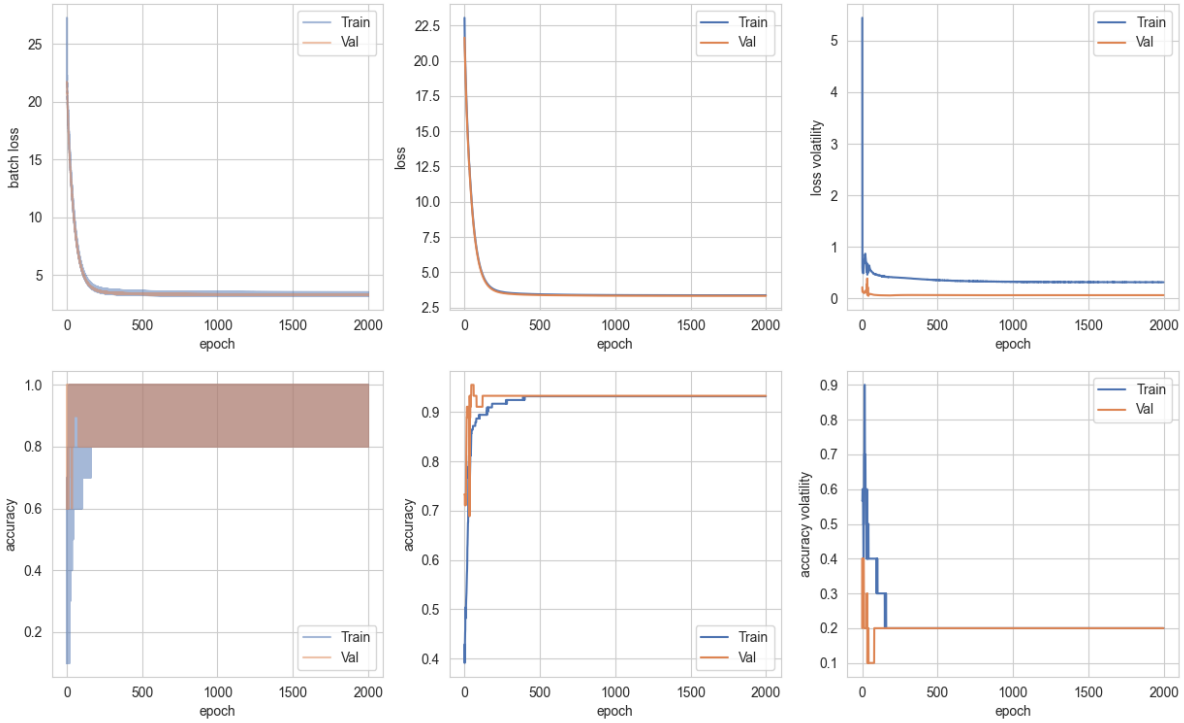


Analyzing the provided graphs reveals key insights into the model's performance:

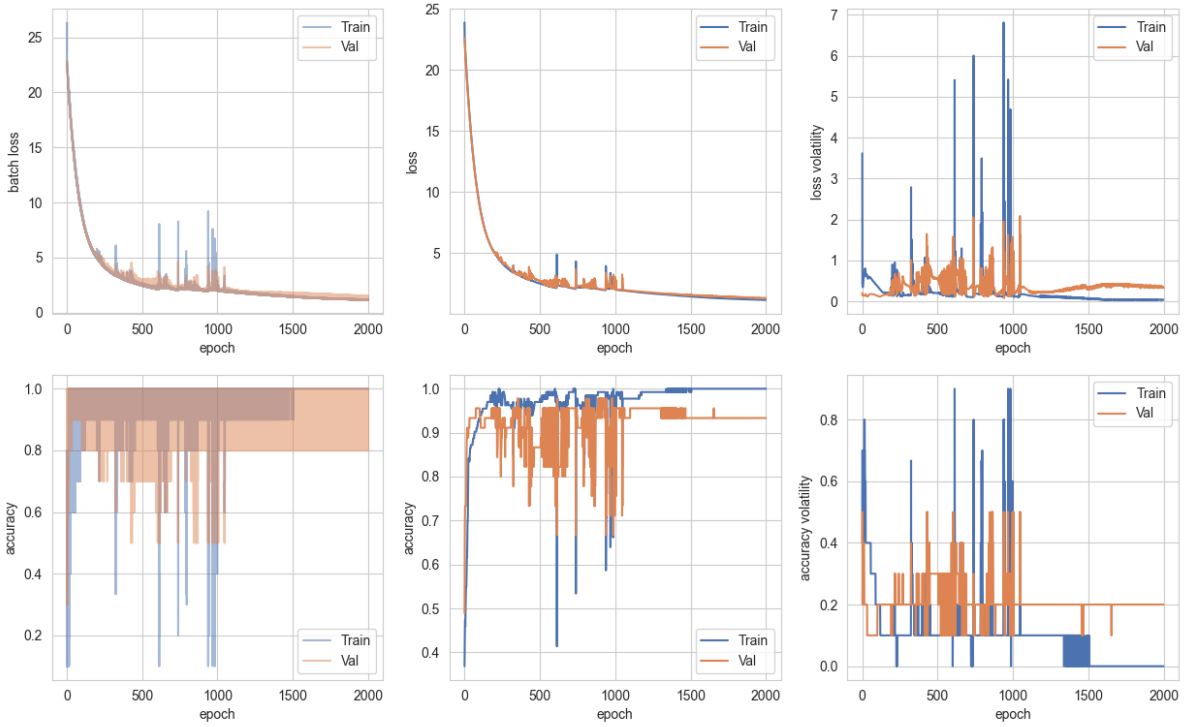
1. The model's training loss decreases steadily, approaching zero, which indicates that the model is effectively learning from the training set.
2. The validation loss also decreases initially but begins to rise after around 250 epochs, suggesting that the model starts to overfit to the training data.
3. Overfitting is further evidenced by the training accuracy reaching near 100%, while the validation accuracy fails to improve and shows significant fluctuations.
4. The apparent overfitting means the model, despite performing well on training data, does not generalize as effectively to the unseen data in the validation set.

Analysis of L1 and L2 Regularized Model Graphs

Learning rate 0.01, batch size 10, number of layers 3, Ridge with with regularization coefficient λ = 0.08.



Learning rate 0.01, batch size 10, number of layers 3, Lasso with with regularization coefficient λ = 0.01.



Analyzing the two sets of graphs for the models with L1 and L2 regularization, the following observations and conclusions can be made:

1. For both regularized models, the validation loss does not show an upward trend, suggesting that regularization is effectively preventing overfitting.
2. The validation accuracy plateaus and remains comparable to the unregularized model, implying that regularization has maintained the model's ability to generalize without necessarily increasing accuracy.
3. The L1-regularized model's loss shows spikes, which may be due to the absolute value operation in L1, which is non-differentiable at zero. This could cause instability during training, an issue that needs further research to resolve.
4. The L2-regularized model's loss curve is smoother, likely due to the squared operation in L2 regularization, which is differentiable everywhere and thus provides stable training dynamics.

Explain Results

Model Performance Interpretation:

For predicting classifications, the three models show consistent performance. Here are my thoughts:

- For Producer 1, the model almost always correctly identifies its products (high precision), and it doesn't miss any (perfect recall). So, it's very reliable for this producer.
- Producer 2's products are always identified correctly when the model says they are from Producer 2 (perfect precision), but the model misses some of their products (recall is less than perfect). This might mean that some of Producer 2's products are a bit harder to distinguish.
- For Producer 3, the model sometimes mistakes other manufacturers' products for Producer 3's (lower precision), but it is excellent at catching all of Producer 3's actual products (perfect recall).
- The overall accuracy is 96%, it is good. The F1-scores are also high, which means the balance between not missing products (recall) and not mislabeling them (precision) is good.
- All three ROC curves are very close to the top-left corner, which indicates excellent performance.

There is room to improve in distinguishing Producer 2's products a bit better and in reducing the mislabeling of Producer 3's products.

My Insight into Overfitting and Model Regularization

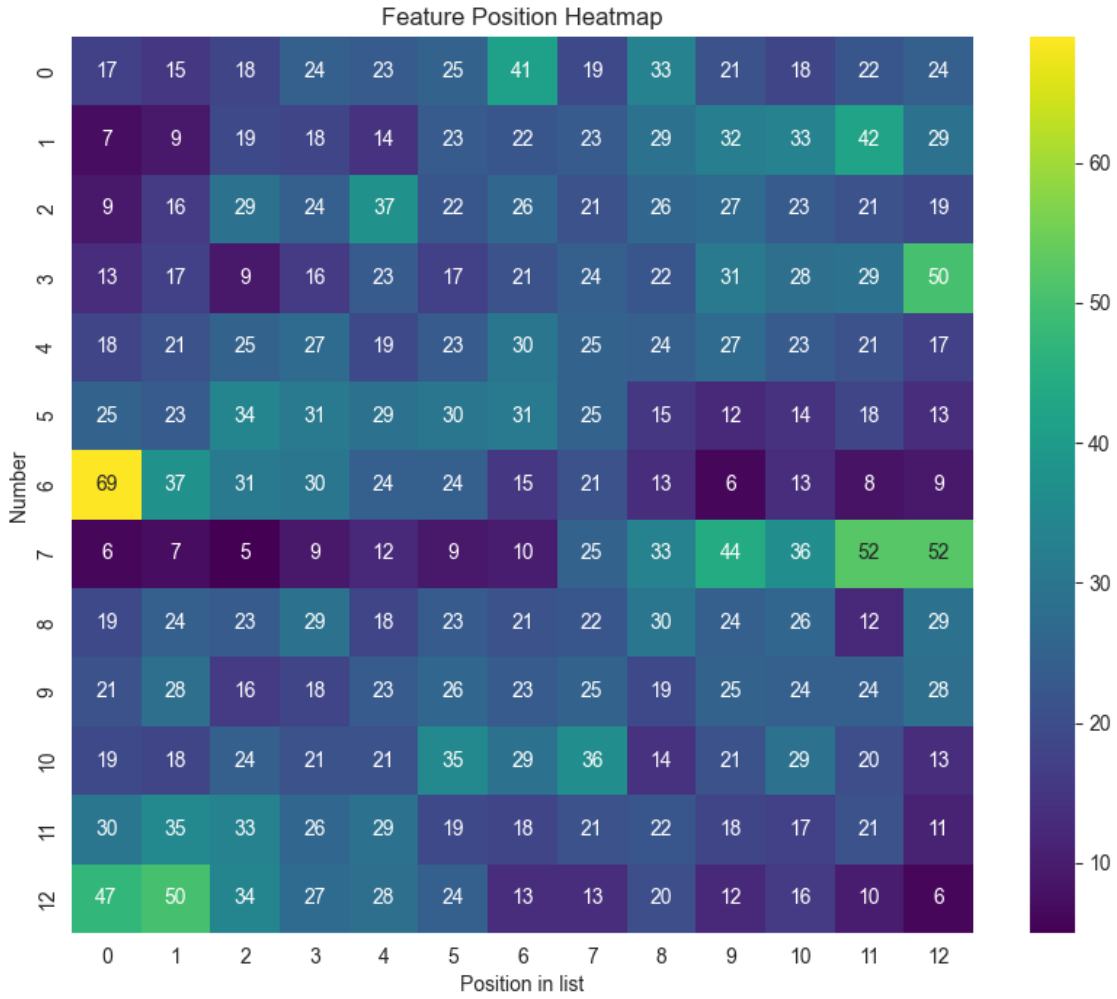
- **Causes of Overfitting**: Overfitting occurs when a model learns the training data too well, capturing not only the genuine features but also the noise or random fluctuations. This happens especially when the training data is limited, allowing the model to quickly learn most or all of the features, which is reflected in the rapid decrease in training loss. However, this over-specialization

means the model doesn't perform as well on unseen data because it has essentially memorized the training set rather than learning the underlying patterns applicable to new data.

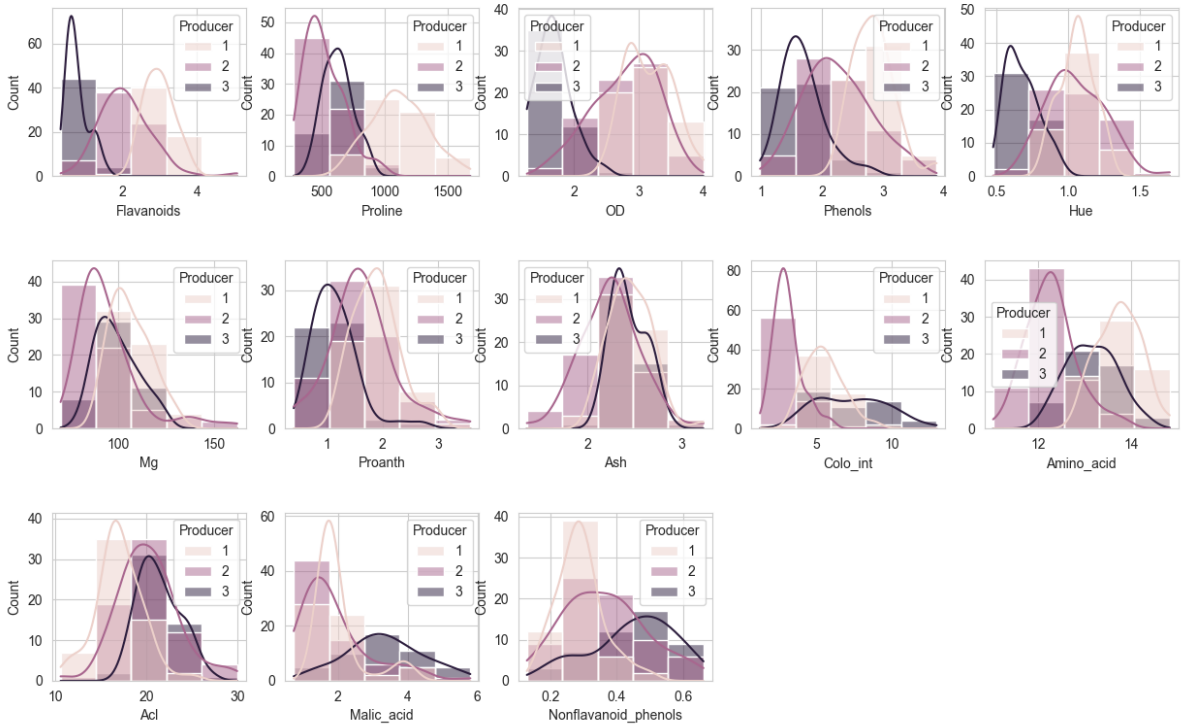
- **Regularization: Balancing Model Adaptability**: Introducing L1 or L2 regularisation adds constraints to parameter adjustments during training. Firstly, the original loss function aims to minimize the difference between predictions and actual targets. Secondly, regularization imposes penalties on the sum of absolute values or squares of the parameters. These dual constraints limit the freedom of gradient descent in altering parameters, allowing only meaningful features to influence the model, thereby enhancing its adaptability and generalization.

Identifying Key Features L1 Regularized Models

To determine which features are most critical for classification, I sought information on the internet and learned that examining the size of parameters or gradients at the model's input layer could indicate feature importance. However, each training session yielded different results due to the model being initialized with random values, leading to various outcomes based on these initial settings. To identify the most important features, I trained an L1 regularized model 300 times, conducting 250 epochs for each session. After obtaining a model, I recorded the size of the input layer's parameters and the cumulative gradient values, organizing and storing this data by rank. Finally, I used this information to create heatmaps. The heatmaps, based on the ranked cumulative gradient values, effectively illustrated the significance of each feature.



From the heatmap, it's observed that features 12 and 6 are the most influential factors affecting classification. Feature 12 corresponds to 'Proline', which measures the proline amino acids content, and feature 6 relates to 'Flavanoids', measuring the flavonoid phenols in the product. These two features stand out as the most significant in determining the classification outcome.The figure below sorts the features by their importance and shows the data distribution of each feature.Some features show almost no overlap between producers (like "Flavanoids" and "Proline"), which means they could be very good predictors for identifying the producer.



Future work

This project has been a meaningful journey. I've made significant progress from building a neural network from scratch to training it and interpreting its output. Going forward, my goal is to spend the next few weeks perfecting the L1 regularized gradient algorithm in my spare time. Additionally, I will delve into scientific methods for identifying key factors that influence classification outcomes.