MACHINE LEARNING

Software Project 2, Components 2 & 3

Jose Luis Galiano Gómez

DESCRIPTION AND ANALYSIS OF THE USED DATA

As mentioned in the previous component, we will use a sizable yet simple enough labeled dataset. It contains data extracted from images that were taken from genuine and forged banknote-like specimens. The goal is to classify each banknote, represented by a data point containing 4 features, as either authentic or inauthentic.

We're looking at 1372 instances, 4 features each, 5 including the class feature which acts as the label. We will now perform a quick analysis of the data using the Weka Explorer tool (the data file was previously converted to a .arff file).

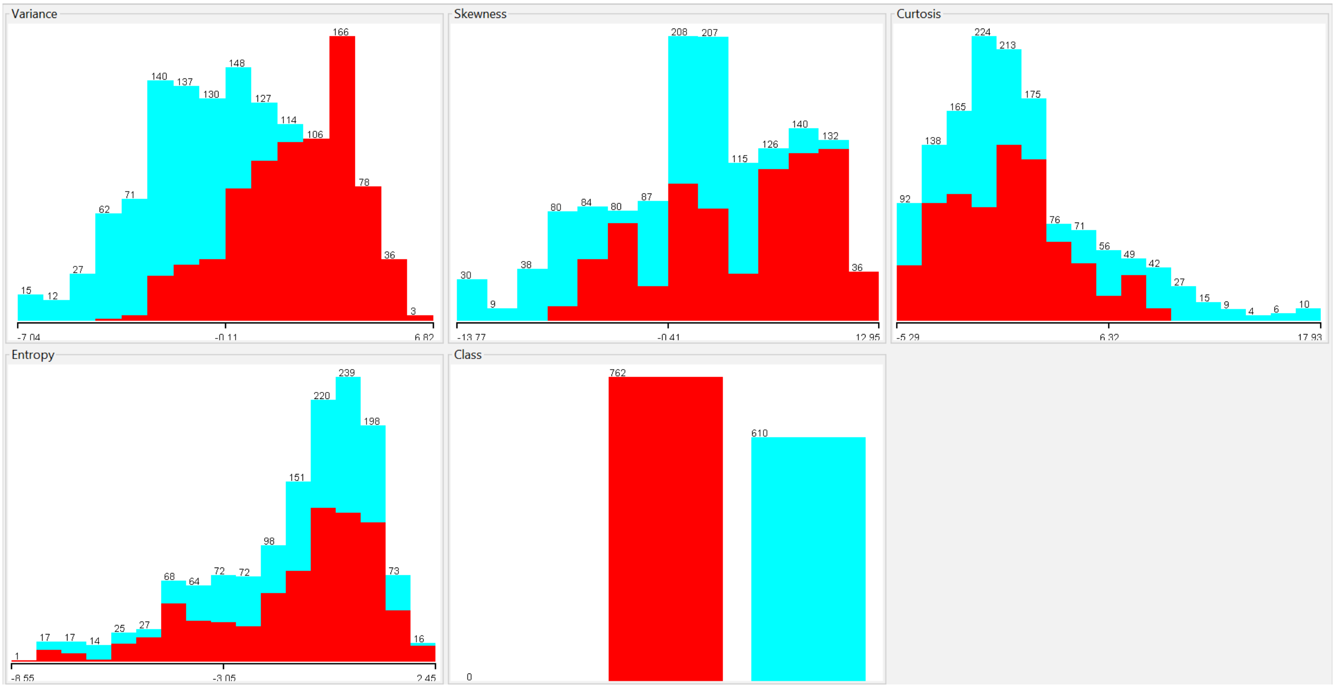


Figure 1: Distribution of values for each attribute as well as class for each data point (red is authentic, blue is inauthentic)

Firstly, we verify that there are no missing values by applying the filter "ReplaceMissingValues" that Weka provides (which replaces all missing values for nominal and numeric attributes in a dataset with the modes and means from the training data). The operation doesn't change any value in the dataset, so we can conclude there are no missing values. We do the same for duplicates; we apply the "RemoveDuplicates" filter. This time however there were some duplicates, since the number of instances drops from 1372 to 1348.

Finally, we can apply the filter "InterquartileRange", which detects outliers and extreme values based on interquartile ranges. After applying it, two more nominal attributes are added to the dataset: outlier and extreme value. These can either be "yes" or "no". After applying the filter to the dataset, we see that there are only 5 instances classified as outliers and no extreme values at all. Since there are so few outliers, the performance of the model should not be affected much by their presence. However, we will still have to consider whether to eliminate them or not. This will depend on the training results; for now, we will keep a dataset with them and another version where they are removed.

FEATURES USED IN LEARNING

The dataset presents four features, all of them real numbers (continuous and can be negative). These features are different values obtained after performing a Wavelet Transform, a mathematical technique used in signal processing and image analysis to decompose a signal or an image into different frequency components. The features are the variance (statistical measure of the spread or dispersion of pixel intensity values), skewness (measure of asymmetry in the intensity values), curtosis (quantify the departure from normal distribution in the distribution of pixel intensities or values), and entropy (quantifies the average amount of information needed to describe the content of a signal or an image) of a given transformed image.

We can check the correlation between each feature and the class in Weka using the "CorrelationAttributeEval" evaluator along with the "Ranker" search method. The correlation values are as follows:

1. Variance: 0.7338
2. Skewness: 0.4458
3. Curtosis: 0.1431
4. Entropy: 0.0312

These values suggest that as the variance increases, the likelihood of belonging to the class (0, authentic banknote) significantly increases. This can easily be seen in the plot of Figure 1 for the variance attribute too. As for the rest of the attributes, there isn't a strong enough correlation to claim a linear relationship between the feature and the class. However, we can further examine Figure 1 and notice that for instances presenting skewness values under -6.9321 (the exact value was extracted with the visualize tool) or curtosis values over 8.8294 the bankote will never be authentic. Given the low correlation and lack of insight gained by its plot against the class, we can safely asume the entropy feature to be independent of the class.

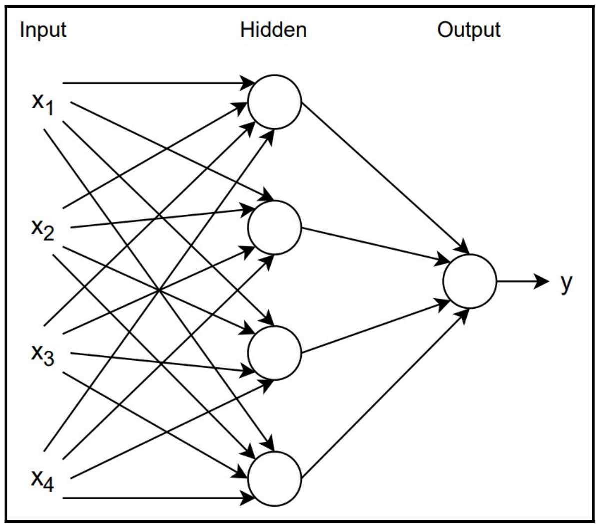
PROPOSED SOLUTION: MULTILAYER PERCEPTRON

The ML model we will be using for this classification model is a multilayer perceptron, a type of artificial neural network that consists of multiple layers of nodes, also known as neurons or perceptrons. The MLP is a feedforward neural network, meaning that information flows through the network in one direction—from the input layer to the output layer—without any cycles or loops.

The MLP architecture consists of the input, output, and hidden layers. The input layer receives the initial input data, each node in it representing a feature. The hidden layers are one or more layers between the input and output layers. Each node in a hidden layer performs a weighted sum of its inputs, applies an activation function, and produces an output. The presence of multiple hidden layers allows the network to learn complex relationships in the data. Finally, the output layer produces the final output of the network. The number of nodes in the output layer depends on the nature of the task, such as binary classification, multi-class classification, or regression. In our case, binary classification, one node suffices.

Each node from each layer is connected to all nodes in the following layer. Connections between nodes have associated weights that determine the strength of the connections. Additionally, each node typically has a bias term. During training, these weights and biases are adjusted to minimize the difference between the predicted and actual outputs. Additionaly, each node in the hidden layers and output layer applies an activation function to introduce non-linearity into the model. Common activation functions include sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU).

As for the specifics of how the architecture of our MLP will be, we will have an input layer of 4 nodes, since our dataset presents 4 features, a single hidden layer of 4 nodes too, and an output layer of 1 node. The activation function of choice will be the sigmoid.



Due to the simplicity of the task, we speculate that one hidden layer will be enough to get satisfactory enough results. However, if the experimental results are not good enough a second hidden layer might be implemented. The same can be said about the activation function; depending of the performance it might change from sigmoid to a different one.

Another important hyperparameter (the configuration settings that are not learned from the training data but need to be set before the training process begins) is the learning rate, which determines the step size at each iteration during the optimization process. It influences how much the model's weights are updated in the direction that minimizes the error. A higher learning rate may cause the model to converge faster, but it could overshoot the optimal weights. On the other hand, a lower learning rate may lead to slower convergence but more precise weight adjustments. A commonly used value and our value of choice will be 0.1, although it should be easy to change its value (in terms of code it will amount to changing the value of one variable) and choose the best learning rate based on performance once we get to the implementation phase.

As for the learning algorithm used to update the weights of the neural network during training, backpropagation is a popular choice for MLPs. Backpropagation computes the gradient in weight space of a feedforward neural network, with respect to a loss function. The loss function we will use is the Binary Cross-Entropy Loss, suitable for binary classification. The optimization algorithm chosen to minimize the loss function is Stochastic Gradient Descent (SGD). The specifics of the three will be detailed later in the learning algorithm section.

TARGET FUNCTION

The target function we want the model to learn, as in any binary classification problem, is simply

$$f(x) = y \in \mathbb{B}$$

where x is the data instance and y is the classification. As for the representation of the learned function that approximates the target function, the function uses the weights and biases along with the input features to produce the output of the network. First, let’s take a look at the output of a single perceptron:

$$\overline{y}(x) = \sigma(\sum_{i=1}^{4} w_i \cdot x_i + b)$$

where *x* is a vector containing the 4 inputs (in the case of a neuron in the hidden layer, these would be the features of the data instance, for the neuron in the output layer (or second hidden layer, if there were one), these would be the output of the 4 neurons in the previous layer), w is a vector containing the weights of the 4 connections, and b the bias assigned to the particular neuron. The weighted sum of inputs and bias is then squashed into the range [0,1] by the sigmoid activation function, and that value is the output of the neuron.

The learned function of the entire network is the result of composing the transformation through each layer; for each layer, the weighted sum of inputs plus the bias is passed through the activation function, and the result is used as the input for the next layer. This process is repeated until the output layer is reached, and the final prediction is obtained. As such, the mathematical representation of the learned function would be the following:

$$\overline{y}(x) = \sigma(W_o \cdot \sigma(W_h \cdot x + b_h) + b_o)$$

Where W_h is a 4x4 matrix with the weights for the connections between the input layer and the hidden layer, W_o is a 4x1 matrix with the weights for the connections between the hidden layer and the output layer, b_h is the 4-dimensional bias vector of the hidden layer and b_o is the bias of the neuron in the output layer.

LEARNING ALGORITHM

The learning method involves several steps: forward propagation, computing the loss (using the binary cross-entropy loss function), backward propagation, and finally updating the weights (using SGD). This process is repeated until convergence.

For the sake of clarity, we will go over all of the notation before showing the algorithm:

· $\boldsymbol{x}$ represents the 4-dimensional input vector containing the input features.
· $\boldsymbol{W_h}$ and $\boldsymbol{W_o}$ are the weight matrix for the hidden layer (4x4) and the output layer (4x1) respectively.
· $\boldsymbol{b_h}$ and $\boldsymbol{b_o}$ are the 4-dimensional bias vector of the hidden layer and the bias of the output layer, respectively.
· $\boldsymbol{z_h}$ and are $\boldsymbol{a_h}$ the linear and activated (binary) outputs for the hidden layer, respectively. $\boldsymbol{z_o}$ and $\boldsymbol{a_o}$ are the same for the output layer. It's important to note that for the hidden layer these are 4-dimensional vectors of binary values and for the output layer one single binary value.
· $\boldsymbol{\sigma(z)}$ is the sigmoid activation function.
· $\boldsymbol{y}$ is the true label.
· $\odot$ represents element-wise multiplication (or Hadamard product).
· The derivative of the sigmoid function with respect to *z*, denoted by $\boldsymbol{\bar{\sigma}(z)}$, is given by:
$\bar{\sigma}(z) = \sigma(z) \cdot (1 - \sigma(z))$
· $\boldsymbol{\eta}$ is the learning rate.

The steps of the training are:

1. Initialization: The weights and biases are initialized randomly with values between 0 and 1.

2. Forward Propagation: The linear and activated outputs for each layer are computed:

$$z_h = x \cdot W_h + b_h \quad z_o = a_h \cdot W_o + b_o$$
$$a_h = \sigma(z_h) \qquad a_o = \sigma(z_o)$$

3. Compute Loss: The binary cross-entropy loss is computed:

$$L = -(y \cdot \log(a_o) + (1-y) \cdot \log(1-a_o))$$

4. Backward Propagation: The gradients of the loss with respect to the parameters is computed:

$$dz_o = a_o - y \qquad dz_h = (W_o)^\intercal \cdot dz_o \odot \overline{\sigma}(z_h)$$
$$dW_o = a_h^\intercal \cdot dz_o \qquad dW_h = x^\intercal \cdot dz_h$$
$$db_o = dz_o \qquad db_h = dz_h$$

5. Update parameters: The weights and biases are updated using gradient descent:

$$W_o = W_o - \eta \cdot dW_o$$
$$b_o = b_o - \eta \cdot db_o$$
$$W_h = W_h - \eta \cdot dW_h$$
$$b_h = b_h - \eta \cdot db_h$$

6. Repeat: Steps 2-5 are repeated until convergence.

DESIGN OF THE APPLICATION

We have already specified the architecture, hyperparameters and training process. Other details regarding the application are:

· **Data preprocessing:** We have already covered the steps of eliminating missing and duplicate values. As for the outliers, they have been detected, but the choice of whether we are going to keep them or not in the training dataset will be decided later based on performance. Lastly, we will perform feature scaling on the dataset before using it for training. Feature scaling standardizes the range of features, ensuring all features contribute equally to model training, promoting faster convergence, and preventing dominance by certain features. The selected method will be Min-Max scaling, where features are transformed to fit the [0,1] range. It involves subtracting the minimum value of the feature and dividing by the range (the difference between the maximum and minimum values).

· **Training/testing datasets:** As for the question of what part of the dataset will be used for testing and what will be used for training, we will use cross-validation. The dataset is divided into *k* subsets (folds), and the model is trained and evaluated *k* times. In each iteration, one fold is used for testing, while the remaining *k-1* folds are used for training. This process is repeated, with a different fold serving as the test set in each iteration. The final performance metric is the average of the metrics obtained in all *k* iterations. Cross-validation helps assess a model's generalization performance and reduces the impact of data partitioning on evaluation. The value of *k* will be 10, a commonly used value, but it should be easy enough to change it later on and decide the optimal value based on performance.

· **Evaluation measures:** We will evaluate the model using the following measures: **Accuracy** (the ratio of correctly predicted instances to the total number of instances), **Precision** (ratio of true positives to the total predicted positives), **Recall** (ratio of true positives to the total actual positives), **F-Measure** (harmonic mean of precision and recall), **AUC** (metric representing the area under the Receiver Operating Characteristic (ROC) curve. It quantifies the model's ability to distinguish between classes), and **AUPRC** (metric representing the area under the Precision-Recall curve. It focuses on the trade-off between precision and recall).

· **Code implementation:** The language of choice will be Python, favored for machine learning due to its readable syntax, simplicity, and a large, supportive community. Its versatility allows integration into diverse applications, and Python's cross-platform compatibility ensures code portability.