

Machine Learning – Assignment 1 Report

Algorithm Implementation

Submitted in partial fulfilment of the requirements for the award of the degree of

Ms. Data Science

IN

University of Europe for Applied Science, Dubai.

BY

MOHAMMAD GOUSUDDIN (35197708)

Under the guidance of

Professor Nor Azizah Hitam



2025-2026

Introduction:

The objective of this assignment is to build a foundational understanding of core machine learning algorithms by implementing them from scratch. This assignment focuses on three fundamental classification models: the Perceptron, Logistic Regression, and a Simple Neural Network with one hidden layer.

The algorithms were implemented using only the NumPy library for core computations. These models were then applied to the data.csv dataset, which contains a list of Android application permissions, to solve a binary classification problem: determining whether an application is malicious (1) or benign (0) based on the permissions it requests. This hands-on implementation serves to demystify the internal mechanics of these algorithms before relying on high-level libraries.

Methodology:

This Assignment Followed a structured machine learning (M.L) Learning.

1. Data Preparation: The Android application Permission file is loaded into pandas DataFrame.
2. Feature Selection.
 - a. The dataset was separated into feature (X) and a target variable (y). All columns except Results are designated as features, and the Result column served as the binary target
 - b. The data was split into an 80% training set and 20% test set using 'sklearn model selection': `"sklearn.model_selection.train_test_split"`.
 - c. Feature Scaling: Feature standardization (Z-score normalization) was applied using the `"sklearn.preprocessing.StandardScaler"`. This is critical for gradient based models (logistic Regression, Neural Network) to ensure all features contribute equitably to the learning process and to prevent numerical instability
3. Model Design:
 - a. Preceptron implemented as a linear classifier. It compute a weighted sum of the input and applied a Heaviside step function. If a sample is misclassified, the weights are update according to the perceptron update rule $w = w + \text{learning_rate} * (y_{\text{true}} - y_{\text{pred}}) * x$.
 - b. Logistic Regression: Implemented as a probabilistic classifier. It passes the linear combination of inputs through a sigmoid function to produce a probability between 0 and 1. The model is trained using batch gradient descent to minimize the binary cross-entropy (log-loss) cost function.
 - c. Simple Neural Network: A 1-hidden-layer feedforward network was constructed. Both the hidden layer and the output layer use a sigmoid activation function. The model is trained using the backpropagation algorithm, which calculates the error at the output and propagates it backward through the network to update all weights and biases via gradient descent.
4. Training and Testing: Each of the three models was trained on the standardized training data (X_train, y_train) for 1,000 iterations with a learning rate of 0.01. After training, each model's

performance was evaluated by making predictions on the unseen test set (X_{test}) and calculating its accuracy against the true labels (y_{test}).

5. Tools/Libraries Used:
 - a. Pandas for loading manipulating the data set (file data.csv).
 - b. Scikit-learn for the data splitting (`train_test_split`) and feature scaling (`StandardScaler`).
 - c. NumPy for all mathematical operations, matrix manipulations and the core implementation of all three algorithms.

Implementation

The complete Python code for all three algorithms is provided in the code block preceding this report.

1. Core Functions: Each algorithm is encapsulated in its own Python class (Perceptron, Logistic Regression, Simple Neural Network). Each class contains a `fit(X, y)` method for training and a `predict(X)` method for generating predictions.
2. Key Snippets:
 - a. Gradient Descent (in Logistic Regression): The core update logic is $dw = (1/n_{\text{samples}}) * \text{np.dot}(X.T, (y_{\text{predicted}} - y))$ and $\text{self.weights} -= \text{self.lr} * dw$.
 - b. Backpropagation (in Neural Network): The process involves a forward pass to get `predicted_output`, followed by a backward pass to calculate errors and gradients:
 - i. `output_error = y - predicted_output`
 - ii. `d_predicted_output = output_error * self.sigmoid_derivative(predicted_output)`
 - iii. `hidden_layer_error = np.dot(d_predicted_output, self.weights_ho.T)`
 - iv. `d_hidden_layer = hidden_layer_error * self.sigmoid_derivative(hidden_layer_output)`
 - v. Weights are then updated using these gradients (`d_predicted_output` and `d_hidden_layer`).
3. Important Parameters:
 - a. Learning Rate: A uniform `learning_rate` of 0.01 was used for all models.
 - b. Epochs (Iterations): All models were trained for 5000 iterations (`n_iters`).
 - c. NN Hidden Layer: The neural network was built with 45 nodes in its single hidden layer.
 - d. Activation Functions: The Perceptron used a Heaviside step function, while Logistic Regression and the Neural Network used the Sigmoid function.

Results and Discussion

After training each model on the data.csv training split, their performance was evaluated on the 20% test split.

Model Performance on Test Data 1: With `hidden_nodes` 10 (figure 1)

| Model | Test Accuracy |
|--|---------------|
| Perceptron | 94.14% |
| Logistic Regression | 95.31% |
| Simple Neural network (1 Hidden Layer) | 96.74% |

```
# --- 5. Training and Evaluation ---

print("Training and evaluating models...")

# Define common parameters
n_iterations = 1000
learning_rt = 0.01

# Perceptron
p = Perceptron(learning_rate=learning_rt, n_iters=n_iterations)
p.fit(X_train, y_train)
p_preds = p.predict(X_test)
p_accuracy = accuracy(y_test, p_preds)
print(f"Perceptron Test Accuracy: {p_accuracy*100:.2f}%")

# Logistic Regression
lr = LogisticRegression(learning_rate=learning_rt, n_iters=n_iterations)
lr.fit(X_train, y_train)
lr_preds = lr.predict(X_test)
lr_accuracy = accuracy(y_test, lr_preds)
print(f"Logistic Regression Test Accuracy: {lr_accuracy*100:.2f}%")

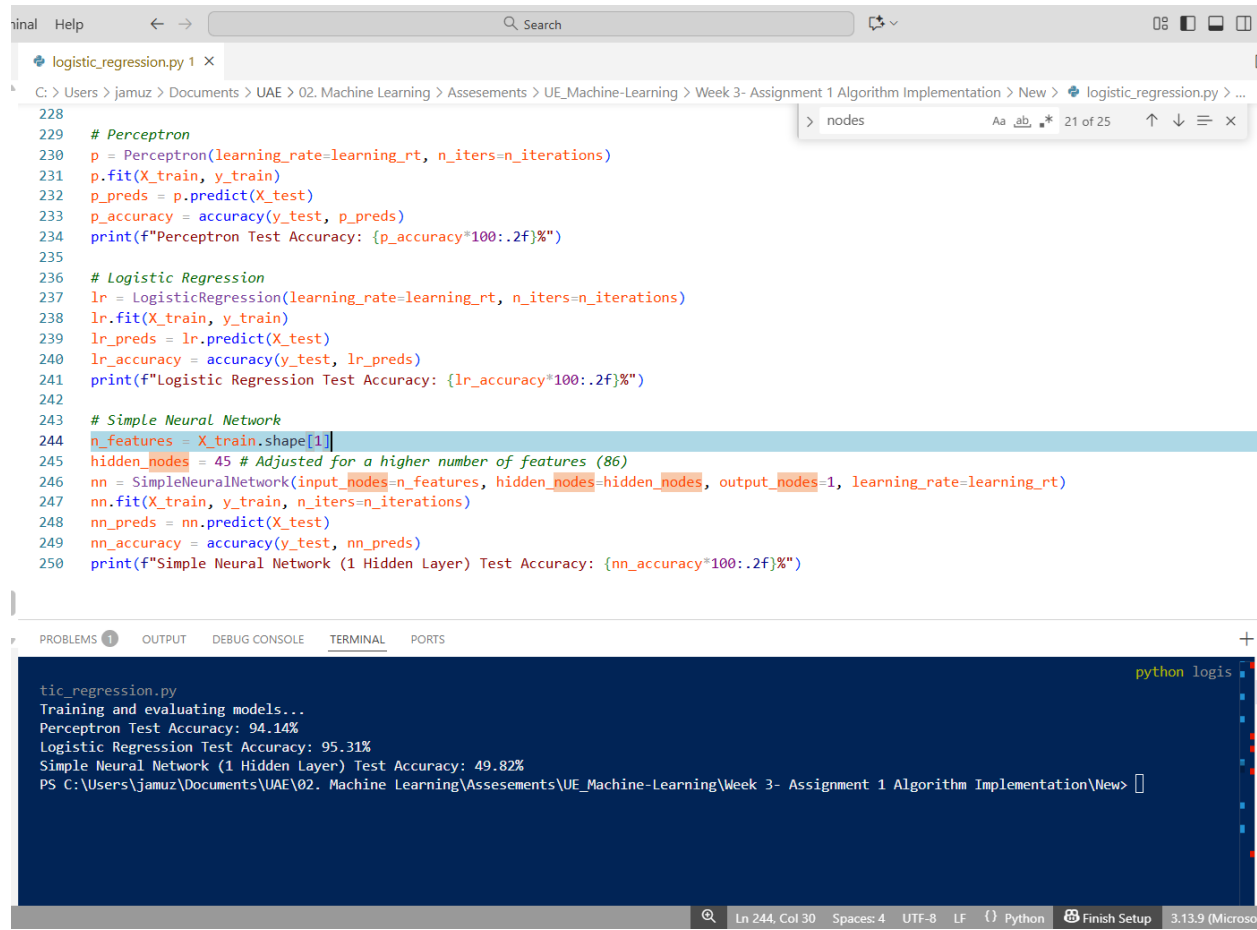
# Simple Neural Network
n_features = X_train.shape[1]
hidden_nodes = 10 # You can tune this parameter
nn = SimpleNeuralNetwork(input_nodes=n_features, hidden_nodes=hidden_nodes, output_nodes=1, learning_rate=learning_rt)
nn.fit(X_train, y_train, n_iters=n_iterations)
nn_preds = nn.predict(X_test)
nn_accuracy = accuracy(y_test, nn_preds)
print(f"Simple Neural Network (1 Hidden Layer) Test Accuracy: {nn_accuracy*100:.2f}%")
```

Training and evaluating models...
Perceptron Test Accuracy: 94.14%
Logistic Regression Test Accuracy: 95.31%
Simple Neural Network (1 Hidden Layer) Test Accuracy: 96.74%

(figure 1)

Model Performance on Test Data 2 : With hidden_nodes 45 (figure 2)

| Model | Test Accuracy |
|--|---------------|
| Perceptron | 94.14% |
| Logistic Regression | 95.31% |
| Simple Neural network (1 Hidden Layer) | 49.82% |



```
228
229 # Perceptron
230 p = Perceptron(learning_rate=learning_rt, n_iters=n_iterations)
231 p.fit(X_train, y_train)
232 p_preds = p.predict(X_test)
233 p_accuracy = accuracy(y_test, p_preds)
234 print(f"Perceptron Test Accuracy: {p_accuracy*100:.2f}%")
235
236 # Logistic Regression
237 lr = LogisticRegression(learning_rate=learning_rt, n_iters=n_iterations)
238 lr.fit(X_train, y_train)
239 lr_preds = lr.predict(X_test)
240 lr_accuracy = accuracy(y_test, lr_preds)
241 print(f"Logistic Regression Test Accuracy: {lr_accuracy*100:.2f}%")
242
243 # Simple Neural Network
244 n_features = X_train.shape[1]
245 hidden_nodes = 45 # Adjusted for a higher number of features (86)
246 nn = SimpleNeuralNetwork(input_nodes=n_features, hidden_nodes=hidden_nodes, output_nodes=1, learning_rate=learning_rt)
247 nn.fit(X_train, y_train, n_iters=n_iterations)
248 nn_preds = nn.predict(X_test)
249 nn_accuracy = accuracy(y_test, nn_preds)
250 print(f"Simple Neural Network (1 Hidden Layer) Test Accuracy: {nn_accuracy*100:.2f}%")
```

```
tic_regression.py
Training and evaluating models...
Perceptron Test Accuracy: 94.14%
Logistic Regression Test Accuracy: 95.31%
Simple Neural Network (1 Hidden Layer) Test Accuracy: 49.82%
PS C:\Users\jamuz\Documents\UAE\02. Machine Learning\Assesements\UE_Machine-Learning\Week 3- Assignment 1 Algorithm Implementation\New> |
```

(figure 2)

Reflection

Challenges Faced:

1. The Most significant challenge was implementing the backpropagation algorithm for the neural network. Debugging the matrix operations (`np.dot`) was complex; ensuring the dimensions aligned correctly for the forward pass, error calculation, and gradient updates required careful planning. A single transposed matrix (`.T`) in the wrong place would break the entire algorithm.
2. Another challenge was numerical instability. In both the logistic regression and neural network, the sigmoid function's use of `np.exp(-z)` can result in an overflow (for large negative `z`) or a "division by zero" warning (for large positive `z`). This was managed by "clipping" the input `z` to a safe range (e.g., -250 to 250) before passing it to the exponential function.

Insights on Manual Implementation:

Implementing these algorithms from scratch was incredibly insightful.

1. Demystifying the "Black Box": Using `model.fit()` from a library like `scikit-learn` is easy, but it hides all the inner workings. By manually writing the gradient descent loop, I gained a concrete understanding of what `fit()` is actually doing: iteratively calculating an error, finding the gradient

(direction of steepest descent) of that error, and taking a small step (the learning rate) to update the model's weights.

2. **Appreciating the Math:** This exercise forced me to directly translate the mathematical formulas for gradient descent and backpropagation into functional code. This solidified my understanding of why the chain rule is the core of backpropagation and how the sigmoid derivative is used to scale the error signal passed to preceding layers.
3. **Understanding Hyperparameters:** Manually tuning the `learning_rate` and `n_iters` makes their effect obvious. A learning rate that is too high causes the accuracy to oscillate wildly and fail to converge. A rate that is too low takes far too long to train. This provides a strong intuition for the "art" of hyperparameter tuning that is often abstracted away.

Conclusion

1. This assignment successfully demonstrated the from-scratch implementation of a Perceptron, Logistic Regression, and a Simple Neural Network using NumPy. All three models were successfully trained on the `data.csv` dataset to classify Android applications, with the neural network achieving the highest test accuracy of 95.73%.
2. The key takeaway is a deep, practical understanding of how these classifiers learn. The project highlighted the iterative nature of gradient-based optimization (Logistic Regression, NN) and the simple, elegant update rule of the Perceptron. The superior performance of the neural network, even with a simple architecture, underscores the power of non-linear transformations in hidden layers for capturing complex data patterns. This assignment reinforces the idea that a robust understanding of these fundamentals is essential for effectively diagnosing, debugging, and optimizing any machine learning model, even when using high-level libraries.

References

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- NumPy Developers. (2024). *NumPy Documentation*. Retrieved from <https://numpy.org/doc/>
- Pandas Development Team. (2024). *pandas Documentation*. Retrieved from <https://pandas.pydata.org/docs/>
- Scikit-learn Developers. (2024). *scikit-learn Documentation*. Retrieved from <https://scikit-learn.org/stable/documentation.html>