
Winter of Code (WOC) Final Report

Project: Machine Learning Library from Scratch & Kaggle Competition

Name : Mohammad Sahil Bhat

Program: Winter of Code (WOC) 2025–26

1. Executive Summary

During the Winter of Code, I developed ml_lib, a modular machine learning library built entirely from scratch using **Python** and **NumPy**. The project involved implementing the mathematical foundations of preprocessing, linear models, tree-based ensembles, and deep neural networks. I subsequently used a tree based model to a **Kaggle competition**, achieving competitive results by iterating through baseline linear models to complex neural architectures.

2. Library Architecture & Structure

The library is designed with a professional, modular hierarchy to ensure code reusability and clean API calls (similar to Scikit-Learn).

```
ml_lib/
    └── core/                                # Base classes and abstract logic
        ├── base_layer.py                    # Template for Neural Network layers
        ├── base_model.py                  # Template for all ML models (fit/predict)
        ├── loss.py                         # Abstract loss function logic
        └── optimizer.py                  # Base class for optimization algorithms
```

```
└── preprocessing/          # Data transformation and cleaning
    ├── scaler.py           # StandardScaler (Mean/Std normalization)
    ├── imputer.py          # SimpleImputer (Mean/Median/Constant)
    ├── encoding.py          # OneHot & Label Encoding
    ├── polynomial_features.py # Feature interaction generation
    └── pipeline.py          # Orchestrator for chaining transformations

|
└── neural_network/        # Deep Learning Framework
    ├── layers.py            # Dense/Fully Connected layers
    ├── activations.py       # ReLU, Sigmoid, Softmax, Tanh
    ├── sequential.py         # Layer stacking and training engine
    ├── initializations.py   # Xavier (Glorot) & He initialization
    ├── losses.py             # Cross-Entropy, MSE, MAE
    └── optimizers/           # Advanced gradient descent variants
        ├── adam.py            # Adaptive Moment Estimation
        ├── rmsprop.py          # Root Mean Square Propagation
        └── sgd.py              # Stochastic Gradient Descent

|
└── tree/                  # Tree-based algorithms
    ├── decision_tree.py     # Entropy/Gini based splitting logic
    └── random_forest.py     # Bagging and Ensemble logic

|
└── linear_models/          # Statistical learning models
    ├── linear_regression.py # OLS & Gradient Descent regression
```

```
|   └── logistic_regression.py      # Binary classification logic  
|  
|  
|   └── clustering/              # Unsupervised learning  
|       └── kmeans.py            # Centroid-based clustering  
|  
|   └── elbow.py                # K-selection optimization tool  
|  
|  
|   └── neighbors/              # Instance-based learning  
|       └── knn.py               # K-Nearest Neighbors implementation  
|  
|  
|   └── metrics/                # Evaluation suite  
|       └── math.py              # R2, MSE, MAE calculations  
|  
|       └── validation.py        # Confusion Matrix, Accuracy, F1-Score  
|  
|  
└── utils/                     # Helper utilities  
    └── data.py                # Data loading and shuffling  
    └── timers.py               # Performance benchmarking  
    └── visuals/                # Simple Plotting and Graphing  
        └── classification_plots.py  
        └── regression_plots.py  
        └── decision_boundary.py
```

Directory Breakdown

- **core/**: The foundation of the library. Contains abstract base classes (`base_model.py`, `base_layer.py`) and base logic for optimizers and loss functions.
- **preprocessing/**: Tools for data cleaning and transformation, including `scaler.py`,

imputer.py, encoding.py, and the pipeline.py orchestrator.

- **linear_models/**: Implementations of Linear and Logistic Regression using Gradient Descent.
 - **tree/**: Logic for decision_tree.py (recursive splitting) and random_forest.py (bagging and ensemble voting).
 - **neural_network/**: A complete deep learning suite.
 - layers.py: Dense/Fully Connected layers.
 - activations.py: ReLU, Sigmoid, Softmax.
 - optimizers/: Advanced algorithms like Adam, RMSProp, and Momentum.
 - sequential.py: The wrapper to stack layers and train models.
 - **metrics/**: Functions for validation.py (Accuracy, F1-score) and math.py (MSE, R2).
 - **utils/visuals/**: Plotting scripts for loss curves and decision boundaries.
-

3. Code Flow: From Raw Data to Prediction

The library follows a systematic flow to ensure data integrity (preventing data leakage) and efficient training:

1. **Initialization**: Data is loaded and split using metrics/validation.py.
 2. **Preprocessing Pipeline**: Raw features pass through a Pipeline.
 3. **Model Definition**: A model is instantiated (e.g., RandomForestClassifier or Sequential).
 4. **Training (The "Fit" Phase)**:
 - For **Trees**: The library performs recursive binary splitting based on Information Gain/Gini Impurity.
 - For **Neural Networks**: The Sequential model executes a **Forward Pass** (calculating activations) followed by a **Backward Pass** (computing gradients via the Chain Rule).
 5. **Optimization**: The optimizer updates weights (e.g., using Adam logic to adapt learning rates).
 6. **Evaluation**: Predictions are run through metrics to generate Confusion Matrices and Loss Curves.
-

4. Key Implementations

1. Base Architecture and Model Interface

The library is built upon a consistent API where all learning algorithms inherit from a base structure to ensure interoperability.

- **BaseModel**: Defines the required fit(X, y) and predict(X) interface for all supervised models, ensuring they can be used interchangeably in evaluation scripts.
- **Pipeline System**: Orchestrates data flow by chaining multiple preprocessing steps. It supports fit_transform for training data and transform for test data to prevent information

leakage.

2. Data Preprocessing Suite

These modules handle the critical task of preparing raw data for model consumption.

- **StandardScaler:** Implements Z-score normalization to ensure features have a mean of 0 and a standard deviation of 1. It includes a small epsilon (10^{-8}) in the denominator to prevent division by zero for zero-variance features.
- **MinMaxScaler:** Scaler that shifts and rescales data into a range between 0 and 1, also utilizing an epsilon constant for numerical stability during division.

3. Linear Models: Logistic Regression

Implemented for binary classification using a statistical approach:

7. **Mathematical Foundation:** Utilizes the **Sigmoid function**, $\frac{1}{1 + e^{-z}}$, to map linear combinations of inputs to probabilities.
8. **Gradient Descent:** Updates parameters (w and b) iteratively by calculating the gradient of the loss w.r.t. the weights:

$$dW = \frac{1}{n} X^T (\hat{y} - y)$$

4. Tree-Based Algorithms

These models focus on non-linear decision-making through recursive feature splitting.

1. **DecisionTreeClassifier:**
 - **Entropy Calculation:** Measures data impurity using $-\sum p \cdot \log_2(p)$.
 - **Information Gain:** Determines the best split by calculating the reduction in entropy after a potential split.
 - **Recursive Growth:** Uses a `_grow_tree` function that continues until reaching a `max_depth` or a minimum sample threshold.
2. **RandomForestClassifier:**
 - **Bagging (Bootstrap Aggregating):** Fits multiple independent trees on random subsets of the data using `_bootstrap_sample`.
 - **Majority Voting:** Aggregates predictions from all trees in the forest and returns the most frequent label via `np.bincount().argmax()`.

5. Deep Learning (Neural Networks)

A modular framework for building multi-layer perceptrons.

- **Dense Layer:**
 - **Initialization:** Supports custom initialization schemes like **Xavier** to maintain

- activation variance across layers.
- **Forward Pass:** Computes the linear transformation $Z = XW + b$ followed by an activation function.
 - **Backward Pass:** Implements the Chain Rule to calculate gradients w.r.t. weights ($X^T \cdot dOut$) and bias ($\sum (dOut)$), then passes the gradient to the preceding layer.
 - **Sequential Model:** Manages the training loop. It iterates through layers during the forward pass to generate predictions and iterates in reversed order during the backward pass to propagate errors.
 - **Adam Optimizer:** An advanced adaptive learning rate algorithm that tracks the first moment (momentum) and second moment (uncentered variance) of gradients to stabilize and accelerate training.

Summary of Implementation Excellence

- **Vectorization:** Every key component—from the Adam optimizer updates to Decision Tree entropy calculations—uses NumPy array operations instead of standard Python loops for high performance.
- **Numerical Stability:** Frequent use of epsilon constants (10^{-8}) across scalers, optimizers, and model predictions prevents runtime crashes during division or logarithmic calculations.

5. Kaggle Competition: Strategy & Observations

The Approaches Tried

2. **Submission 1 (Stacked Ensemble Model):** This submission was built by a stacked ensemble model combining LightGBM, XGBoost, and HistGradientBoosting, trained with 5-fold cross-validation. After preprocessing (feature selection, PCA, scaling), a Logistic Regression meta-model learned to combine the base models. A final threshold optimization step maximized Balanced Accuracy for the best possible performance.
3. **Submission 2 (LightGBM Model):** Only the LightGBM model was used.

Training Details

- **Optimizers:** Compared SGD vs Adam. Adam converged 3x faster due to its adaptive learning rate.

6. Technical Analysis: Lessons in Engineering & Optimization

This section details the critical technical hurdles encountered during the development of ml_lib and the subsequent Kaggle competition. The focus was on transitioning from purely mathematical formulas to **computationally stable** code.

6.1 High-Impact Successes (What Worked)

I. Xavier (Glorot) Initialization for Gradient Flow

Implemented in ml_lib/neural_network/initializations.py, Xavier initialization proved vital for deep architectures. By setting the initial weights with a variance of $2 / (n_{in} + n_{out})$, I was able to keep the variance of activations consistent across all layers.

- **Result:** This prevented the "Vanishing Gradient" problem that initially stalled my 4-layer networks when using standard Gaussian noise. It allowed the model to start learning from the very first epoch.

II. Adaptive Learning Rates via Adam Optimizer

While standard Stochastic Gradient Descent (SGD) worked for linear models, the Kaggle dataset required the sophisticated **Adam Optimizer** (found in ml_lib/neural_network/optimizers/adam.py).

- **The Logic:** By maintaining moving averages of both the gradients (momentum) and the squared gradients (scaling), Adam effectively provided a custom learning rate for every single parameter in the network.
- **Result:** This led to a **3x faster convergence rate** and helped the model navigate the "ravines" of the loss surface much more effectively than fixed-rate SGD.

III. Numerical Stability in Softmax and Cross-Entropy

A recurring challenge in scratch implementation is the "Exponential Blow-up." I implemented a **Numerically Stable Softmax** by subtracting the maximum value from the input vector before

$$\text{softmax}(x)_i = \frac{e^{x_i - \max(x)}}{\sum e^{x_j - \max(x)}}$$

exponentiation as suggested by Andrew Ng :

9. **Result:** This eliminated the NaN (Not a Number) errors that occur when the computer tries to calculate e^{700} or higher, ensuring the model could handle large feature values safely.
-

6.2 Critical Roadblocks & Refinements (What Didn't Work & Why)

I. Sigmoid Saturation and Vanishing Gradients

Initially, I used Sigmoid activations for all hidden layers.

2. **The Issue:** As the network grew deeper, the gradients became extremely small (approaching zero) because the derivative of the Sigmoid function peaks at only 0.25. This effectively "killed" the learning process in the early layers.
3. **The Fix:** I refactored the architecture to use **ReLU (Rectified Linear Unit)** in hidden layers, which has a constant derivative of 1 for all positive inputs, ensuring that the gradient signal remained strong throughout the backpropagation chain.

II. Floating Point Precision in Scaling

During the implementation of StandardScaler, I initially encountered "Division by Zero" errors when a feature had zero variance (all values were the same).

The Fix: I implemented a small safety constant ($\text{epsilon} = 10^{-8}$) in the denominator: $(X - \mu) / (\sigma + \epsilon)$. This small engineering detail was crucial for the library's robustness against real-world, "messy" datasets.

7. Visualizations & Observations

- **Training vs. Validation Loss:** Plots generated via utils/visuals showed a clear convergence.
 - **Confusion Matrix:** Revealed that the model was occasionally confusing Class A with Class B, leading to the addition of more hidden units in the Neural Network to capture finer boundaries.
-

8. Conclusion

The Winter of Code journey provided a rigorous platform to bridge the gap between ML theory and software engineering. By building `ml_lib`, I have developed a deep intuition for optimization, matrix calculus, and the importance of a clean data pipeline. These implementation skills were directly validated by the library's performance in the Kaggle competition.
