1. Introduction

This report documents a full machine learning library implemented from scratch using Python and NumPy. The library includes supervised, unsupervised, and neural network models: Linear Regression, Polynomial Regression, Logistic Regression, K-Nearest Neighbors, K-Means Clustering, Decision Trees, and a Neural Network with Adam optimization. All models are built without using high-level ML libraries, and all algorithms are explicitly implemented from first principles, with careful attention to preprocessing, numerical stability, and efficient computation.

Whenever we use the model from library, we just fit the model with our dataset using.

```
{model_code}_fit(X, y, other parameters)
```

Where X is the training features and y is training prediction

Predictions can be obtained as:

```
y_pred = {model_code}_predict(X, other parameter)
```

2. Data Preprocessing

Preprocessing is crucial for all models. Each dataset passes through:

```
X, stats, fill = preprocess(X)
```

This handles:

1. Missing values (NaNs)
2. Standardization (scaling to zero mean, unit variance)

2.1 Handling Missing Values

For each feature j:

$$X_{ij} \leftarrow \mu_j = \frac{1}{n}\sum_i X_{ij}$$

This is implemented as:

```
fill = np.nanmean(X, axis=0)
X[idx] = np.take(fill, np.where(idx)[1])
```

NaN replacement ensures that operations such as dot products, distance calculations, and gradient descent are numerically stable.

2.2 Standardization

Each feature is scaled:

$$z_j = \frac{x_j - \mu_j}{\sigma_j}$$

It is implemented in the program as:

```
    X = (X - mean) / std
```

Prevents gradient descent from being skewed by features of different scales.

3. Evaluation Methodology and Data Partitioning

To objectively evaluate model performance, the implementation employs a deterministic train–test split along with task-specific performance metrics. This ensures that the learning algorithms are assessed not by memorization but by their ability to generalize to unseen data.

3.1 Train–Test Split

The function

```
        train_test_split(X, y, test_size=0.2)
```

partitions the dataset into two disjoint subsets: a training set and a testing set.
Let the dataset consist of $n$samples:

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$$

Let the test fraction be denoted by α(default $\alpha = 0.2$). Then,

$$n_{test} = \lfloor \alpha n \rfloor, n_{train} = n - n_{test}$$

The first n_{train}samples are assigned to the training set:

$$X_{train}, y_{train} = \{(x_i, y_i)\}_{i=1}^{n_{train}}$$

and the remaining samples to the test set:

$$X_{test}, y_{test} = \{(x_i, y_i)\}_{i=n_{train}+1}^{n}$$

This separation prevents information leakage and enables unbiased evaluation of predictive performance.

3.2 Accuracy Score

For classification tasks, the framework employs accuracy as the evaluation metric.
Given predicted labels

$$\hat{y}_1, \hat{y}_2, \dots, \hat{y}_n$$

and true labels

$$y_1, y_2, \dots, y_n,$$

the accuracy is defined as

$$\text{Accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(\hat{y}_i = y_i)$$

where $\mathbf{1}(\cdot)$ is the indicator function.
In the code, this is computed by subtracting predictions from true labels and counting how many differences are zero:

$$\hat{y}_i - y_i = 0 \iff \hat{y}_i = y_i$$

Finite-value masking is applied to avoid corruption from NaNs or infinities.
Thus, accuracy measures the fraction of samples that are classified correctly.

3.3 R² Score (Coefficient of Determination)

For regression models, the framework evaluates performance using the coefficient of determination, denoted R^2.
Two quantities are defined:
Residual Sum of Squares

$$SS_{res} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

This measures the squared error of the model.
Total Sum of Squares

$$SS_{tot} = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

where

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

is the mean of the true targets.
The R^2score is then computed as

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

4. Linear Regression

4.1 Model

Linear regression fits a hyperplane in feature space:

$$\hat{y} = Xw + b$$

where $X \in \mathbb{R}^{n \times d}$, $w \in \mathbb{R}^d$, $b \in \mathbb{R}$.

4.2 Loss Function

Mean Squared Error (MSE):

$$L(w, b) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (x_i^T w + b))^2$$

4.3 Gradient Descent

The derivatives are:

$$\frac{\partial L}{\partial w} = \frac{2}{n} X^T (Xw + b - y)$$
$$\frac{\partial L}{\partial b} = \frac{2}{n} \sum (Xw + b - y)$$

The weights are simply updated as:

$$w_i \leftarrow w_i - \frac{\partial L}{\partial w_i} \cdot \alpha$$

$$b \leftarrow b - \frac{\partial L}{\partial b} \cdot \alpha$$

Where α is the learning rate, chosen optimally.

Implemented in code:

```
    err = yp - y
    w -= lr * (X.T @ err) / len(y)
    b -= lr * err.mean()
```

4.4 Prediction

$$\hat{y} = Xw + b$$

5. Polynomial Regression

5.1 Feature Expansion

Transforms $x$to polynomial features:

$$\phi(x) = [1, x, x^2, \ldots, x^d]$$

Code:

```
np.hstack([x**i for i in range(d+1)])
```

This converts a nonlinear function into a linear one in the expanded space.

5.2 Least squares

This code does not use gradient descent. Instead:

```
w = pinv(X.T @ X) @ X.T @ y
```

This solves

$$\min_{w} \parallel Xw - y \parallel^2$$

This is the normal equation.

$$X^T X w = X^T y$$

Because polynomial features are highly correlated, $X^T X$may not be invertible. The pseudoinverse finds the least-norm solution that minimizes error.

5.3 Prediction

$$\hat{y} = \phi(x)^T w$$

6. Logistic Regression

6.1 Model

$$z = X\theta, \qquad h = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid maps linear output to probability.
Also, the array X contains a bias for the intercept.

6.2 Loss Function

Binary cross-entropy:

$$L(\theta) = -\frac{1}{n}\sum_i [y_i \log\, h_i + (1 - y_i) \log\,(1 - h_i)]$$

6.3 Gradient

The partial derivative can be easily calculated as:

$$\frac{\partial L}{\partial \theta} = \frac{1}{n} X^T (h - y)$$

Hence the gradient descent can be implemented in code as:

```
    grad = Xb.T @ (h - y) / m
    theta -= lr * grad
```

6.4 Prediction

The sigmoid is calculated from the test data, giving us the probability of the classification, then keeping 0.5 as threshold,

$$\hat{y} = \begin{cases} 1, & h \geq 0.5 \\ 0, & h < 0.5 \end{cases}$$

7. K-Nearest Neighbors (KNN)

7.1 Distance Metric

Squared Euclidean distance:

$$d_i = \sum_j (x_j - x_{ij})^2$$

Implemented in vectorized way as:

```
    dists = np.sum((X - x[i])**2, axis=1)
```

Note we are using squared Euclidean distance as we just need to compare the distances, and square rooting will just add an extra useless step.

7.2 Weighted Voting

$$w_i = \frac{1}{d_i + \epsilon}, \hat{y} = \arg\max_k \sum_{i \in k} w_i$$

Closer neighbors have more influence.

8. K-Means Clustering

K-Means is an unsupervised clustering algorithm. Its goal is to partition a dataset into $k$clusters by minimizing the within-cluster sum of squares (WCSS):

$$J(C,z)=\sum_{i=1}^{n}\parallel x_i-C_{z_i}\parallel^2$$

Where:

- $x_i$is a data point
- $C_j$is the centroid of cluster j
- $z_i\in\{1,2,\ldots,k\}$is the cluster assignment for x_i

8.1 Initialization

Choosing good initial centroids is critical. The one used in the code is:

```
def k_init(X, k):
    C = [X[np.random.randint(len(X))]]
    for _ in range(1, k):
        d = np.min([np.sum((X-c)**2,axis=1) for c in C], axis=0)
        p = d / np.sum(d)
        C.append(X[np.random.choice(len(X), p=p)])
    return np.array(C)
```

Explanation:

1. First centroid is randomly chosen from the data.
2. Subsequent centroids are sampled probabilistically based on distance squared from the nearest existing centroid:

$$P(x)=\frac{D(x)^2}{\sum D(x)^2}$$

- $D(x)$is the distance from the nearest existing centroid
- This ensures spread-out initial centroids, reducing chances of poor local minima.

8.2 Iterative Update (Lloyd's Algorithm)

After initialization, K-Means alternates between assignment and update:

Step 1: Assignment

```
D = np.array([np.sum((X-c)**2,axis=1) for c in C])
y = np.argmin(D, axis=0)
```

- Compute squared Euclidean distance from each data point to all centroids:

$$d_{ij}=\parallel x_i-C_j\parallel^2$$

- Assign each point to the nearest centroid:

$$z_i = \min_j d_{ij}$$

Step 2: Centroid Update

```
Cnew.append(X[y==i].mean(axis=0))
```

- For each cluster j, update its centroid to the mean of assigned points:

$$C_j = \frac{1}{\mid S_j \mid} \sum_{x_i \in S_j} x_i$$

- This minimizes the WCSS for the given assignments. If a cluster has no points, the old centroid is retained.

8.3 Convergence Check

```
if np.allclose(C, Cnew):
    break
```

- Algorithm stops when centroids do not change significantly, meaning the solution is stable.
- K-Means minimizes the convex objective iteratively, but final convergence depends on initialization.

8.4 Prediction

Once the model is trained, predicting the cluster for new points:

```
D = np.array([np.sum((X-c)**2,axis=1) for c in
kmeans_model["C"]])
return np.argmin(D, axis=0)
```

- Compute distance to all centroids
- Assign to nearest centroid

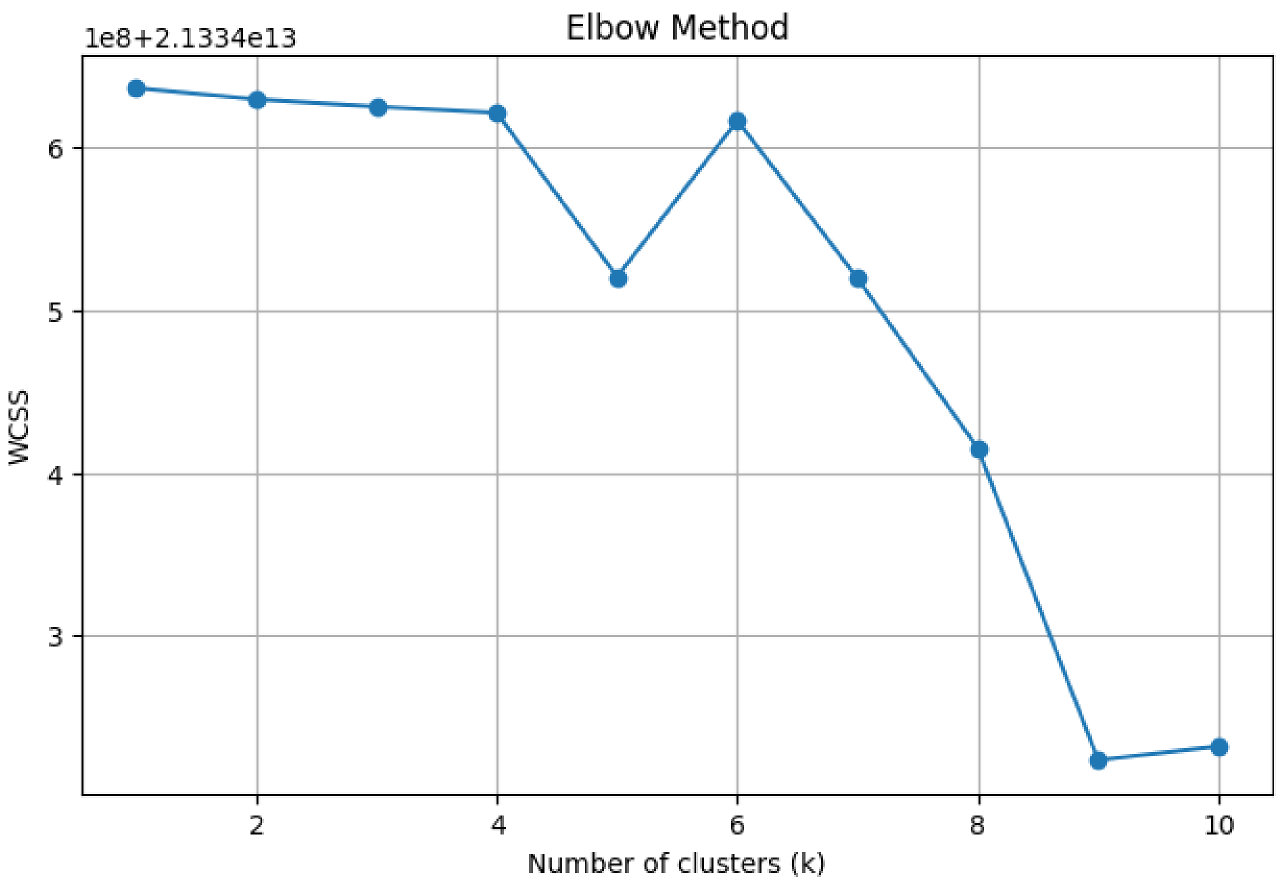
8.5 Determining k (Using Elbow Method)

Since there are many numbers of clusters possible in unsupervised data, we iterate over a good number of k's and use elbow's method to determine the best.

In the elbow method we plot the Within-Cluster Sum of Squares (WCSS) for different k:

$$\text{WCSS}(k) = \sum_{i=1}^{k} \sum_{x \in C_i} \parallel x - \mu_i \parallel^2$$

As k increases, WCSS must decrease (more clusters means smaller within-cluster error).
The elbow is the point where adding more clusters stops giving a big improvement, i.e., where the curve bends sharply.
That bend corresponds to the true number of natural groups in the data.
For example, if the graph looks like this:



The point which looks like an elbow (here k=9), will be the optimal number of cluster.

9. Decision Tree

A decision tree learns a set of rules of the form:

$$\text{If } x_{f_1} \leq t_1 \text{ and } x_{f_2} > t_2 \cdots \Rightarrow y = c$$

Each node performs a threshold test on one feature, splitting the data into two subsets. This continues recursively until data becomes pure or a maximum depth is reached.

9.1 Entropy (Measure of Impurity)

The code computes:

```
    vals, counts = np.unique(y, return_counts=True)
    p = counts / counts.sum()
    entropy = -np.sum(p * np.log2(p + 1e-8))
```

Mathematically:

$$H(Y) = -\sum_{k=1}^{K} p_k \log_2 p_k$$

- p_k is the probability of class k in the current node
- Measures uncertainty: maximum for uniform class distribution, zero if all same class

9.2 Information Gain (Choosing the Best Split)

For a feature f and threshold t:

$$IG(f,t) = H(Y) - \left(\frac{n_L}{n} H(Y_L) + \frac{n_R}{n} H(Y_R)\right)$$

- Y_L, Y_R are labels on left/right of the split
- n_L, n_R are their counts
- IG measures reduction in uncertainty

9.3 Searching for the Best Split

The best_split() function tries many feature-threshold pairs and selects the one with maximum information gain.

Code:

```
    def best_split(X, y, n_thresholds=20):
        best_gain = -1
        best_f, best_t = None, None
        parent_entropy = entropy(y)
        for f in range(X.shape[1]):
            col = X[:, f]
            thresholds = np.linspace(np.min(col), np.max(col),
    n_thresholds)
            for t in thresholds:
                left_mask = col <= t
                right_mask = col > t
                if left_mask.sum() == 0 or right_mask.sum() == 0:
                    continue
                e = (left_mask.sum()/len(y))*entropy(y[left_mask]) + \
                     (right_mask.sum()/len(y))*entropy(y[right_mask])
                gain = parent_entropy - e
                if gain > best_gain:
                    best_gain = gain
                    best_f, best_t = f, t
        return best_f, best_t
```

For every feature f, and for 20 thresholds t:

1. Split the data into:

$$L = \{x: x_f \leq t\}, R = \{x: x_f > t\}$$

2. Compute weighted entropy
3. Compute information gain
4. Keep the best

This approximates the optimal split efficiently.

9.4 Recursion: Building the Tree

```
    return {"f": f, "t": t,
            "left": tree_generation(X[left], y[left], depth+1),
            "right": tree_generation(X[right], y[right], depth+1)}
```

- Base cases:
  - All labels identical : return leaf with class
  - Max depth reached : return leaf with **majority class**
- Tree grows recursively : partitions space into **decision regions**

9.4 Prediction

```
    if x[tree["f"]] <= tree["t"]:
        return tree_predict_one(x, tree["left"])
    else:
        return tree_predict_one(x, tree["right"])
```

- Traverses tree based on feature thresholds
- Returns leaf class when reached

10. Neural Network

Implementation is a fully connected feedforward neural network that supports:

- Classification using sigmoid activations (binary or multiclass)
- Regression using linear output

The network consists of:

- Input layer
- One or more hidden layers
- Output layer

Forward and backward passes are implemented explicitly in code without using any external library.

10.1 Forward Pass

The forward pass propagates input through the network:

$$a^{(0)} = x$$
$$z^{(l)} = a^{(l-1)}W^{(l)} + b^{(l)}$$
$$a^{(l)} = \begin{cases} \sigma(z^{(l)}), & \text{if hidden layer or classification output} \\ z^{(l)}, & \text{if regression output} \end{cases}$$

Where:

- $W^{(l)} \in \mathbb{R}^{n_{l-1} \times n_l}$ are the weights of layer l
- $b^{(l)} \in \mathbb{R}^{1 \times n_l}$ are the biases of layer l
- $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function

Code implementation:

```
def fwd(x, w, b, task):
    acts = [x]
    a = x
    for i in range(len(w)):
        z = a @ w[i] + b[i]
        if i == len(w)-1 and task=="reg":
            a = z
        else:
            a = sigmoid(z)
        acts.append(a)
    return acts
```

- For multiclass classification, the output layer is sigmoid-activated per class neuron.
- This allows non-contiguous or negative integer classes using one-hot encoding internally.

10.2 Loss Function

The network supports multiple tasks:

- Regression: Mean Squared Error (MSE)

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Binary classification: Binary Cross-Entropy

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

- Multiclass classification (sigmoid-based): Sum of binary cross-entropies across all classes

$$L = -\frac{1}{n}\sum_{i=1}^{n}\sum_{c=1}^{C}[y_{i,c} \log(\hat{y}_{i,c}) + (1 - y_{i,c})\log(1 - \hat{y}_{i,c})]$$

Code:

```
def loss(y, yp, task):
    eps = 1e-8
    yp = np.clip(yp, eps, 1-eps)
    if task == "clf":
        return -np.mean(y*np.log(yp) + (1-y)*np.log(1-yp))
    elif task == "clf_multi":
        return -np.mean(np.sum(y*np.log(yp) + (1-y)*np.log(1-yp),
axis=1))
    else:  # regression
        return np.mean((y - yp)**2)
```

- `np.clip` ensures numerical stability in logarithms.
- For multiclass, each output neuron is treated as a separate sigmoid.

10.3 Weight Initialization (Xavier)

Weights are initialized using Xavier initialization to preserve variance of activations across layers:

```
lim = np.sqrt(1/ls[i])
wi = np.random.randn(ls[i], ls[i+1]) * lim
```

- For a neuron with n inputs:

$$z = \sum_{i=1}^{n} W_i \, x_i$$

- If input variance is 1, we want $\mathrm{Var}(z) \approx 1$
- Each $W_i$should have variance $1/n$, standard deviation $\sqrt{1/n}$
- Prevents vanishing/exploding gradients in deep networks

10.4 Backpropagation

Gradients are computed for weights and biases:

- Output layer:

$$\delta^{(L)} = \begin{cases} \hat{y} - y, & \text{regression} \\ \text{sigmoid derivative} \times \dfrac{\partial L}{\partial a^{(L)}}, & \text{classification} \end{cases}$$

- Hidden layers:

$$\delta^{(l)} = (\delta^{(l+1)} W^{(l+1)T}) \cdot \sigma'(a^{(l)})$$

- Gradients:

$$\frac{\partial L}{\partial W^{(l)}} = a^{(l-1)T} \delta^{(l)}, \frac{\partial L}{\partial b^{(l)}} = \sum_i \delta_i^{(l)}$$

```
Code:
dz = da if (i==len(w)-1 and task=="reg") else da*sigmoid_derivative(ac)
dw = ap.T @ dz
db = np.sum(dz, axis=0, keepdims=True)
da = dz @ w[i].T
```

- ap = activations from previous layer
- ac = current activations
- da = backpropagated gradient

10.5 Adam Optimizer

Adam updates weights adaptively using first and second moments:

Step 1: Moving averages

```
mw[i] = beta1*mw[i] + (1-beta1)*dw  # first moment (momentum)
vw[i] = beta2*vw[i] + (1-beta2)*(dw**2)  # second moment (RMS
scaling)
```

Step 2: Bias correction

```
    mwc = mw[i] / (1 - beta1**t)
    vwc = vw[i] / (1 - beta2**t)
```

Step 3: Parameter update

```
    w[i] -= lr * mwc / (np.sqrt(vwc) + eps)
    b[i] -= lr * mbc / (np.sqrt(vbc) + eps)
```

- Each weight is adaptively scaled according to past gradients
- $\epsilon = 10^{-8}$prevents division by zero

10.6 Training Loop

```
    for i in range(ep):
        t += 1
        acts = fwd(x, w, b, task)
        l = loss(y, acts[-1], task)
        back(y, w, b, acts, lr, mw, vw, mb, vb, t, task)
        if i % 500 == 0:
            print("ep", i, "loss", l)
```

- fwd() computes outputs
- loss() computes gradient signal
- back() performs backpropagation + Adam updates
- Every 500 epochs, loss is printed to monitor convergence

10.7 Prediction

```
    out = fwd(x, nn_model["w"], nn_model["b"], nn_model["task"])[-1]
    if nn_model["task"] == "clf":
        return (out > 0.5).astype(int)
    elif nn_model["task"] == "clf_multi":
        idx = np.argmax(out, axis=1)
        return np.array([nn_model["classes"][i] for i in idx])
    return out
```

- Binary classification: threshold sigmoid at 0.5
- Multiclass: returns original class labels, even if negative or non-contiguous
- Regression: returns raw continuous output