# CSCI 580 Assignment5: Perceptron

Kshiraj Kunta

## Part1:

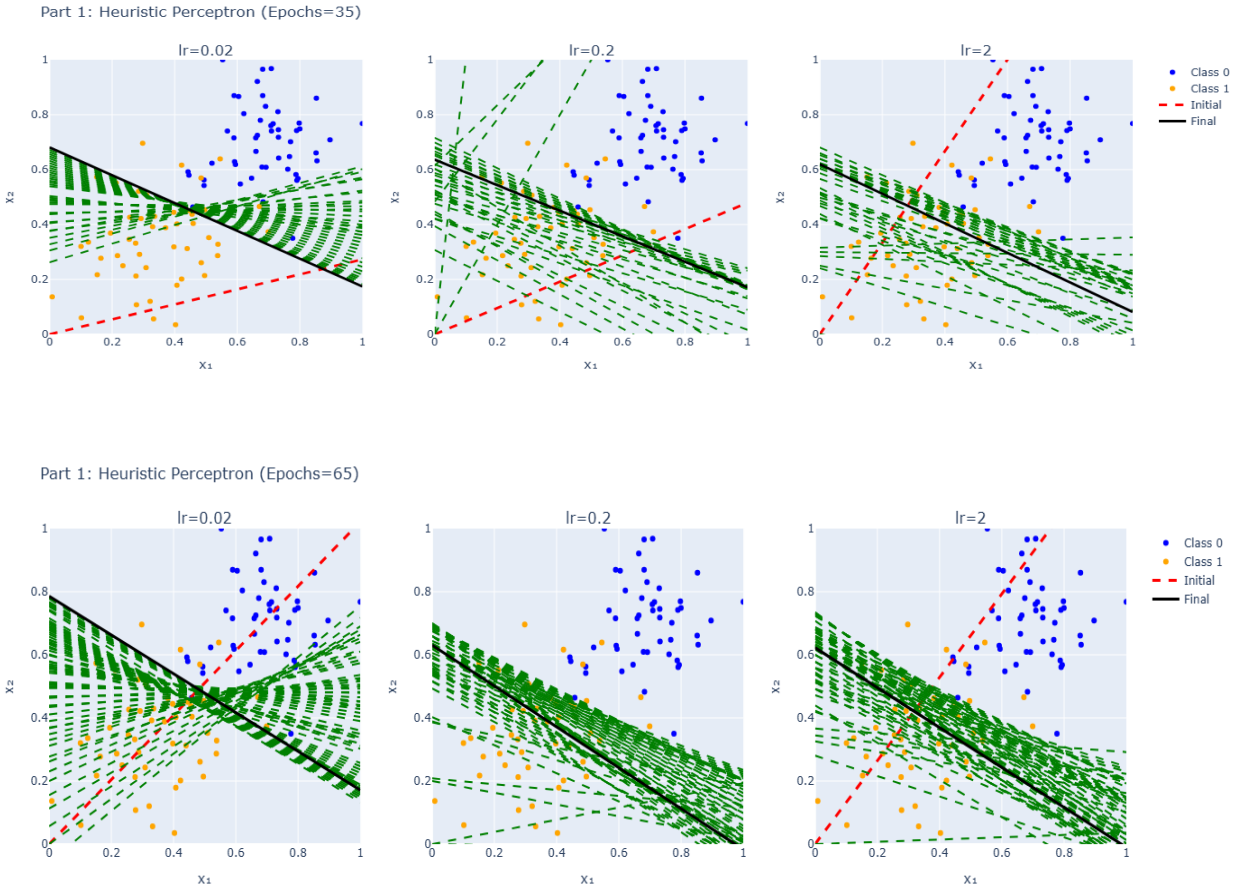
Code Snippet

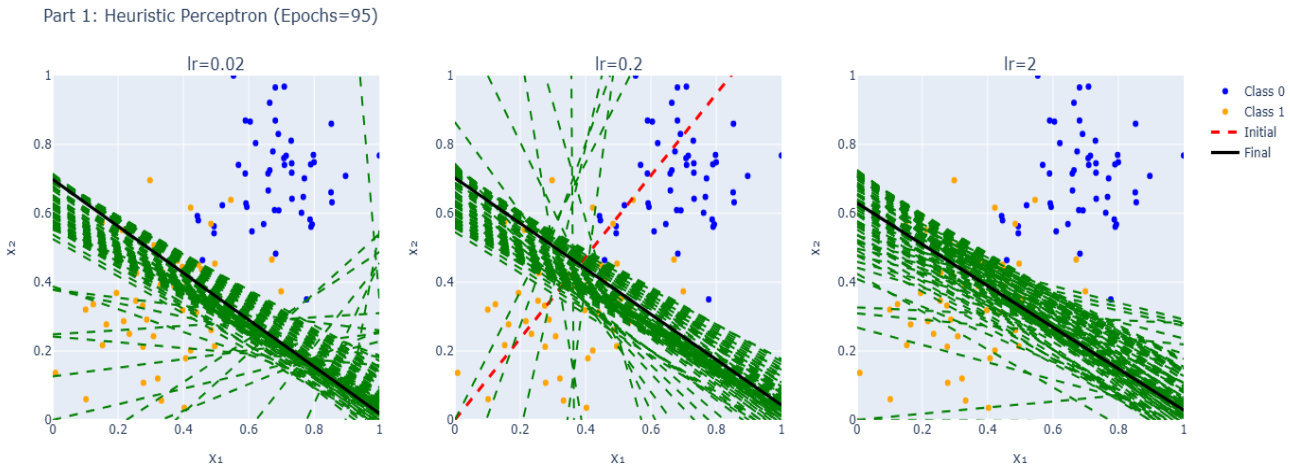
```python
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots

df = pd.read_csv('data.csv', header=None, names=['x1','x2','y'])
X, y = df[['x1','x2']].values, df['y'].values

def train_perceptron(X, y, lr=0.2, epochs=65):
    w = np.random.randn(X.shape[1])
    b = 0.0
    history = [(w.copy(), b)]
    for _ in range(epochs):
        for xi, yi in zip(X, y):
            z = np.dot(w, xi) + b
            pred = 1 if z > 0 else 0
            error = yi - pred
            w += lr * error * xi
            b += lr * error
        history.append((w.copy(), b))
    return history

def plot_all_boundaries(X, y, lrs, epochs):
    fig = make_subplots(rows=1, cols=3, subplot_titles=[f"lr={lr}" for lr in lrs])

    for col, lr in enumerate(lrs, start=1):
        hist = train_perceptron(X, y, lr=lr, epochs=epochs)

        for lbl, color in zip([0,1], ['blue','orange']):
            mask = (y == lbl)
            fig.add_trace(go.Scatter(
                x=X[mask, 0], y=X[mask, 1],
                mode='markers', marker=dict(size=6, color=color),
                name=f'Class {lbl}' if col == 1 else None,
                showlegend=(col == 1)
            ), row=1, col=col)

        for i, (w, b) in enumerate(hist):
```

```
        for i, (w, b) in enumerate(hist):
            x0, x1 = 0.0, 1.0
            if abs(w[1]) > 1e-6:
                y0 = -(w[0] * x0 + b) / w[1]
                y1 = -(w[0] * x1 + b) / w[1]
            else:
                x0 = x1 = -b / w[0]
                y0, y1 = 0.0, 1.0

            if i == 0:
                colr, dash, width, name = 'red', 'dash', 3, 'Initial'
            elif i == len(hist) - 1:
                colr, dash, width, name = 'black', 'solid', 3, 'Final'
            else:
                colr, dash, width, name = 'green', 'dash', 2, None

            fig.add_trace(go.Scatter(
                x=[x0, x1], y=[y0, y1],
                mode='lines',
                line=dict(color=colr, dash=dash, width=width),
                name=name if (col == 1 and name) else None,
                showlegend=(col == 1 and bool(name))
            ), row=1, col=col)

    fig.update_layout(
        title_text=f"Part 1: Heuristic Perceptron (Epochs={epochs})",
        height=500,
        width=1400,
        showlegend=True
    )

    for i in range(1, 4):
        fig.update_xaxes(range=[0, 1], title_text="x₁", row=1, col=i)
        fig.update_yaxes(range=[0, 1], title_text="x₂", row=1, col=i)

    fig.show()

plot_all_boundaries(X, y, lrs=[0.02,0.2, 2], epochs=35)
plot_all_boundaries(X, y, lrs=[0.02,0.2, 2], epochs=65)
plot_all_boundaries(X, y, lrs=[0.02,0.2, 2], epochs=95)
```

This Python script implements and visualizes the training process of a **heuristic perceptron classifier** on a 2D dataset using various learning rates. The perceptron is a simple binary classifier that updates its weights and bias based on prediction error. The dataset is loaded using pandas, with x1 and x2 as input features and y as the binary label. The train_perceptron function initializes random weights and bias, then iterates over the dataset for a specified number of epochs. For each data point, it computes the weighted sum ($z = w \cdot x + b$) and makes a binary prediction using a step function. Based on the error between the true label and predicted output, the weights and bias are updated using the specified learning rate. The model stores the weight and bias after each epoch to track how the decision boundary evolves over time.

To visualize this process, the plot_all_boundaries function is used. It creates subplots (using Plotly's make_subplots) to compare the effect of different learning rates (e.g., 0.02, 0.2, and 2.0). For each learning rate, the perceptron is trained, and both the data points and evolving decision boundaries are plotted. Class 0 and class 1 are shown using blue and orange dots, respectively. The decision boundaries are represented by lines computed from the weight and bias values: the **initial boundary** is red and dashed, **intermediate boundaries** are green and dashed, and the **final boundary** is solid black. The subplot layout is customized to ensure all graphs have the same axis ranges and are displayed side-by-side for easy comparison. Finally, the function is called three times with varying numbers of epochs (35, 65, and 95), allowing for an analysis of how both the **learning rate** and **training duration** affect convergence and boundary adjustment.






# Analysis:

These three sets of graphs illustrate how the **heuristic perceptron** learning process evolves over different combinations of **learning rates (lr)** and **number of epochs (35, 65, and 95)**. Each row corresponds to a different number of training epochs, and each column compares results for different learning rates: 0.02, 0.2, and 2.0. In every subplot, the red dashed line represents the **initial decision boundary**, the green dashed lines are the **intermediate updates**, and the black line marks the **final decision boundary** after training. As the number of epochs increases from 35 to 95, we observe more refinement and better alignment of the decision boundary with the data separation, particularly at moderate learning rates. With **low learning rate (0.02)**, the model takes small steps, requiring more epochs to converge and resulting in many fluctuating green lines. With a **moderate learning rate (0.2)**, the model converges efficiently with fewer updates, producing smoother transitions and more stable final boundaries. In contrast, a **high learning rate (2.0)** accelerates convergence but may cause aggressive updates and oscillations, especially with fewer epochs. As the epochs increase, all learning rates eventually stabilize, but the path to convergence varies, highlighting the importance of balancing learning rate and training duration.

# Part 2:

Code Snippet

```python
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Load dataset
df = pd.read_csv('data.csv', header=None, names=['x1', 'x2', 'y'])
X, y = df[['x1', 'x2']].values, df['y'].values

def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def train_logistic(X, y, lr=0.1, epochs=100):
    w = np.random.randn(X.shape[1])
    b = 0.0
    history = [(w.copy(), b)]
    losses = []

    for _ in range(epochs):
        for xi, yi in zip(X, y):
            z = np.dot(w, xi) + b
            y_hat = sigmoid(z)
            error = yi - y_hat
            w += lr * error * xi
            b += lr * error
        history.append((w.copy(), b))

        # Log-loss
        z_all = X.dot(w) + b
        y_hat_all = sigmoid(z_all)
        loss = -np.mean(y * np.log(y_hat_all + 1e-9) + (1 - y) * np.log(1 - y_hat_all + 1e-9))
        losses.append(loss)

    return history, losses

# Subplot for boundaries
def plot_multiple_boundaries(X, y, lrs, epochs):
    fig = make_subplots(rows=1, cols=3, subplot_titles=[f"lr={lr}" for lr in lrs])

    for i, lr in enumerate(lrs, start=1):
        hist, _ = train_logistic(X, y, lr=lr, epochs=epochs)
```

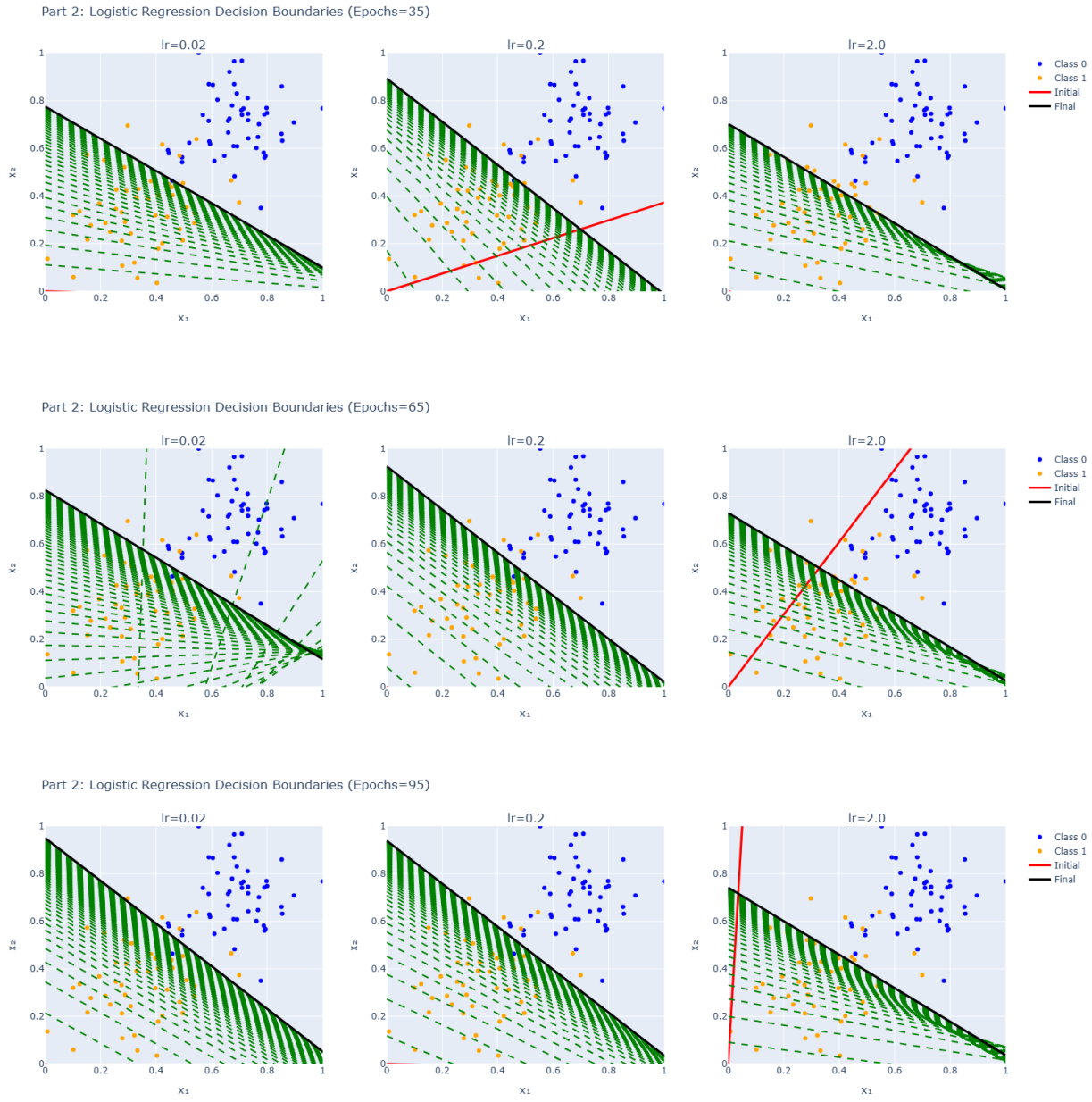
```python
# Subplot for boundaries
def plot_multiple_boundaries(X, y, lrs, epochs):
    fig = make_subplots(rows=1, cols=3, subplot_titles=[f"lr={lr}" for lr in lrs])

    for i, lr in enumerate(lrs, start=1):
        hist, _ = train_logistic(X, y, lr=lr, epochs=epochs)

        for lbl, color in zip([0, 1], ['blue', 'orange']):
            mask = (y == lbl)
            fig.add_trace(go.Scatter(
                x=X[mask, 0], y=X[mask, 1],
                mode='markers', marker=dict(size=6, color=color),
                name=f'Class {lbl}' if i == 1 else None,
                showlegend=(i == 1)
            ), row=1, col=i)

        for j, (w, b) in enumerate(hist):
            x0, x1 = 0.0, 1.0
            if abs(w[1]) > 1e-6:
                y0 = -(w[0] * x0 + b) / w[1]
                y1 = -(w[0] * x1 + b) / w[1]
            else:
                x0 = x1 = -b / w[0]; y0, y1 = 0.0, 1.0

            if j == 0:
                colr, dash, width, name = 'red', 'solid', 3, 'Initial'
            elif j == len(hist) - 1:
                colr, dash, width, name = 'black', 'solid', 3, 'Final'
            else:
                colr, dash, width, name = 'green', 'dash', 2, None

            fig.add_trace(go.Scatter(
                x=[x0, x1], y=[y0, y1],
                mode='lines',
                line=dict(color=colr, dash=dash, width=width),
                name=name if i == 1 and name else None,
                showlegend=(i == 1 and bool(name))
            ), row=1, col=i)
```

```
    fig.update_layout(
        title=f"Part 2: Logistic Regression Decision Boundaries (Epochs={epochs})",
        height=500, width=1500
    )

    for i in range(1, 4):
        fig.update_xaxes(range=[0, 1], title_text="x₁", row=1, col=i)
        fig.update_yaxes(range=[0, 1], title_text="x₂", row=1, col=i)

    fig.show()

# Subplot for log loss plots
def plot_multiple_losses(X, y, lrs, epochs):
    fig = make_subplots(rows=1, cols=3, subplot_titles=[f"lr={lr}" for lr in lrs])

    for i, lr in enumerate(lrs, start=1):
        _, losses = train_logistic(X, y, lr=lr, epochs=epochs)
        fig.add_trace(go.Scatter(
            x=list(range(1, len(losses)+1)),
            y=losses,
            mode='lines+markers',
            name=f'lr={lr}',
            marker=dict(color='blue')
        ), row=1, col=i)

    fig.update_layout(
        title_text=f"Part 2: Error Plot (Epochs={epochs})",
        height=500, width=1500
    )

    for i in range(1, 4):
        fig.update_xaxes(title_text="Epoch", row=1, col=i)
        fig.update_yaxes(title_text="Log Loss", row=1, col=i)

    fig.show()

lrs = [0.02, 0.2, 2.0]
plot_multiple_boundaries(X, y, lrs, 35)
plot_multiple_losses(X, y, lrs, 35)
plot_multiple_boundaries(X, y, lrs, 65)
plot_multiple_losses(X, y, lrs, 65)
plot_multiple_boundaries(X, y, lrs, 95)
plot_multiple_losses(X, y, lrs, 95)
```

This Python code implements a **logistic regression model using gradient descent**, visualizing how the decision boundary and log-loss evolve across different learning rates and epochs. It begins by loading a dataset containing two features (x1, x2) and a binary label (y). The sigmoid function is defined to convert linear outputs into probabilities. The train_logistic function initializes random weights and bias, then iteratively updates them using the gradient of the log-loss function. For each epoch, the model predicts outputs using the sigmoid function, calculates the prediction error, and adjusts the weights and bias accordingly. After each epoch, the log-loss is computed across the entire dataset and stored for later visualization.

To analyze performance, the script contains two main plotting functions. The plot_multiple_boundaries function uses Plotly's subplot features to compare decision boundary

evolution for three different learning rates (0.02, 0.2, and 2.0) across a consistent number of epochs. Each subplot shows the training progression with red (initial), green (intermediate), and black (final) decision boundaries, along with class-labeled data points. The plot_multiple_losses function then plots the corresponding log-loss values over epochs, visually tracking how quickly and effectively the model converges. Each learning rate is tested across 35, 65, and 95 epochs, allowing for a comprehensive comparison of how both learning rate and training duration affect model accuracy and stability. This code highlights the sensitivity of gradient descent training to hyperparameters and provides visual tools to interpret training dynamics.



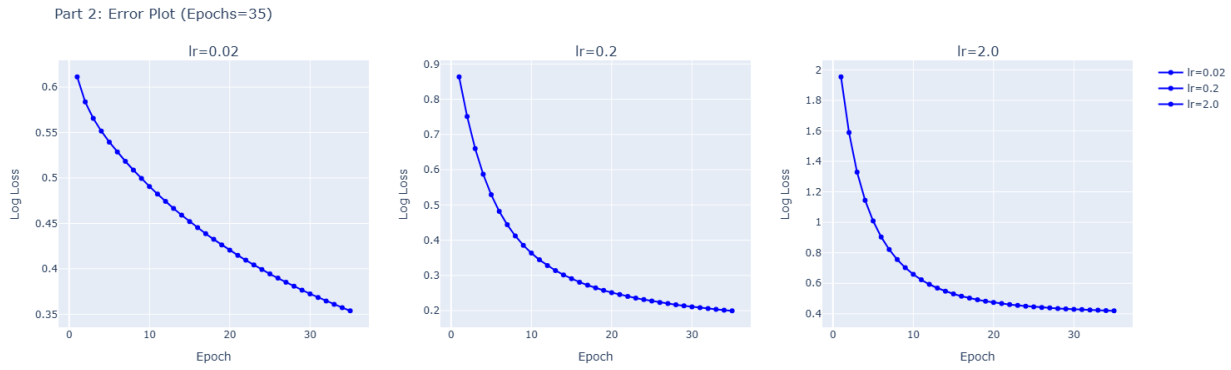
# Analysis:

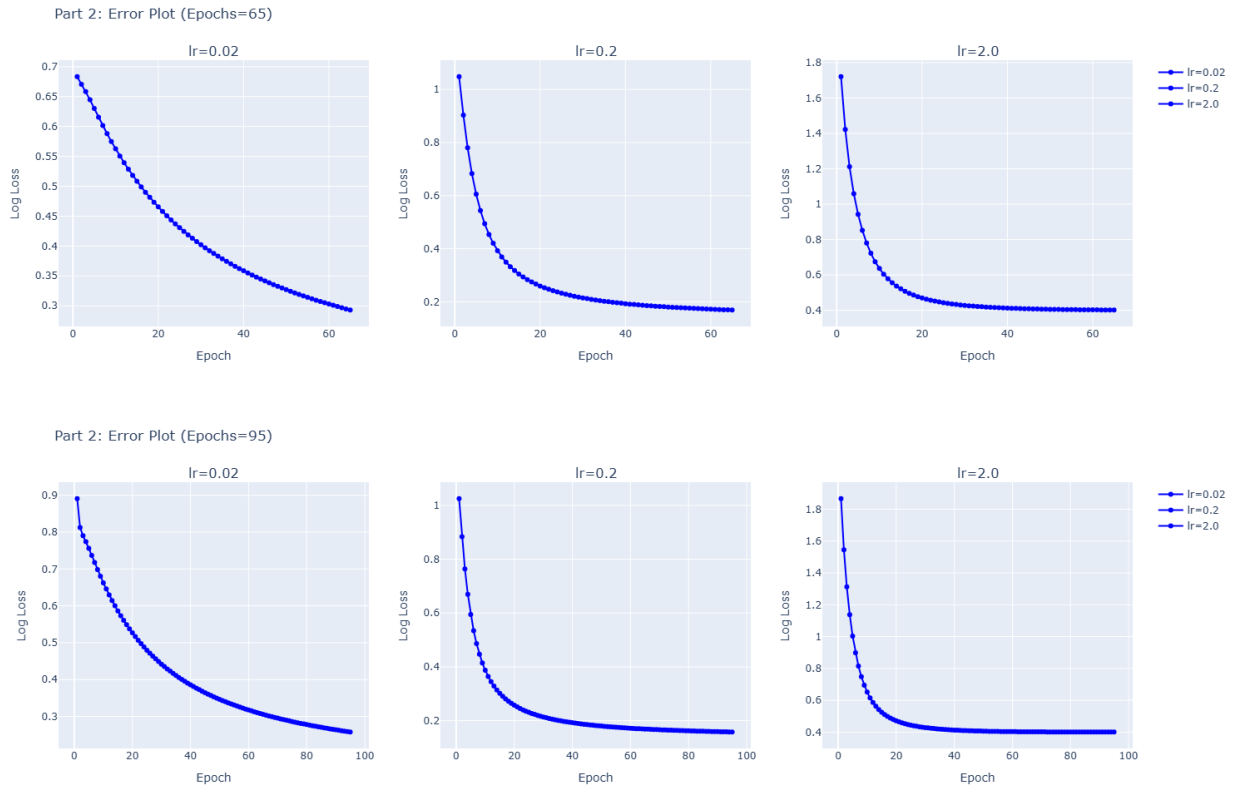
These graphs show the evolution of **logistic regression decision boundaries** using **gradient descent** across different **learning rates (lr)** and **epochs (35, 65, 95)**. Each row represents a different epoch count, and each column corresponds to a different learning rate (lr = 0.02, lr = 0.2, and lr = 2.0). In all subplots, the **red dashed line** shows the initial decision boundary, the **green dashed lines** represent updates made during training, and the **black line** indicates the final learned boundary.

As the number of epochs increases, the boundaries progressively converge toward a clear separation between Class 0 (blue) and Class 1 (orange). For **lr = 0.02**, the learning is very gradual. At 35 epochs, the boundary is close to the initial position, and even by 95 epochs, the updates are small, though convergence improves steadily. With **lr = 0.2**, the model converges faster. By 65 epochs, the boundary is already well aligned, and at 95, it's nearly perfect—this rate shows a good balance of speed and stability. With **lr = 2.0**, the learning is aggressive. Although convergence appears rapid by epoch 35, the initial updates can be large and erratic, as seen by the steep red line and some widely spaced green lines. Despite this, the model manages to stabilize a good final decision boundary due to the simplicity of the data.

In summary, **lower learning rates** need more epochs to reach convergence, while **moderate learning rates** (like 0.2) offer fast and stable learning. **Very high learning rates** (like 2.0) can still work, but risk instability or overshooting, especially in early training stages. This comparison highlights the importance of choosing the right learning rate for optimal model performance.

# Error Plot:






These log-loss plots represent the error reduction over training epochs for logistic regression using gradient descent across three different learning rates (0.02, 0.2, and 2.0) and three different epoch settings (35, 65, and 95). Each row of graphs corresponds to a specific epoch setting, and each column compares how the learning rate affects the log-loss over time.

For all learning rates, the log-loss decreases with more epochs, indicating that the model is learning and improving its predictions. At lr = 0.02, the loss decreases gradually and steadily. Even after 95 epochs, it continues to improve but at a slower pace, showing that this rate is stable but slow to converge. For lr = 0.2, the log-loss drops much faster in the early epochs and quickly levels out to a low value, demonstrating efficient and stable convergence. This rate achieves a good balance between speed and reliability. In contrast, lr = 2.0 starts with a very high loss and decreases rapidly in the first few epochs, but stabilizes earlier than the others at a slightly higher final loss value. This suggests that while high learning rates can accelerate training, they may lead to less precise convergence or overshooting, especially in noisier data scenarios.

Overall, the plots clearly show that a moderate learning rate like 0.2 offers the best performance, combining fast convergence with stable error minimization. The effectiveness of learning rates and the diminishing return of extended epochs are well illustrated in these visualizations.