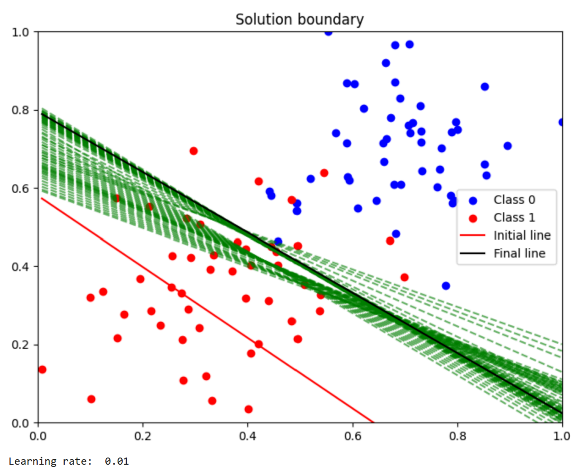
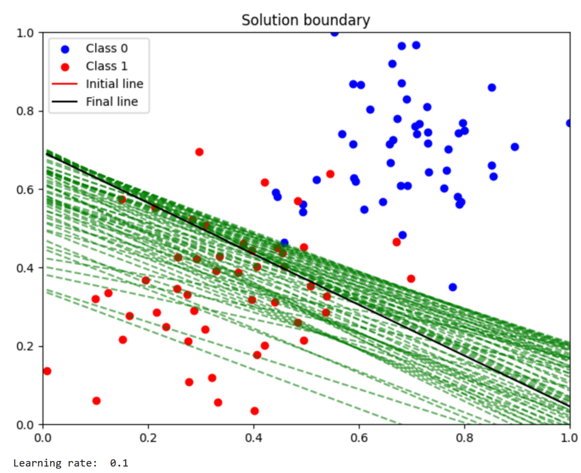
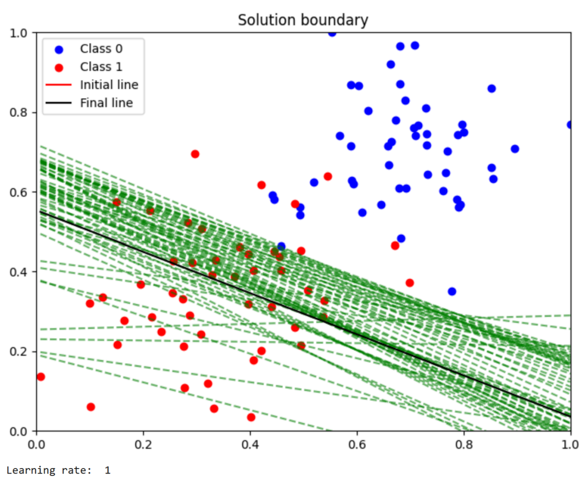
Part 1

```python
plot(w, b, 0)

for iter in range(1, iteration+1):
    for i in range(len(X)):
        x_i = X[i]
        target = y[i]
        prediction = 1 if (np.dot(w, x_i) + b) > 0 else 0

        if prediction != target:
            update = r * x_i
            if prediction == 0:
                b += r
                w += update
            else:
                b -= r
                w -= update

    plot(w, b, iter)
```

Main loop snippet





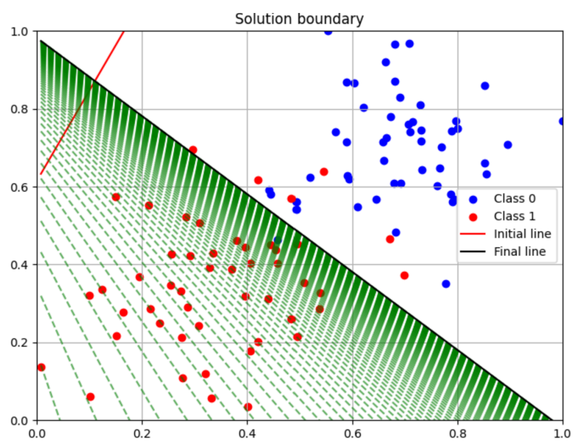


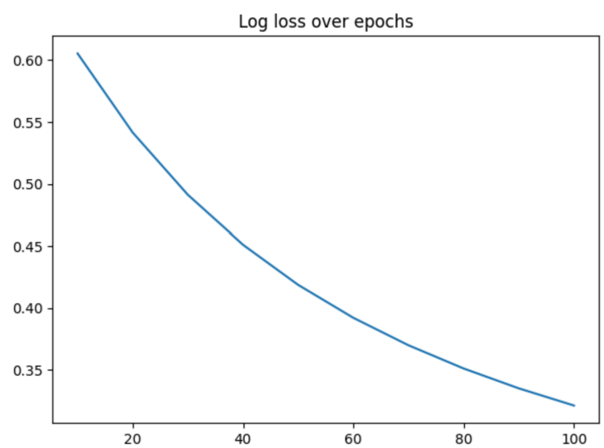
According to these results, a lower learning rate may produce a more accurate final weight and bias for a given dataset. Of course, this isn’t guaranteed as the initial weights and bias are randomized for each run.

Part 2

```
plot(w, b, 0)

log_losses = []

for epoch in range(1, epochs+1):
    y_preds = []

    for i in range(len(X)):
        x_i = X[i]
        target = y[i]

        z = np.dot(w, x_i) + b
        y_hat = sigmoid(z)
        error = target - y_hat

        w += r * error * x_i
        b += r * error
        y_preds.append(y_hat)

    if epoch % 10 == 0:
        log_loss = calc_log_loss(y, np.array(y_preds))
        log_losses.append((epoch, log_loss))

    plot(w, b, epoch)
```

Gradient descent loop snippet





Compared to the part 1 results, the final line here seems to be the most accurate in dividing the red and blue points. We can also see from the log loss over epochs that the error rate decreases as the epochs increase (x-axis = # of epochs, y-axis = error).