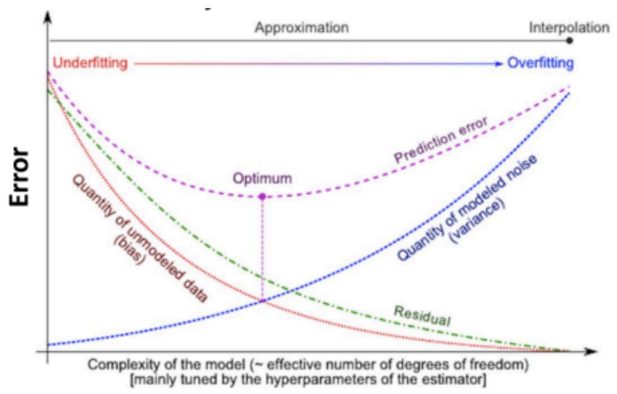
SECTION - A

(a) As the complexity of a machine-learning model increases, for example, by adding more features or higher-order polynomial terms, the model tends to **overfit** the training data. This can be explained in terms of **bias** and **variance**:

- **Bias**: Bias refers to the error introduced by overly simplistic assumptions in the model. In a low-complexity model (e.g., linear models with few features), the bias is high because the model is unable to capture all the underlying patterns, leading to underfitting. As model complexity increases, bias decreases since the model better fits the training data.
- **Variance**: Variance measures how much the model's predictions change when trained on different datasets. In a highly complex model (e.g., a polynomial model with many features), variance is high because the model becomes overly sensitive to small fluctuations in the training data, leading to overfitting. As complexity increases, the variance also increases.



Thus, increasing model complexity typically leads to **lower bias but higher variance**. This creates a trade-off, known as the **bias-variance trade-off**. A model that is too simple has high bias and underfits, while a model that is too complex has high variance and overfits.

Graphical representation.

(b) Given:
True Positives (TP): 200
False Negatives (FN): 50
True Negatives (TN): 730
False Positives (FP): 20

Precision = $\frac{TP}{TP+FP}$ = $\frac{200}{200+20} = 0.9090$

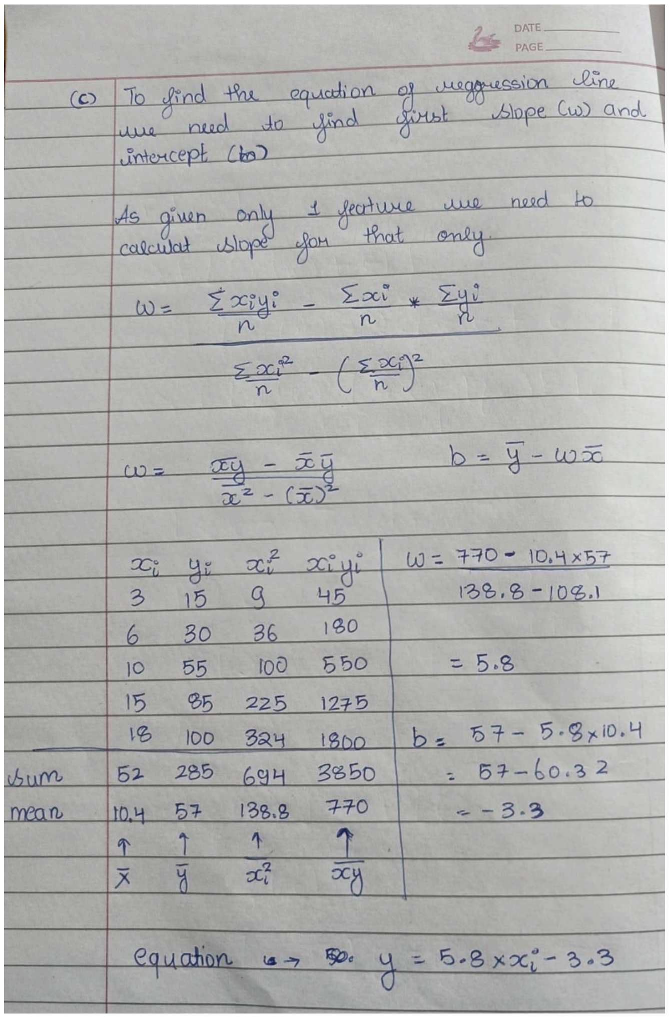
Accuracy = $\frac{TP+TN}{TP+TN+FP+FN} = \frac{200+730}{200+50+730+20} = 0.93$

Recall = $\frac{TP}{TP+FN} = \frac{200}{200+50} = \frac{200}{250} = 0.8$

F1 Score = $2 * \frac{Precision \times Recall}{Precision + Recall} = 2\,X\,\frac{0.909\times0.8}{0.909+0..8} = 0.851$

Precision: 0.909 (90.9%)
Recall: 0.800 (80%)
Accuracy: 0.930 (93%)
F1 Score: 0.851 (85.1%)

(c)



DATE
PAGE

(c) To find the equation of reggression line we need to find first slope (w) and intercept (b)

As given only 1 feature we need to calculat slope for that only.

$$w = \frac{\frac{\sum x_i y_i}{n} - \frac{\sum x_i}{n} * \frac{\sum y_i}{n}}{\frac{\sum x_i^2}{n} - \left(\frac{\sum x_i}{n}\right)^2}$$

$$w = \frac{\overline{xy} - \bar{x}\bar{y}}{\overline{x^2} - (\bar{x})^2} \qquad b = \bar{y} - w\bar{x}$$

	x_i	y_i	x_i^2	$x_i y_i$
	3	15	9	45
	6	30	36	180
	10	55	100	550
	15	85	225	1275
	18	100	324	1800
sum	52	285	694	3850
mean	10.4	57	138.8	770
	$\uparrow$ $\bar{x}$	$\uparrow$ $\bar{y}$	$\uparrow$ $\overline{x_i^2}$	$\uparrow$ $\overline{xy}$

$$w = \frac{770 - 10.4\times57}{138.8 - 108.1}$$

$= 5.8$

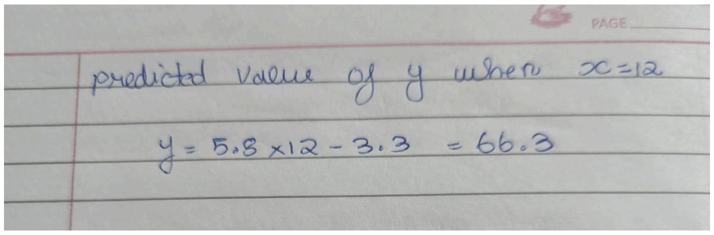
$b = 57 - 5.8\times10.4$

$= 57 - 60.32$

$= -3.3$

equation $\rightarrow$ $y = 5.8 \times x_i - 3.3$

equation: y = 5.8xi -3.3

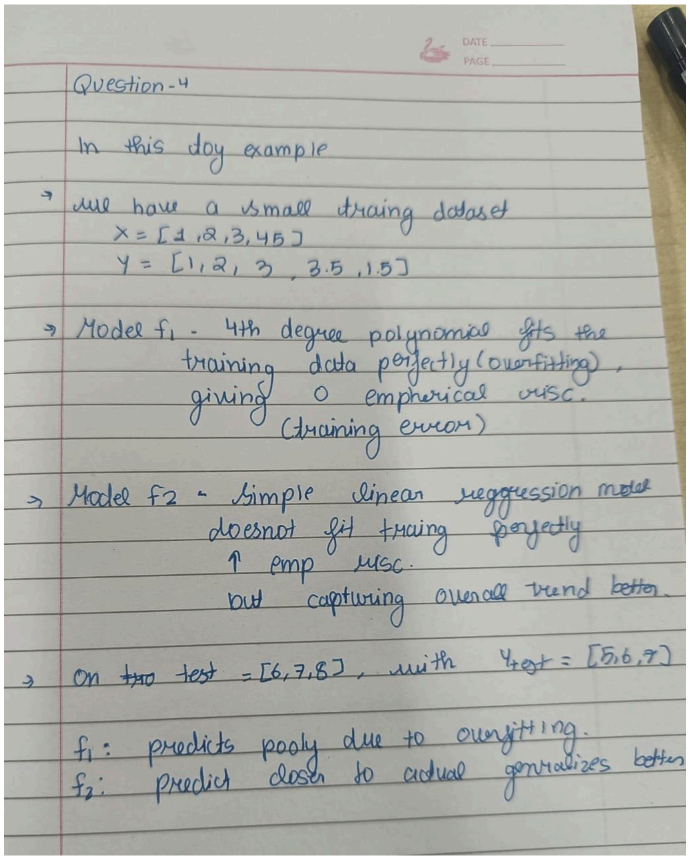


predicted value of y when $x = 12$

$y = 5.8 \times 12 - 3.3 = 66.3$

predicted value of y when x = 12
Y = 66.3

(d)



Question-4

In this doy example

→ we have a small traing dataset
$X = [1, 2, 3, 45]$
$Y = [1, 2, 3, 3.5, 1.5]$

→ Model f_1 - 4th degree polynomial fits the training data perfectly (overfitting), giving 0 emphrical risc. (training error)

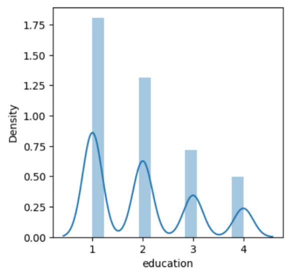
→ Model f2 - simple linear reggression model doesnot fit traing perfectly
↑ emp risc.
but capturing overall trend better.

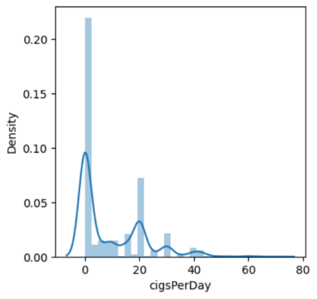
→ On ~~two~~ test = [6, 7, 8], with $Y_{tgt} = [5, 6, 7]$

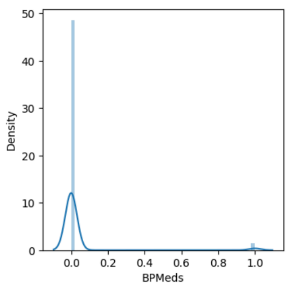
f_1: predicts pooly due to overfitting.
f_2: predict closer to actual generalizes better

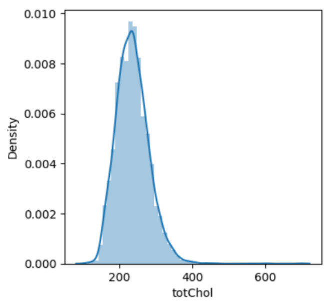
SECTION B

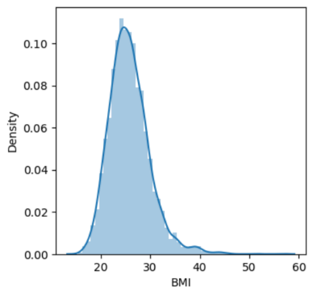
- First we try to clean our data as our heart disease data contain null values we try to fill them with their mean, median and mode according to their distribution.

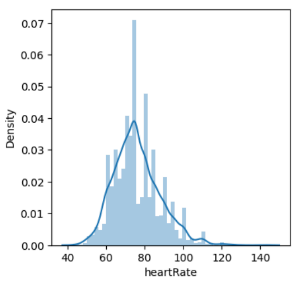












After filling the null values we now scale the data i.e. normalize the data.

Part (a)

Implementation of logistic regression from scratch

- **Sigmoid function**
  This function computes the sigmoid of the input `z`. The sigmoid function is commonly used in logistic regression to map any real number to a value between 0 and 1, making it suitable for binary classification problems. The sigmoid transformation of the input.

- **Compute cost function**

Computes the cost (or loss) of logistic regression using the current model parameters w and b. The function iterates over all the training examples to compute the cross-entropy loss, which measures how well the model's predictions align with the actual labels y.

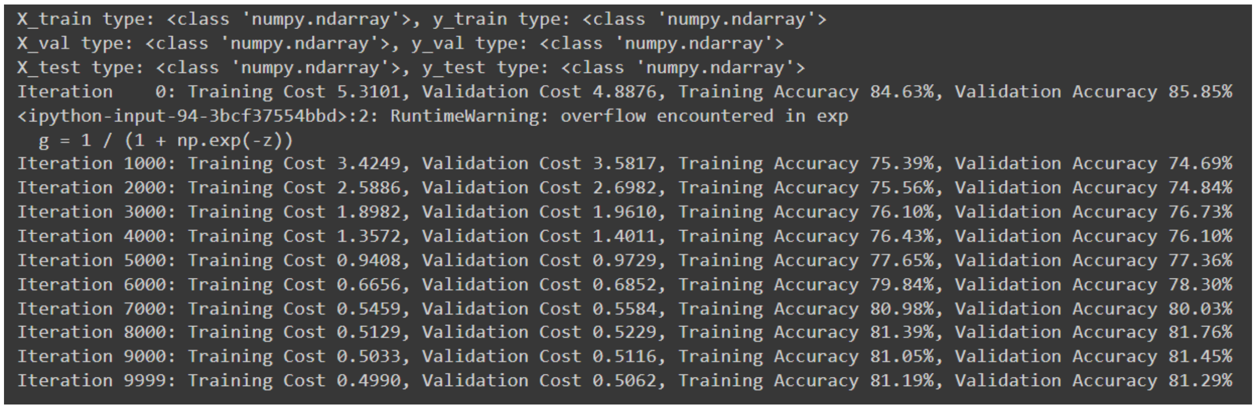
- **Compute gradient**
  Computes the gradients of the cost function with respect to the model parameters `w` (weights) and `b` (bias). The gradients (derivatives) are essential for updating the model parameters using gradient descent. This function computes both the gradient for `w` and `b` by iterating over the dataset and calculating how each feature and sample contributes to the cost. And it helps to find the derivative term in the gradient descent term .

- **Gradient descendent**
  Performs gradient descent optimization to minimize the cost function and update the model parameters over multiple iterations.
  The function uses the gradients computed by `gradient_function` to update the weights `w_in` and bias `b_in`.
  It tracks and logs the training and validation costs, as well as accuracies, at each iteration.
  By adjusting the learning rate `alpha` and the number of iterations `num_iters`, the model converges to an optimal solution.
  The `accuracy_function` is used to monitor the model's performance during the training process.
  It returns Updated weights and bias.
  Lists of training and validation losses.
  Lists of training and validation accuracies.

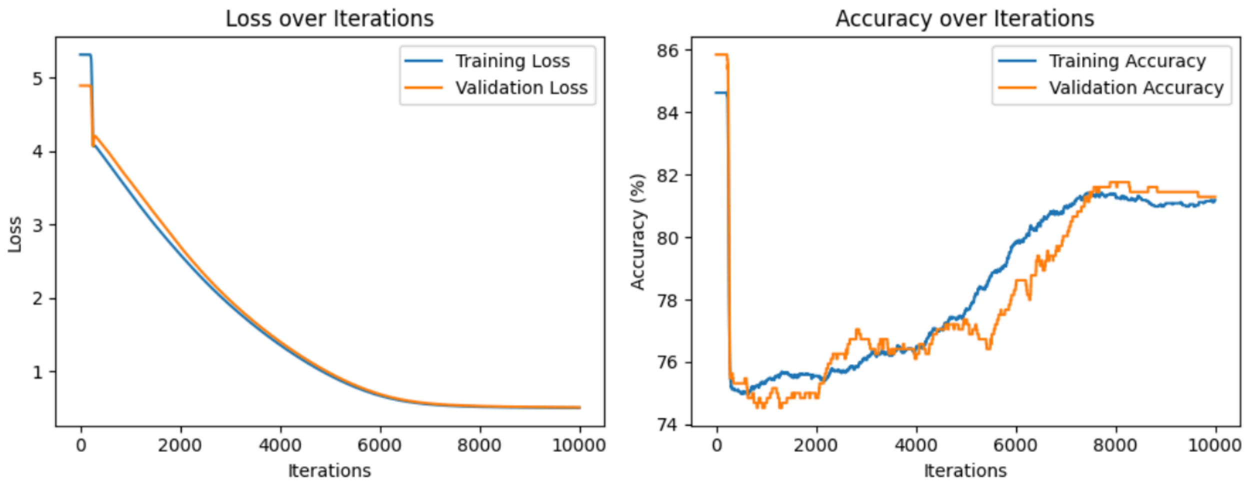
- **Accuracy function**
  Computes the accuracy of the logistic regression model on a given dataset. The function applies the learned weights `w` and bias `b` to compute predictions using the sigmoid function. The predictions are then thresholded at 0.5 to classify samples as either class 0 or 1. The accuracy is computed by comparing the predicted labels with the true labels.

After defining these functions we define the appropriate hyperparameters. And then call the gradient descendent function which returns the parameters w, b and losses and accuracy.

In this part we are not normalizing or scaling the data.



```
X_train type: <class 'numpy.ndarray'>, y_train type: <class 'numpy.ndarray'>
X_val type: <class 'numpy.ndarray'>, y_val type: <class 'numpy.ndarray'>
X_test type: <class 'numpy.ndarray'>, y_test type: <class 'numpy.ndarray'>
Iteration    0: Training Cost 5.3101, Validation Cost 4.8876, Training Accuracy 84.63%, Validation Accuracy 85.85%
<ipython-input-94-3bcf37554bbd>:2: RuntimeWarning: overflow encountered in exp
  g = 1 / (1 + np.exp(-z))
Iteration 1000: Training Cost 3.4249, Validation Cost 3.5817, Training Accuracy 75.39%, Validation Accuracy 74.69%
Iteration 2000: Training Cost 2.5886, Validation Cost 2.6982, Training Accuracy 75.56%, Validation Accuracy 74.84%
Iteration 3000: Training Cost 1.8982, Validation Cost 1.9610, Training Accuracy 76.10%, Validation Accuracy 76.73%
Iteration 4000: Training Cost 1.3572, Validation Cost 1.4011, Training Accuracy 76.43%, Validation Accuracy 76.10%
Iteration 5000: Training Cost 0.9408, Validation Cost 0.9729, Training Accuracy 77.65%, Validation Accuracy 77.36%
Iteration 6000: Training Cost 0.6656, Validation Cost 0.6852, Training Accuracy 79.84%, Validation Accuracy 78.30%
Iteration 7000: Training Cost 0.5459, Validation Cost 0.5584, Training Accuracy 80.98%, Validation Accuracy 80.03%
Iteration 8000: Training Cost 0.5129, Validation Cost 0.5229, Training Accuracy 81.39%, Validation Accuracy 81.76%
Iteration 9000: Training Cost 0.5033, Validation Cost 0.5116, Training Accuracy 81.05%, Validation Accuracy 81.45%
Iteration 9999: Training Cost 0.4990, Validation Cost 0.5062, Training Accuracy 81.19%, Validation Accuracy 81.29%
```



convergence of the model:

Loss over iterations:
I observed that Training Loss and Validation Loss both follow a similar downward trajectory, which indicates that the model is learning well. Initially, at iteration 0, the losses start high (around 5.31 for training and 4.88 for validation), but as the iterations increase, both losses steadily decrease. At **iteration 9999**, the training loss is around **0.499** and validation loss is **0.506**, showing that both losses have converged and the model generalizes well to unseen data.

From this we can conclude that the decreasing loss over time is a positive sign, indicating that the model is fitting the data properly without overfitting or underfitting.

Accuracy over Iterations:
Training Accuracy and Validation Accuracy exhibit significant fluctuations early on. For instance, at iteration 0, training accuracy is **84.63%** and validation accuracy is **85.85%**, but both drop steeply and then recover over time. After about 6000 iterations, the accuracy stabilizes. By

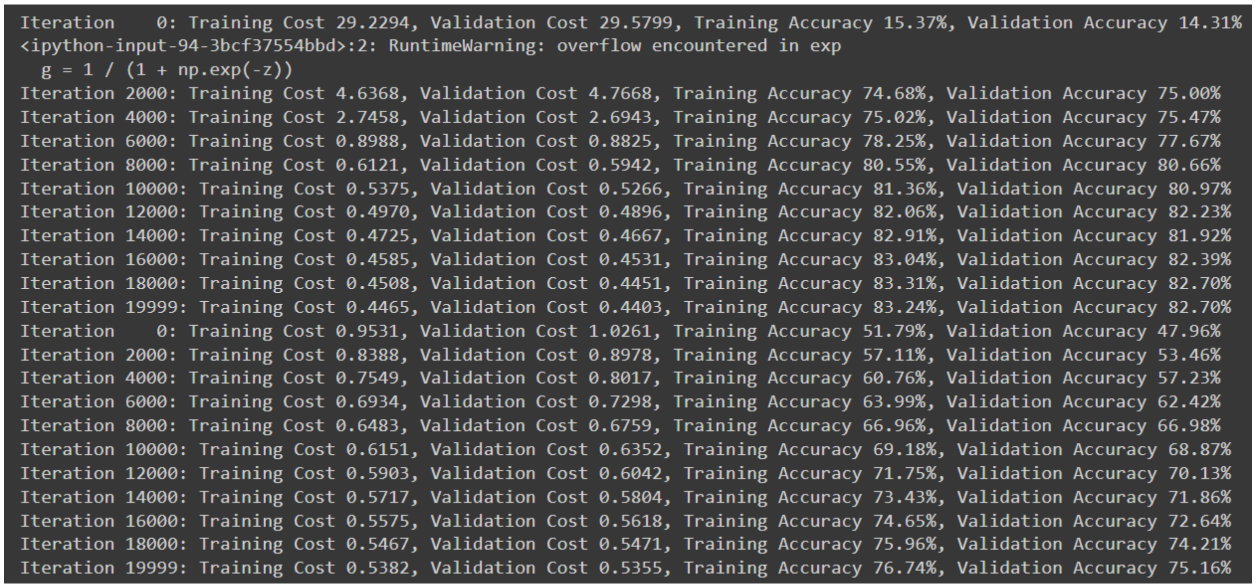
iteration 9999, training accuracy reaches **81.19%**, while validation accuracy is **81.29%**, with both curves showing that the model is performing consistently on both datasets.

Part (b)

Min-Max scaling function

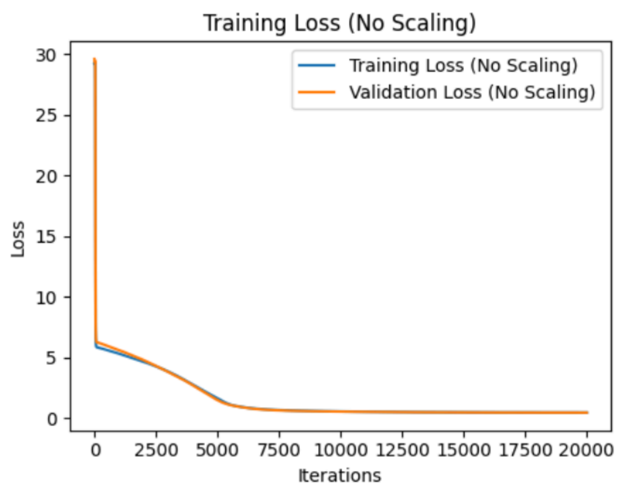
The min_max_scaling function applies Min-Max scaling to the feature matrix X, while also clipping extreme values to reduce the impact of outliers.

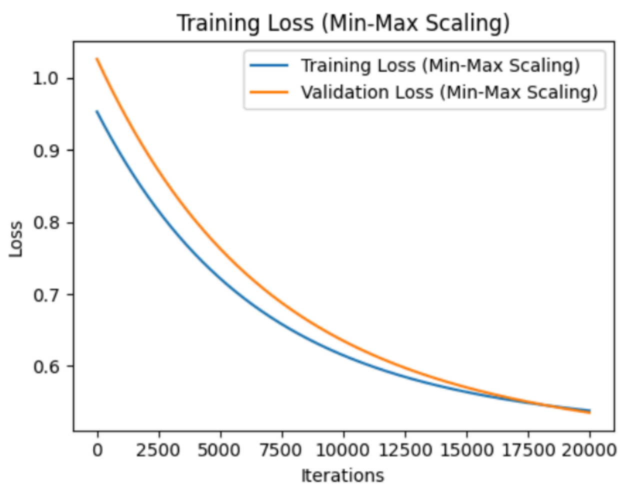
- Outliers are extreme values that can distort or skew the scaling process. For instance, a few very large or small values can affect how the Min-Max scaling stretches the range of the feature. Clipping limits the influence of these extreme values, making the scaling process more robust.The function calculates the lower and upper percentiles (default 5th and 95th) for each feature to create X_min and X_max. This clips extreme values, mitigating the effect of outliers.
- After clipping, the function applies Min-Max scaling to rescale the features between 0 and 1. And epsilon ensures there's no division by zero when the range is zero.

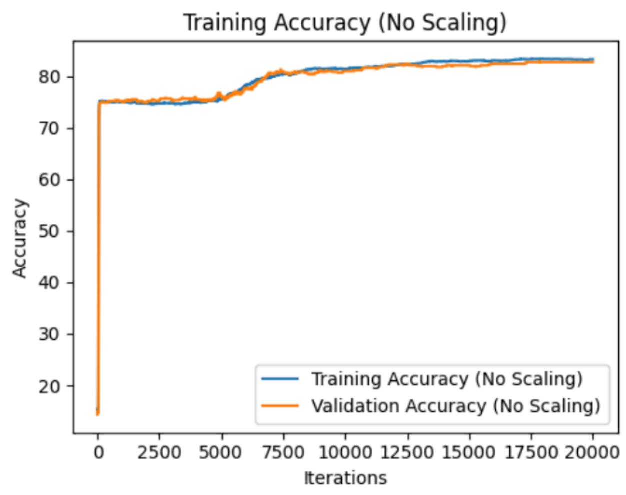


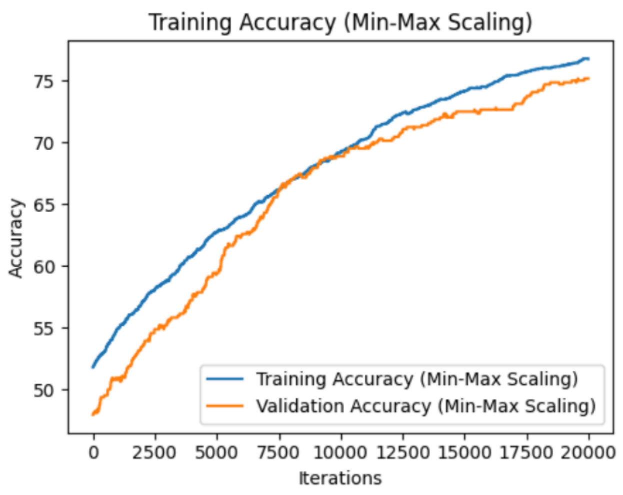
```
Iteration    0: Training Cost 29.2294, Validation Cost 29.5799, Training Accuracy 15.37%, Validation Accuracy 14.31%
<ipython-input-94-3bcf37554bbd>:2: RuntimeWarning: overflow encountered in exp
  g = 1 / (1 + np.exp(-z))
Iteration 2000: Training Cost 4.6368, Validation Cost 4.7668, Training Accuracy 74.68%, Validation Accuracy 75.00%
Iteration 4000: Training Cost 2.7458, Validation Cost 2.6943, Training Accuracy 75.02%, Validation Accuracy 75.47%
Iteration 6000: Training Cost 0.8988, Validation Cost 0.8825, Training Accuracy 78.25%, Validation Accuracy 77.67%
Iteration 8000: Training Cost 0.6121, Validation Cost 0.5942, Training Accuracy 80.55%, Validation Accuracy 80.66%
Iteration 10000: Training Cost 0.5375, Validation Cost 0.5266, Training Accuracy 81.36%, Validation Accuracy 80.97%
Iteration 12000: Training Cost 0.4970, Validation Cost 0.4896, Training Accuracy 82.06%, Validation Accuracy 82.23%
Iteration 14000: Training Cost 0.4725, Validation Cost 0.4667, Training Accuracy 82.91%, Validation Accuracy 81.92%
Iteration 16000: Training Cost 0.4585, Validation Cost 0.4531, Training Accuracy 83.04%, Validation Accuracy 82.39%
Iteration 18000: Training Cost 0.4508, Validation Cost 0.4451, Training Accuracy 83.31%, Validation Accuracy 82.70%
Iteration 19999: Training Cost 0.4465, Validation Cost 0.4403, Training Accuracy 83.24%, Validation Accuracy 82.70%
Iteration    0: Training Cost 0.9531, Validation Cost 1.0261, Training Accuracy 51.79%, Validation Accuracy 47.96%
Iteration 2000: Training Cost 0.8388, Validation Cost 0.8978, Training Accuracy 57.11%, Validation Accuracy 53.46%
Iteration 4000: Training Cost 0.7549, Validation Cost 0.8017, Training Accuracy 60.76%, Validation Accuracy 57.23%
Iteration 6000: Training Cost 0.6934, Validation Cost 0.7298, Training Accuracy 63.99%, Validation Accuracy 62.42%
Iteration 8000: Training Cost 0.6483, Validation Cost 0.6759, Training Accuracy 66.96%, Validation Accuracy 66.98%
Iteration 10000: Training Cost 0.6151, Validation Cost 0.6352, Training Accuracy 69.18%, Validation Accuracy 68.87%
Iteration 12000: Training Cost 0.5903, Validation Cost 0.6042, Training Accuracy 71.75%, Validation Accuracy 70.13%
Iteration 14000: Training Cost 0.5717, Validation Cost 0.5804, Training Accuracy 73.43%, Validation Accuracy 71.86%
Iteration 16000: Training Cost 0.5575, Validation Cost 0.5618, Training Accuracy 74.65%, Validation Accuracy 72.64%
Iteration 18000: Training Cost 0.5467, Validation Cost 0.5471, Training Accuracy 75.96%, Validation Accuracy 74.21%
Iteration 19999: Training Cost 0.5382, Validation Cost 0.5355, Training Accuracy 76.74%, Validation Accuracy 75.16%
```

I have trained my model using two different settings: first with **no scaling** on the data, and then with **Min-Max scaling** applied. I plotted both the **training and validation loss** as well as the **training and validation accuracy** for each scenario over 20,000 iterations.









In the above plots, we can see the training loss, validation loss, training accuracy, and validation accuracy with and without Min-Max scaling.

1) Loss over Iterations Analysis:

   Without min-max scaling
   The training started with a very high loss (around 30) for both the training and validation sets. This indicates that the model initially struggled to fit the data, which is common when the features are not scaled. Then around 5000 iterations the loss decreases. The training and validation losses remained very close to each other, showing good generalization. At the end the loss reaches to almost 0 indicating that it has minimized the error.

   With min-max scaling

The loss values for both training and validation decreased steadily, showing a smooth learning process. Then by the 20000 th iteration the loss has reduced further which implies that model is working fine.

2) Accuracy over Iterations analysis:

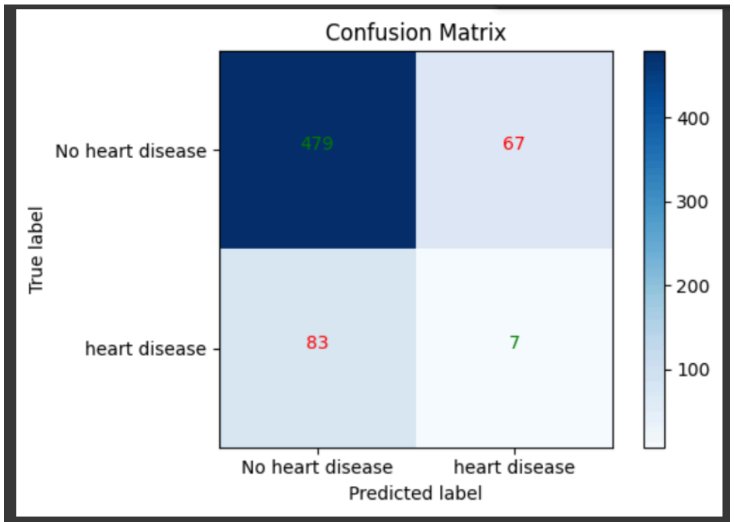
   Without Scaling:
   Initially, the accuracy was very low for both training and validation, indicating that the model was not predicting well in the early stages. Then from the 5000th iteration accuracy started becoming constant. The final accuracy reached about **81%** for both training and validation, showing that the model learned well without scaling.

   With scaling:
   The training started with a much higher accuracy compared to the no-scaling scenario. The model achieved over 50% accuracy within the first 2000 iterations. Unlike the no-scaling scenario where accuracy shot up quickly, the accuracy in this case increased more gradually over the iterations.

Impact of feature scaling on convergence:
Faster convergence, Improved stability, Reduced overfitting risk.

Part (c)

For unscaled data



```
Accuracy: 0.76
Precision: 0.09
Recall: 0.08
F1 Score: 0.09
```

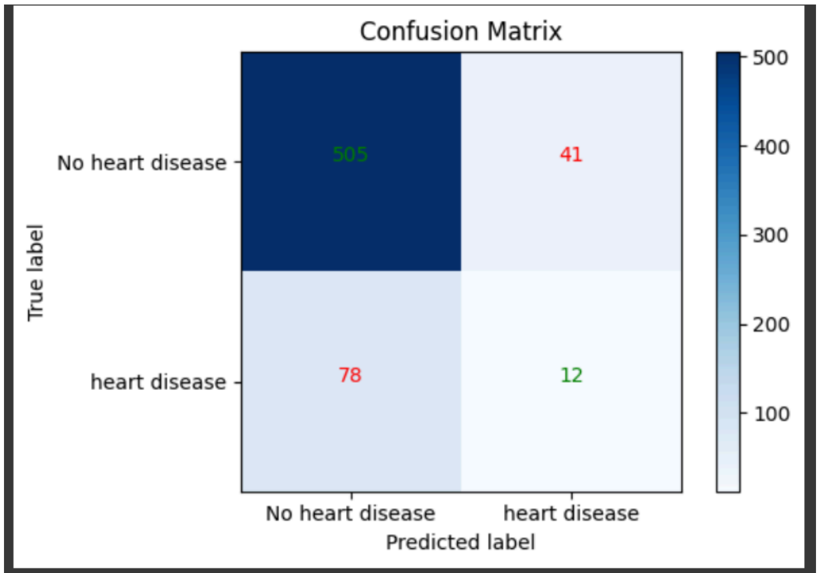
The above confusion matrix and metrics (Accuracy, Precision, Recall, F1 Score) provide important insights into the model's performance for predicting heart disease.

True Negatives = 479

False Positives = 67
False Negatives = 83
True Positives = 7

1) Accuracy: Accuracy indicates the proportion of correct predictions (both true positives and true negatives) out of the total number of cases. In this case, the model correctly classified 76% of the cases.
2) Precision: Precision measures the proportion of correctly predicted "heart disease" cases (true positives) out of all the cases predicted as "heart disease."
3) Recall: Recall measures the ability of the model to correctly identify true "heart disease" cases (true positives).
4) F1 score: The F1 Score is the harmonic mean of Precision and Recall. It provides a balance between the two metrics, especially useful for imbalanced datasets. The low F1 score (9%) reflects poor overall performance in identifying "heart disease," due to both low precision and recall.

Confusion matrix for Unscaled data.



```
Accuracy: 0.81
Precision: 0.23
Recall: 0.13
F1 Score: 0.17
```

Part (d)

Mini-batch gradient descendent

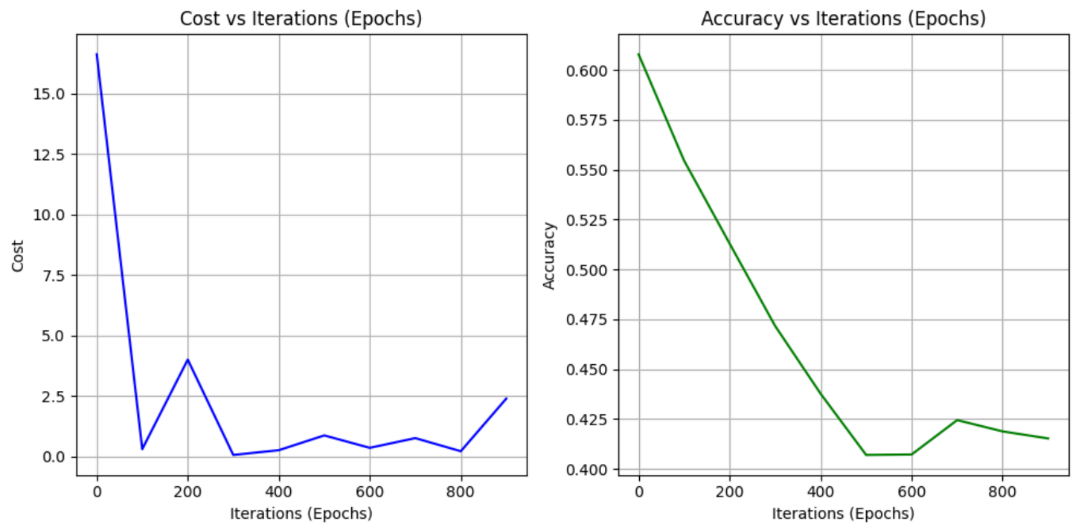
gradient descent where the model parameters (weights w and bias b) are updated using small random subsets of the data called "mini-batches."

The data is shuffled and split into mini-batches of a specified size. For each mini-batch, gradients of the cost function are computed and used to update the weights and bias. The cost and accuracy are calculated every 100 epochs to track model performance.

Stochastic gradient descendent
gradient descent, where model parameters are updated after every single data point, rather than using a full batch or mini-batch.

A single random sample is selected from the dataset for each iteration. The gradient is computed for that single sample, and the weights and bias are updated accordingly. Cost and accuracy are computed and logged every 100 epochs.

Stochastic gradient descendent



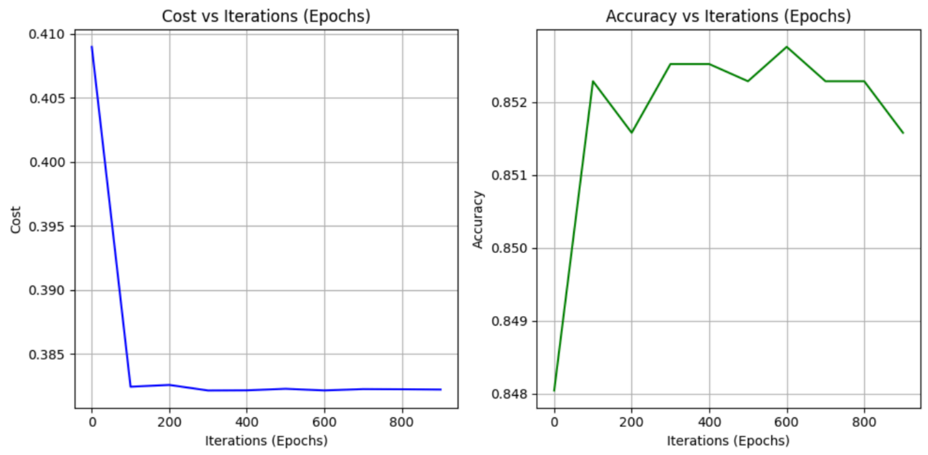
Cost vs Iterations
I observed that the cost started quite high, near **16**, and rapidly decreased in the initial iterations, dropping below **2**. However, after about 200 iterations, the cost started fluctuating, moving up and down without a consistent downward trend.
It appears that the cost didn't settle at a low point, implying that the learning may not have converged well using this method.

Accuracy vs Iterations

I observed a downward trend in accuracy as the iterations progressed. Initially, accuracy was around **0.6** but gradually decreased, approaching **0.4** by the 400th iteration. After that, there was a brief stabilization, but it didn't improve much, ending at **~0.42.**

In this, I observed that SGD, in this case, demonstrated instability both in cost and accuracy, which suggests it may have overfitted or the learning rate may be inappropriate for the model and dataset.

Mini-batch gradient decendent.

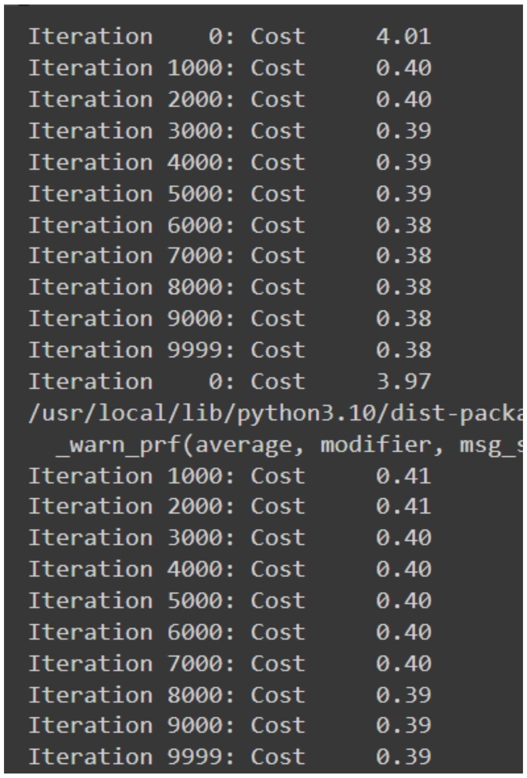


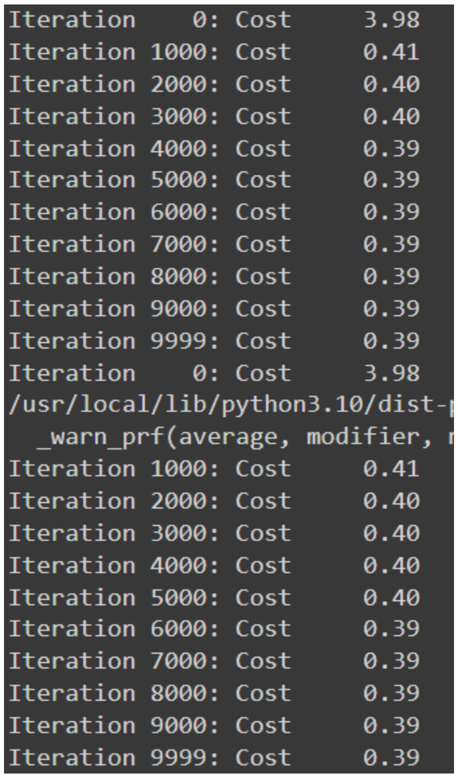
Cost vs Iterations:
I observed that the cost started at **~0.41** and dropped significantly in the first few iterations. Unlike stochastic gradient descent, the cost graph shows more stability, with a clear convergence to a minimum value without large fluctuations. This stability suggests that batch gradient descent provides more reliable updates by using the entire dataset to compute gradients.

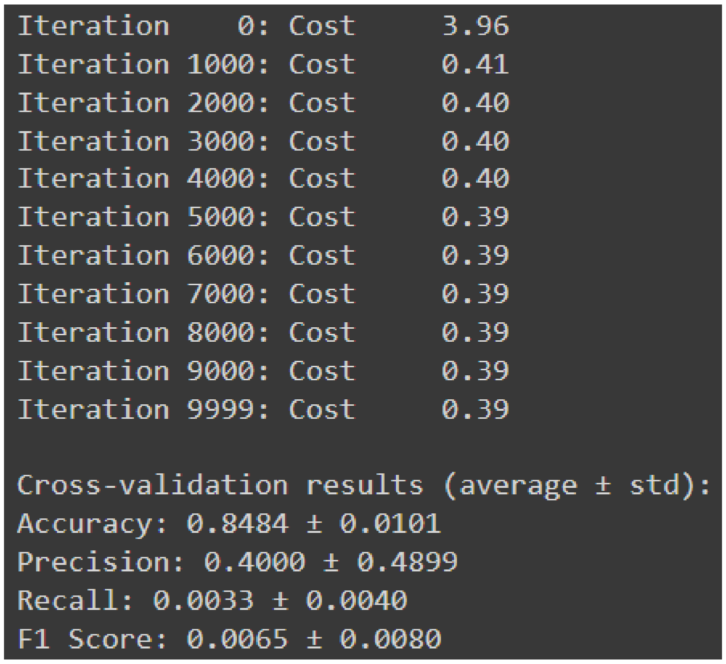
Accuracy vs Iterations
I noticed that the accuracy started at around **0.848** and quickly improved, reaching its peak around **0.852** after around 200 iterations. After reaching its peak, the accuracy fluctuated slightly but remained in a higher range compared to the SGD plot.

From this we can conclude that batch gradient descent produced much better results in terms of both cost and accuracy compared to stochastic gradient descent. It showed more stability, converging on a lower cost and achieving higher and more consistent accuracy.

Part (e)


```
Iteration    0: Cost     4.01
Iteration 1000: Cost     0.40
Iteration 2000: Cost     0.40
Iteration 3000: Cost     0.39
Iteration 4000: Cost     0.39
Iteration 5000: Cost     0.39
Iteration 6000: Cost     0.38
Iteration 7000: Cost     0.38
Iteration 8000: Cost     0.38
Iteration 9000: Cost     0.38
Iteration 9999: Cost     0.38
Iteration    0: Cost     3.97
/usr/local/lib/python3.10/dist-packa
  _warn_prf(average, modifier, msg_s
Iteration 1000: Cost     0.41
Iteration 2000: Cost     0.41
Iteration 3000: Cost     0.40
Iteration 4000: Cost     0.40
Iteration 5000: Cost     0.40
Iteration 6000: Cost     0.40
Iteration 7000: Cost     0.40
Iteration 8000: Cost     0.39
Iteration 9000: Cost     0.39
Iteration 9999: Cost     0.39
```


```
Iteration    0: Cost     3.98
Iteration 1000: Cost     0.41
Iteration 2000: Cost     0.40
Iteration 3000: Cost     0.40
Iteration 4000: Cost     0.39
Iteration 5000: Cost     0.39
Iteration 6000: Cost     0.39
Iteration 7000: Cost     0.39
Iteration 8000: Cost     0.39
Iteration 9000: Cost     0.39
Iteration 9999: Cost     0.39
Iteration    0: Cost     3.98
/usr/local/lib/python3.10/dist-p
  _warn_prf(average, modifier, r
Iteration 1000: Cost     0.41
Iteration 2000: Cost     0.40
Iteration 3000: Cost     0.40
Iteration 4000: Cost     0.40
Iteration 5000: Cost     0.40
Iteration 6000: Cost     0.39
Iteration 7000: Cost     0.39
Iteration 8000: Cost     0.39
Iteration 9000: Cost     0.39
Iteration 9999: Cost     0.39
```

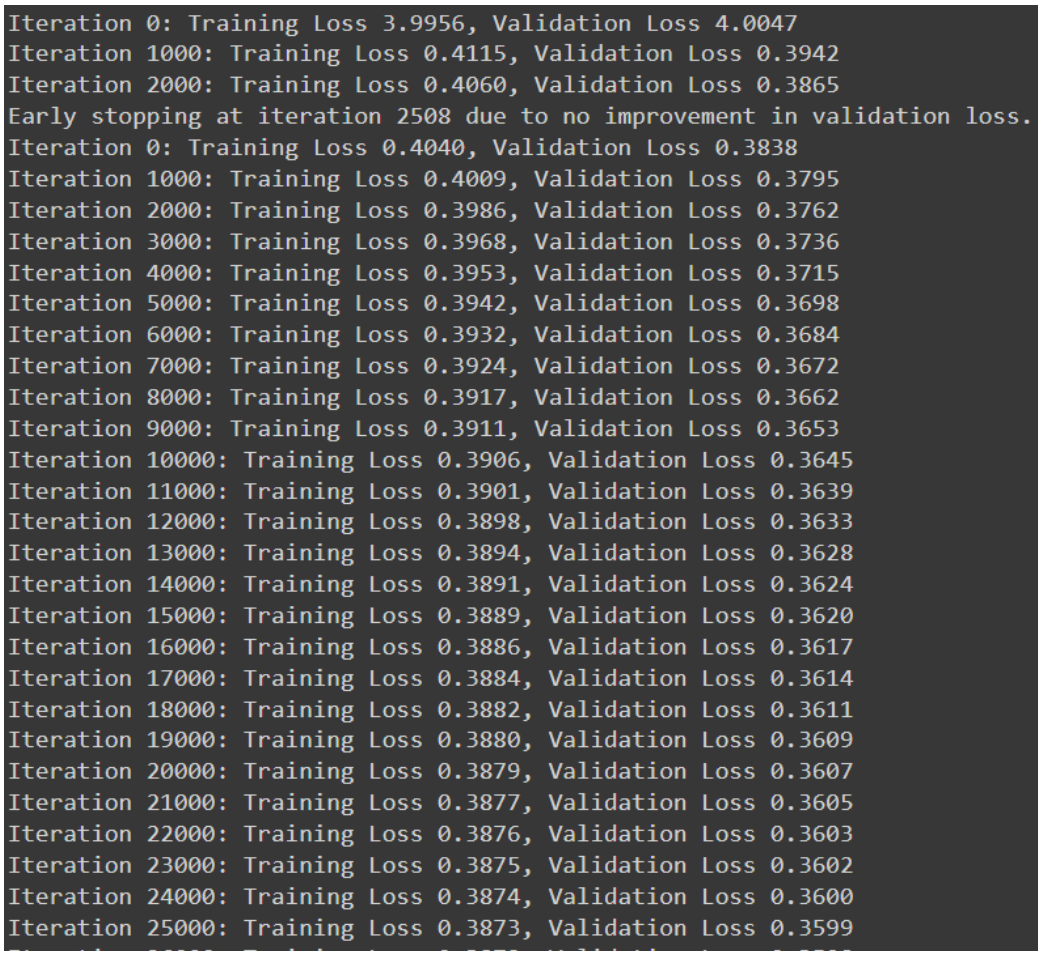

```
Iteration    0: Cost     3.96
Iteration 1000: Cost     0.41
Iteration 2000: Cost     0.40
Iteration 3000: Cost     0.40
Iteration 4000: Cost     0.40
Iteration 5000: Cost     0.39
Iteration 6000: Cost     0.39
Iteration 7000: Cost     0.39
Iteration 8000: Cost     0.39
Iteration 9000: Cost     0.39
Iteration 9999: Cost     0.39

Cross-validation results (average ± std):
Accuracy: 0.8484 ± 0.0101
Precision: 0.4000 ± 0.4899
Recall: 0.0033 ± 0.0040
F1 Score: 0.0065 ± 0.0080
```

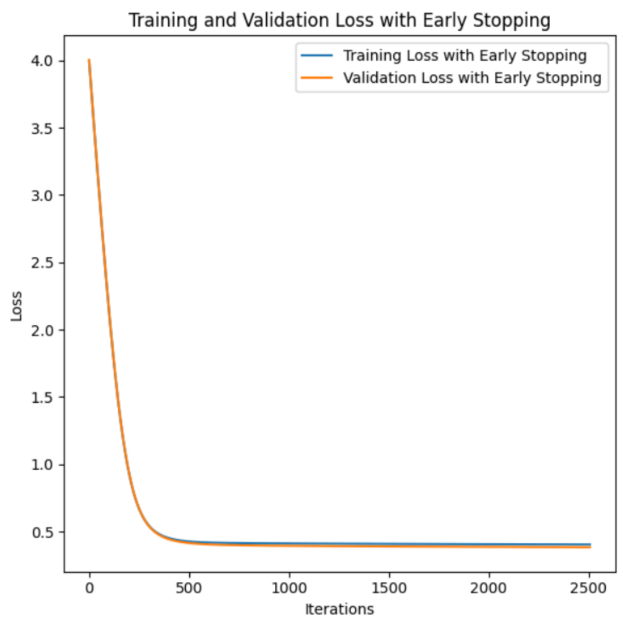
I conducted k-fold cross-validation using 5 folds. For instance, after 1000 iterations, the cost had reduced to approximately 0.40 in most cases. By the time I reached 9999 iterations, the cost

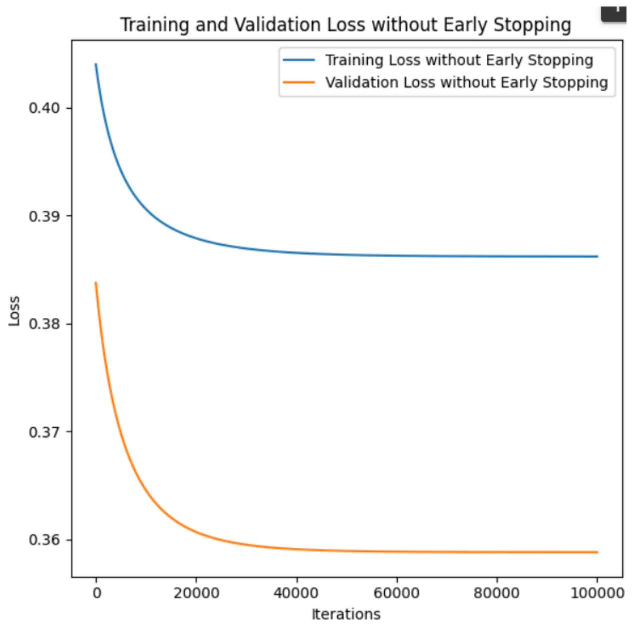
had stabilized at around 0.38, which shows that the model was effectively minimizing the error over time across all the folds.
At the end of the k-fold process, I calculated the average performance metrics across all 5 folds. The final accuracy was **0.8484**, which is quite stable, with a small standard deviation of **0.0101**. However, I observed significant issues with other metrics. Precision averaged **0.4000**, but the large standard deviation of **0.4899** indicates that the model's precision varied greatly across the folds. The recall was very low at **0.0033**, and the F1 score was similarly low at **0.0065**, indicating that the model was not effectively capturing positive instances, despite the reasonable accuracy.

Part (f)



```
Iteration 0: Training Loss 3.9956, Validation Loss 4.0047
Iteration 1000: Training Loss 0.4115, Validation Loss 0.3942
Iteration 2000: Training Loss 0.4060, Validation Loss 0.3865
Early stopping at iteration 2508 due to no improvement in validation loss.
Iteration 0: Training Loss 0.4040, Validation Loss 0.3838
Iteration 1000: Training Loss 0.4009, Validation Loss 0.3795
Iteration 2000: Training Loss 0.3986, Validation Loss 0.3762
Iteration 3000: Training Loss 0.3968, Validation Loss 0.3736
Iteration 4000: Training Loss 0.3953, Validation Loss 0.3715
Iteration 5000: Training Loss 0.3942, Validation Loss 0.3698
Iteration 6000: Training Loss 0.3932, Validation Loss 0.3684
Iteration 7000: Training Loss 0.3924, Validation Loss 0.3672
Iteration 8000: Training Loss 0.3917, Validation Loss 0.3662
Iteration 9000: Training Loss 0.3911, Validation Loss 0.3653
Iteration 10000: Training Loss 0.3906, Validation Loss 0.3645
Iteration 11000: Training Loss 0.3901, Validation Loss 0.3639
Iteration 12000: Training Loss 0.3898, Validation Loss 0.3633
Iteration 13000: Training Loss 0.3894, Validation Loss 0.3628
Iteration 14000: Training Loss 0.3891, Validation Loss 0.3624
Iteration 15000: Training Loss 0.3889, Validation Loss 0.3620
Iteration 16000: Training Loss 0.3886, Validation Loss 0.3617
Iteration 17000: Training Loss 0.3884, Validation Loss 0.3614
Iteration 18000: Training Loss 0.3882, Validation Loss 0.3611
Iteration 19000: Training Loss 0.3880, Validation Loss 0.3609
Iteration 20000: Training Loss 0.3879, Validation Loss 0.3607
Iteration 21000: Training Loss 0.3877, Validation Loss 0.3605
Iteration 22000: Training Loss 0.3876, Validation Loss 0.3603
Iteration 23000: Training Loss 0.3875, Validation Loss 0.3602
Iteration 24000: Training Loss 0.3874, Validation Loss 0.3600
Iteration 25000: Training Loss 0.3873, Validation Loss 0.3599
```





1) Training and Validation Loss with Early Stopping
   Initially, both training and validation loss started high but rapidly decreased as the model learned from the data. The training and validation loss reached a minimum after around **500 iterations.** As expected, early stopping resulted in both training and validation loss curves converging smoothly, indicating no significant overfitting.

2) Training and Validation Loss without Early Stopping
   The training loss steadily decreased over time, eventually approaching **0.35** after around **100,000 iterations**. And the validation curve follows a different path. decreased along with the training loss, it soon reached a minimum of around **0.36** and started diverging. This divergence in validation loss shows signs of **overfitting**, where the model continues to learn the noise in the training data and no longer generalizes well to unseen data.

From this we can conclude that in early stopping, both the training and validation loss values converged, indicating good generalization performance. While in without early stopping, Training without early stopping leads to a model that fits the training data too well, compromising its ability to generalize and leads to overfitting.