BITS F464 MACHINE LEARNING
ASSIGNMENT 1

GROUP MEMBERS :

M. ASHRITHA 2019B3A70472H
SUSHMA REDDY KOLLI 2019B5A70671H

Introduction:

In this report, we will be discussing the performance of various classification models - Perceptron Learning Algorithm, Fisher's Linear Discriminant Analysis, and Logistic Regression, on a given dataset. We will compare the performance of these models and identify the best performing model based on average performance metrics of 10 random training and testing splits.

The criteria used to grade this assignment are accuracy, recall, and precision, which are all basic measures of performance.

TP: true positives (number of correctly classified positive instances)
TN: true negatives (number of correctly classified negative instances)
FP: false positives (number of negative instances classified as positive)
FN: false negatives (number of positive instances classified as negative)

Accuracy: (TP + TN) / (TP + TN + FP + FN)
Accuracy measures the proportion of correctly classified instances (both positive and negative) among all instances in the dataset.

Precision: TP / (TP + FP)
Precision measures the proportion of correctly classified positive instances among all instances classified as positive.

Recall: TP / (TP + FN)
Recall measures the proportion of correctly classified positive instances among all actual positive instances.

Part A - Perceptron Learning Algorithm:

Perceptron Learning Algorithm is a binary classification algorithm used to classify input data into two categories. We have been given a dataset, and we need to build a classifier using the Perceptron Model. We need to determine whether the dataset is linearly separable by building the model.

Here are the steps of the perceptron algorithm

- Initialize the weight vector and bias term to zero.
- For each training example:
    - Compute the predicted output label as the sign of the dot product between the weight vector and the input features plus the bias term.
    - If the predicted output label is different from the true output label, update the weight vector and bias term as follows:
        - Add the product of the true output label and input features to the weight vector.
        - Add the true output label to the bias term.
- Repeat the previous step for a fixed number of iterations or until the algorithm converges (i.e., no more errors are made on the training examples).
- Use the learned weight vector and bias term to predict the output label for new examples.

Libraries Used: numpy, pandas, math

PM1

Upon running the perception algorithm on the provided dataset, the obtained results indicate an accuracy of 0.37, a recall score of 1, and a precision score of 0.372 as depicted in below picture. It is noteworthy that these outcomes were obtained without performing any shuffling or normalization of the data.

```
Mounted at /content/drive
<ipython-input-2-d5f2462c0b53>:39: Futur
  X = np.array(data.drop(["diagnosis"],1
Accuracy: 0.372583479789103 67
Precision: 0.372583479789103 67
Recall: 1.0
```

we need to change the order of training examples and build another classifier called **PM2**. We will compare the differences between the models PM1 and PM2.

Upon running the perception algorithm on the provided dataset, the obtained results indicate an accuracy of 0.92, a recall score of 0.924, and a precision score of 0.886 as depicted in below picture.
The aforementioned results were obtained after shuffling the dataset. Upon comparing these results with those of the PM1 model, it is evident that accuracy and precision have both increased, while recall has decreased. Notably, shuffling the data has the effect of reducing variance and enhancing the generalization ability of the models, thereby mitigating the risk of overfitting.

```
<ipython-input-6-47a77b3055f8>:3: Futur
  X = np.array(data.drop(["diagnosis"],
Accuracy: 0.92794376098418 27
Precision: 0.8868778280542986
Recall: 0.9245283018867925
```

we build another classifier called **PM3** using the perceptron algorithm on normalized data. We will compare the differences between PM1 and PM3.

Upon running the perception algorithm on the provided dataset, the obtained results indicate an accuracy of 0.98, a recall score of 0.98, and a precision score of 0.99 as depicted in below picture.
The aforementioned results were obtained after normalizing the dataset. Upon comparing these results with those of the PM1 model, it is evident that accuracy and precision have both increased, while recall has decreased.Normalization ensures that each variable is given equal weight and importance in the model, thereby preventing any individual variable from dominating the model's performance simply because of its larger numerical values.

```
<ipython-input-4-bc0bfe82e15c>:4: Futur
  X = np.array(data.drop(["diagnosis"],
Accuracy: 0.9894551845342706
Precision: 0.9904761904761905
Recall: 0.9811320754716981
```

we need to randomly change the order of features in the dataset and build another classifier called **PM4**. We will compare the differences between PM1 and PM4 and their respective performances.

Upon running the perception algorithm on the provided dataset, the obtained results indicate an accuracy of 0.88, a recall score of 0.95, and a precision score of 0.77 as depicted in below picture.
The aforementioned results were obtained after shuffling the features of the dataset. Upon comparing these results with those of the PM1 model, it is evident that accuracy and precision have both increased, while recall has decreased.

```
Drive already mounted at /content/drive;
Accuracy: 0.8822495606326889
Precision: 0.7799227799227799
Recall: 0.9528301886792453
```

Based on the output of the models PM1, PM2, PM3, and PM4, it appears that the data is not linearly separable. This is indicated by the fact that the converged variable for each of these models is false, which means that the optimization algorithm did not converge to a solution that fully separates the two classes of data as depicted in below picture.

```
# Check if data is linearly separable
if model.converged:
    print('Data is linearly separable')
else:
    print('Data is not linearly separable')
```

```
Data is not linearly separable
```

Part B - Fishers Linear Discriminant:

Fisher's Linear Discriminant Analysis (LDA) is a statistical technique used for binary classification problems. It is used to find a linear combination of features that best separates two classes in a dataset. Fisher's LDA is based on the assumption that the class-conditional densities are multivariate normal with the same covariance matrix for both classes.

Algorithm:

1. First we split the data into 2 classes (positive and negative) according to the target variable.
2. We calculate means of both classes

$$M_0 = \frac{1}{N_0} \sum_{n \in C_0} (x_n) \quad \text{and } M_1 = \frac{1}{N_1} \sum_{n \in C_1} (x_n)$$

x_n, negative and positive respectively.

3. We calculate the total within class covariance matrix S_w which can be obtained as the sum of the covariance matrices of both classes, i.e. $S_w = S_0 + S_1$
4. The unit vector onto which the points are projected can be found out by

$$\overline{w} = S_w^{-1}(M_1 - M_0) \text{ and } \widehat{w} = \frac{\overline{w}}{|\overline{w}|}$$

5. Now we project the data points in the direction of the unit vector. Positive points have red colors and Negative points have blue color as shown in the below figures.
6. With the help of the reduced 1D features as inputs, we plot the Gaussian distributions for both the classes and find the intersection point (α) by solving the following quadratic equation -

$$\left(\frac{1}{\sigma_1^2} - \frac{1}{\sigma_0^2}\right)x^2 + 2\left(\frac{\mu_1}{\sigma_1^2} - \frac{\mu_1}{\sigma_1^2}\right)x + \frac{\mu 2_0}{\sigma_0^2} + \frac{\mu_1}{\sigma_1^2} + log(\frac{\mu_0}{\mu_1})$$

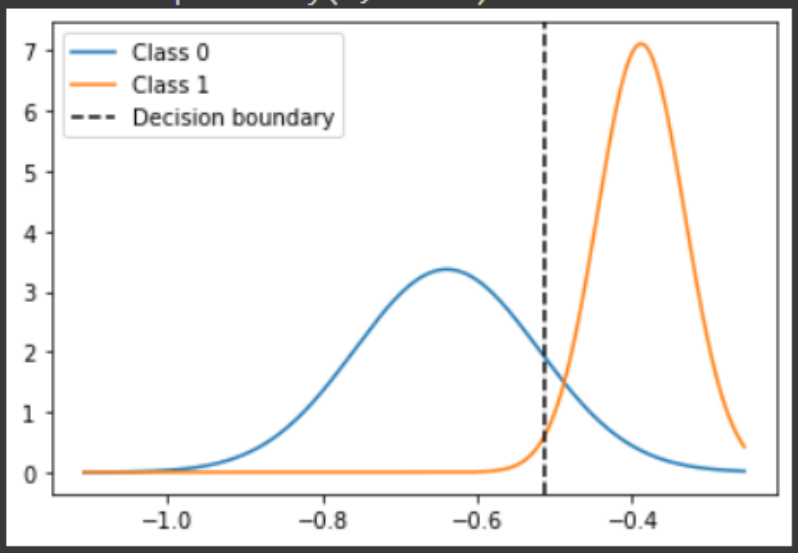
Where μ_0 *and* μ_1 are means of negative and positive classes and σ_1 *and* σ_2 are variances of negative and positve classes respectively.

7. The point α is the discriminating point (Threshold) in 1D space. The equation for decision boundary is found in the original feature space using $\widehat{w}$ T · x = α. Therefore if $\widehat{w}$ T · ~x > α then ~x will be classified as positive point, else it will be a negative point.

FLDM1:

When the given dataset was trained using Fisher's Linear Discriminant Analysis (FLDA), the resulting accuracy was 0.6408, which means that the model correctly classified 64% of the test examples. However, looking at the precision and recall scores, we can see that the model's performance is not great. The precision score of 0.91 means that out of all the examples predicted as positive, only 91% of them are actually positive, while the rest are false positives. On the other hand, the recall score of 0.0 means that the model did not correctly identify any of the positive examples in the test set, meaning that it has a high number of false negatives. This suggests that the model is biased towards the negative class, and may need to be further tuned or improved to better handle the positive class. Thresholdin univariate dimension is (-0.51403680336085+0j)

```
Drive already mounted at /content/drive; to attempt to forci
/usr/local/lib/python3.9/dist-packages/matplotlib/cbook/__ir
  return np.asarray(x, float)
```




```
Accuracy: 0.6408450704225352
Precision: 0.91
Recall: 0.0
Threshold (-0.51403680336085+0j)
```

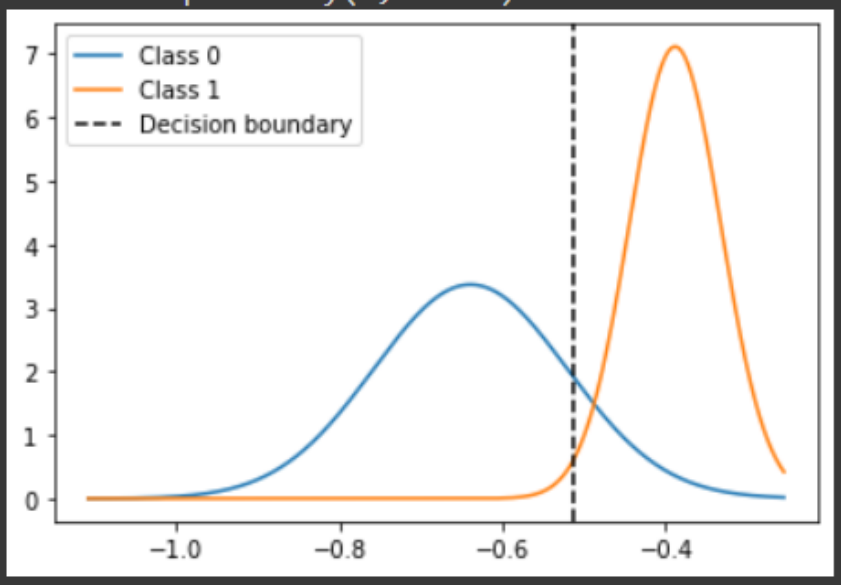
FLDM2:

When we shuffled the features of Fisher's Linear Discriminant Analysis (FLDA) model, FLDM1, to create FLDM2, we did not see any significant changes in the model's performance metrics. This indicates that the model is robust to changes in the order of the features. The fact that the performance metrics did not change suggests that the model is not relying on any specific feature or set of features to make its predictions, but is instead able to extract the relevant information from the dataset regardless of the order of the

features. This can be seen as a positive characteristic of the model, as it indicates that it is not overfitting to the training data, and can generalize well to new data. However, it is important to note that this may not hold true for all datasets or models, and further analysis and experimentation may be necessary to ensure the robustness and generalizability of the model.

Thresholdin univariate dimension is (-0.51403680336085+0j)

```
Drive already mounted at /content/drive; to attempt to forci
/usr/local/lib/python3.9/dist-packages/matplotlib/cbook/__ir
  return np.asarray(x, float)
```




```
Accuracy: 0.6408450704225352
Precision: 0.91
Recall: 0.0
Threshold (-0.51403680336085+0j)
```

The decision probability threshold is the threshold value that separates the positive and negative class predictions made by a binary classification model. When we change the decision probability threshold, we are essentially changing the classification rule of the model, which may result in a change in the number of true positives, true negatives, false positives, and false negatives.

However, in some cases, the accuracy of a model may remain the same even when we change the decision probability threshold. This can happen when the model's predicted probabilities are already well separated and do not overlap significantly between the positive and negative classes. In these situations, changing the decision probability

threshold might not have a big effect on how well the model classifies, and the accuracy might stay the same.

Part C - Logistic Regression:

Logistic regression is a statistical model used for binary classification problems where the target variable takes only two possible values, usually encoded as 0 and 1. Based on the values of the input features, the model figures out how likely it is that the target variable will belong to a certain class. Logistic regression is a linear model, but it uses the logistic function, also known as the sigmoid function, to map the output to the probability range [0, 1]. The model is trained by minimizing a cost function using gradient descent, where the goal is to find the values of the model parameters that best fit the data.

We can use three gradient descent methods:

1. Batch Gradient Descent
2. Mini-Batch Gradient Descent
3. Stochastic Gradient Descent

Batch Gradient Descent involves calculating the gradients of the loss function with respect to the model parameters for the entire training dataset. This means that all the training examples are considered in each iteration, and the parameters are updated based on the average gradient of the entire dataset. Batch Gradient Descent can be slow for large datasets, but it can lead to more stable convergence and better solutions.

Learning Task 1:

```
# train the logistic regression model
lr = LogisticRegression(learning_rate=0.1, num_iterations=1000)
lr.fit(X_train, y_train)
```

For Decision Probability Threshold = 0.5 and learning rate = 0.1

```
Accuracy: 0.6052631578947368
Precision: 0.0
Recall: 0.0
```

For Decision Probability Threshold = 0.3 and learning rate = 0.1

```
Accuracy: 0.6052631578947368
Precision: 0.0
Recall: 0.0
```

For Decision Probability Threshold = 0.4 and learning rate = 0.1

```
Accuracy: 0.6052631578947368
Precision: 0.0
Recall: 0.0
```

For Decision Probability Threshold = 0.6 and learning rate = 0.1

```
Accuracy: 0.6052631578947368
Precision: 0.0
Recall: 0.0
```

The accuracy doesn't change with the decision probability threshold due to over-fitting.

Learning Task 2:

For **Engineering Task 1:**

For Decision Probability Threshold = 0.3 and learning rate = 0.1

```
accuracy, precision, recall = lr.score(X_test, y_test, threshold=0.3)
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
```

```
Testing accuracy: 0.94
Testing precision: 0.84
Recall: 1.00
```

For Decision Probability Threshold = 0.4 and learning rate = 0.1

```
# evaluate the model on the testing set with a decision threshold of 0.5
accuracy, precision, recall = lr.score(X_test, y_test, threshold=0.4)
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
```

```
Testing accuracy: 0.95
Testing precision: 0.86
Recall: 1.00
```

For Decision Probability Threshold = 0.5 and learning rate = 0.1

```
# evaluate the model on the testing set with a decision threshold of 0.5
accuracy, precision, recall = lr.score(X_test, y_test, threshold=0.5)
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
```

```
Testing accuracy: 0.95
Testing precision: 0.88
Recall: 0.97
```

For Decision Probability Threshold = 0.6 and learning rate = 0.1

```
# evaluate the model on the testing set with a decision threshold of 0.6
accuracy, precision, recall = lr.score(X_test, y_test, threshold=0.6)
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Recall: {recall:.2f}')
```

```
Testing accuracy: 0.95
Testing precision: 0.90
Recall: 0.95
```

For **Engineering Task 2**:
(NORMALISATION):

```
# normalize the data
X = df.drop(['diagnosis'], axis=1).values
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)
```

For Decision Probability Threshold = 0.5 and learning rate = 0.1

```
Testing accuracy: 0.95
Testing precision: 0.88
Testing recall: 0.97
```

For Decision Probability Threshold = 0.4 and learning rate = 0.1

```
Testing accuracy: 0.95
Testing precision: 0.86
Testing recall: 1.00
```

For Decision Probability Threshold = 0.3 and learning rate = 0.1

```
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Testing recall: {recall:.2f}')
```

```
Testing accuracy: 0.94
Testing precision: 0.84
Testing recall: 1.00
```

For Decision Probability Threshold = 0.6 and learning rate = 0.1

```
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Testing recall: {recall:.2f}')
```

```
Testing accuracy: 0.95
Testing precision: 0.90
Testing recall: 0.95
```

Changing Learning Rates :

For Decision Probability Threshold = 0.5 and **learning rate = 0.001**

```
# train the logistic regression model
lr = LogisticRegression(learning_rate=0.001, num_iterations=1000)
lr.fit(X_train, y_train)
```

```
print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Testing recall: {recall:.2f}')
```

```
Testing accuracy: 0.96
Testing precision: 0.92
Testing recall: 0.95
```

For Decision Probability Threshold = 0.5 and **learning rate = 0.0001**

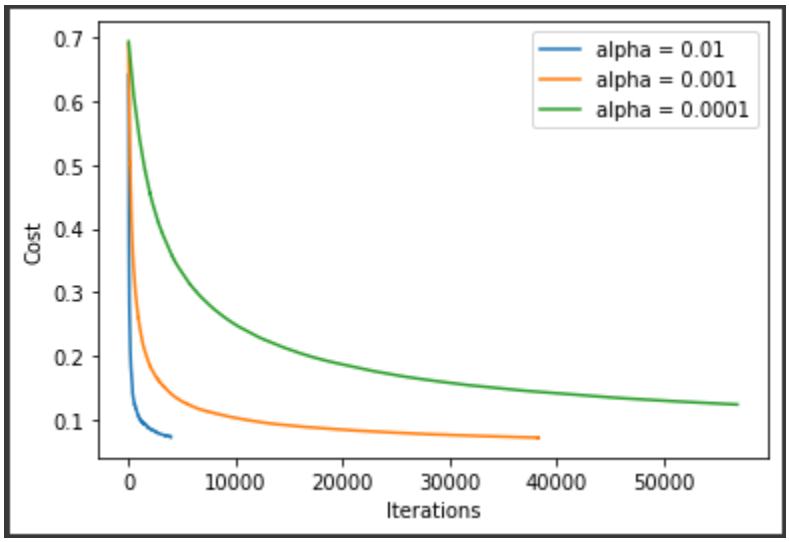
```
# train the logistic regression model
lr = LogisticRegression(learning_rate=0.0001, num_iterations=1000)
lr.fit(X_train, y_train)

# evaluate the model on the testing set
accuracy = lr.accuracy(X_test, y_test)
precision = lr.precision(X_test, y_test)
recall = lr.recall(X_test, y_test)

print(f'Testing accuracy: {accuracy:.2f}')
print(f'Testing precision: {precision:.2f}')
print(f'Testing recall: {recall:.2f}')
```

```
Testing accuracy: 0.91
Testing precision: 0.89
Testing recall: 0.84
```

Plotting the Cost vs Iterations for a Batch Gradient:

Plotting the graph for different learning rate = 0.01,0.001,0.0001

```
learning_rates = [0.01, 0.001, 0.0001]
for lr in learning_rates:
    model = LogisticRegression(learning_rate=lr, num_iterations=1000)
    models.append(model)
```



Mini-Batch Gradient Descent:

Mini-Batch Gradient Descent is a compromise between Batch Gradient Descent and Stochastic Gradient Descent. It involves updating the parameters based on the gradients calculated on a small random subset of the training examples, called a mini-batch. This approach combines the advantages of both Batch Gradient Descent and Stochastic Gradient Descent, as it can be more efficient than Batch Gradient Descent and more stable than Stochastic Gradient Descent.

LR1:

For decision boundary = 0.3 alpha = learning rate = 0.01 :

```
Accuracy for alpha = 0.01: 0.373
Precision for alpha = 0.01: 0.373
Recall for alpha = 0.01: 1.000
Testing accuracy: 0.37
Precision: 0.37
Recall: 1.00
Learned Parameters:
[ 2.82126642e+04  1.70546676e+00 -6.06835312e+00  1.75837973e+01
  7.12051500e+02 -4.86105703e-02  7.42380578e-02  1.99942348e-01
  1.08081265e-01 -9.25607031e-02 -4.77161656e-02  4.45010078e-01
 -9.50220219e-01  3.20738356e+00  8.87726103e+01 -6.33340406e-03
  5.93141462e-03  1.25657934e-02  2.94213313e-03 -1.53639456e-02
 -1.94827872e-03  5.87489938e+00 -5.89316094e+00  4.63193719e+01
  1.35515626e+03 -5.33008672e-02  2.59070744e-01  4.57952354e-01
  1.65287940e-01 -9.07940469e-02 -3.49713094e-02]
```

For decision boundary = 0.4 alpha = learning rate = 0.01 :

```
Accuracy for alpha = 0.01: 0.373
Precision for alpha = 0.01: 0.373
Recall for alpha = 0.01: 1.000
Testing accuracy: 0.37
Precision: 0.37
Recall: 1.00
Learned Parameters:
[ 2.82126642e+04  1.70546676e+00 -6.06835312e+00  1.75837973e+01
  7.12051500e+02 -4.86105703e-02  7.42380578e-02  1.99942348e-01
  1.08081265e-01 -9.25607031e-02 -4.77161656e-02  4.45010078e-01
 -9.50220219e-01  3.20738356e+00  8.87726103e+01 -6.33340406e-03
  5.93141462e-03  1.25657934e-02  2.94213313e-03 -1.53639456e-02
 -1.94827872e-03  5.87489938e+00 -5.89316094e+00  4.63193719e+01
  1.35515626e+03 -5.33008672e-02  2.59070744e-01  4.57952354e-01
  1.65287940e-01 -9.07940469e-02 -3.49713094e-02]
```

For decision boundary = 0.5 alpha = learning rate = 0.01 :

```
Accuracy for alpha = 0.01: 0.373
Precision for alpha = 0.01: 0.373
Recall for alpha = 0.01: 1.000
Testing accuracy: 0.37
Precision: 0.37
Recall: 1.00
Learned Parameters:
[ 2.82126642e+04  1.70546676e+00 -6.06835312e+00  1.75837973e+01
  7.12051500e+02 -4.86105703e-02  7.42380578e-02  1.99942348e-01
  1.08081265e-01 -9.25607031e-02 -4.77161656e-02  4.45010078e-01
 -9.50220219e-01  3.20738356e+00  8.87726103e+01 -6.33340406e-03
  5.93141462e-03  1.25657934e-02  2.94213313e-03 -1.53639456e-02
 -1.94827872e-03  5.87489938e+00 -5.89316094e+00  4.63193719e+01
  1.35515626e+03 -5.33008672e-02  2.59070744e-01  4.57952354e-01
  1.65287940e-01 -9.07940469e-02 -3.49713094e-02]
```

For decision boundary = 0.6 alpha = learning rate = 0.01 :

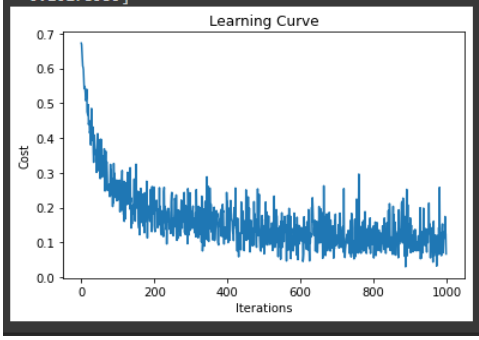
```
Accuracy for alpha = 0.01: 0.373
Precision for alpha = 0.01: 0.373
Recall for alpha = 0.01: 1.000
Testing accuracy: 0.37
Precision: 0.37
Recall: 1.00
Learned Parameters:
[ 2.82126642e+04  1.70546676e+00 -6.06835312e+00  1.75837973e+01
  7.12051500e+02 -4.86105703e-02  7.42380578e-02  1.99942348e-01
  1.08081265e-01 -9.25607031e-02 -4.77161656e-02  4.45010078e-01
 -9.50220219e-01  3.20738356e+00  8.87726103e+01 -6.33340406e-03
  5.93141462e-03  1.25657934e-02  2.94213313e-03 -1.53639456e-02
 -1.94827872e-03  5.87489938e+00 -5.89316094e+00  4.63193719e+01
  1.35515626e+03 -5.33008672e-02  2.59070744e-01  4.57952354e-01
  1.65287940e-01 -9.07940469e-02 -3.49713094e-02]
```

LR2:

For alpha = learning rate = 0.01 :

```
# Set hyperparameters
alpha = 0.01
epochs = 1000
batch_size = 32
```

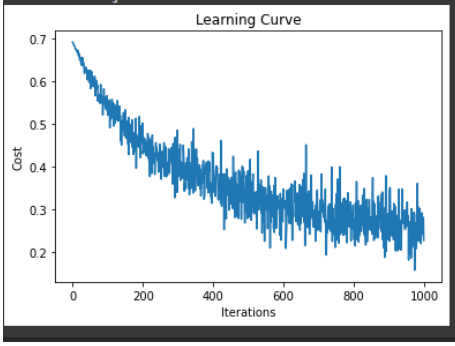
```
Accuracy for alpha = 0.01: 0.974
Precision for alpha = 0.01: 0.958
Recall for alpha = 0.01: 0.972
Testing accuracy: 0.97
Precision: 0.96
Recall: 0.97
Learned Parameters:
[ 0.02824512  0.38931309  0.32300871  0.38819789  0.39628608  0.14163273
  0.14046954  0.30713125  0.40187072  0.10738274 -0.15639258  0.36383117
  0.00743901  0.31268997  0.33728286  0.00674392 -0.09586061 -0.07676671
  0.04928771 -0.05055553 -0.17755283  0.46681794  0.39769428  0.44886644
  0.44930541  0.30343748  0.20468747  0.28664298  0.40759067  0.28150204
  0.10178959]
```



For alpha = learning rate = 0.001

```
# Set hyperparameters
alpha = 0.001
epochs = 1000
batch_size = 32
```

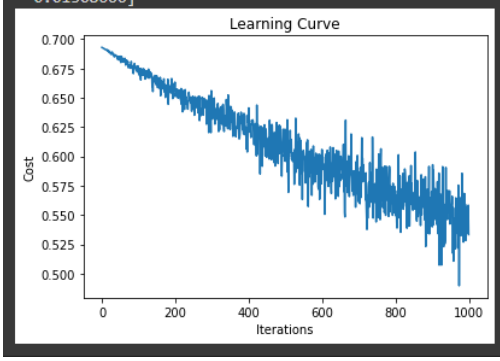
```
Accuracy for alpha = 0.001: 0.954
Precision for alpha = 0.001: 0.931
Recall for alpha = 0.001: 0.948
Testing accuracy: 0.95
Precision: 0.93
Recall: 0.95
Learned Parameters:
[ 0.00606049  0.1571437   0.0994903   0.15863081  0.15180747  0.06376731
  0.10199236  0.13213335  0.1589289   0.0562953  -0.03098259  0.11564876
 -0.00673602  0.10873911  0.1105416  -0.01987117  0.02529793  0.01943481
  0.0592823  -0.01281617 -0.0199825   0.17118793  0.11460442  0.16965018
  0.15902248  0.09387164  0.11147888  0.12851795  0.16587516  0.09559368
  0.05449537]
```



For alpha = learning rate = 0.0001

```
# Set hyperparameters
alpha = 0.0001
epochs = 1000
batch_size = 32
```

```
Accuracy for alpha = 0.0001: 0.935
Precision for alpha = 0.0001: 0.907
Recall for alpha = 0.0001: 0.920
Testing accuracy: 0.93
Precision: 0.91
Recall: 0.92
Learned Parameters:
[ 0.00149927  0.03075841  0.01763438  0.0313031   0.0298165   0.01429217
  0.02422397  0.02869004  0.03237436  0.01315316 -0.00167181  0.02317953
 -0.00092121  0.02257758  0.02233514 -0.003402    0.0111179   0.0098668
  0.01636562 -0.00079154  0.00204313  0.03290123  0.01968329  0.0330673
  0.03085375  0.0176501   0.0244424   0.02751306  0.03345928  0.01791512
  0.01308666]
```

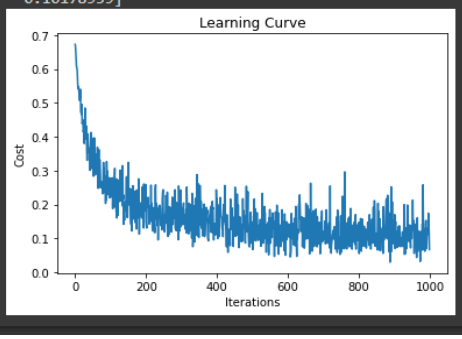


For **Decision Probability Threshold = 0.3 and learning rate = 0.1**

```
# compute accuracy, precision, and recall
y_pred = np.round(sigmoid(X.dot(theta)) + 0.3 - 0.5)
```

```
# Set hyperparameters
alpha = 0.01
epochs = 1000
batch_size = 32
```

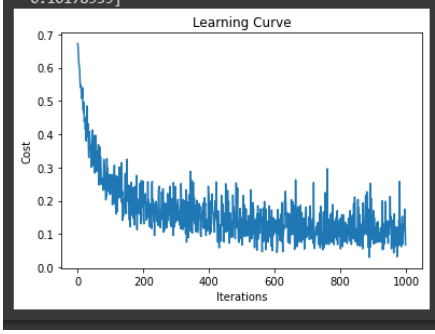
```
Accuracy for alpha = 0.01: 0.967
Precision for alpha = 0.01: 0.995
Recall for alpha = 0.01: 0.915
Testing accuracy: 0.97
Precision: 0.99
Recall: 0.92
Learned Parameters:
[ 0.02824512  0.38931309  0.32300871  0.38819789  0.39628608  0.14163273
  0.14046954  0.30713125  0.40187072  0.10738274 -0.15639258  0.36383117
  0.00743901  0.31268997  0.33728286  0.00674392 -0.09586061 -0.07676671
  0.04928771 -0.05055553 -0.17755283  0.46681794  0.39769428  0.44886644
  0.44930541  0.30343748  0.20468747  0.28664298  0.40759067  0.28150204
  0.10178959]
```



For **Decision Probability Threshold = 0.4 and learning rate = 0.1**

```
# compute accuracy, precision, and recall
y_pred = np.round(sigmoid(X.dot(theta)) + 0.4 - 0.5)
accuracy = np.mean(y_pred == y)
```

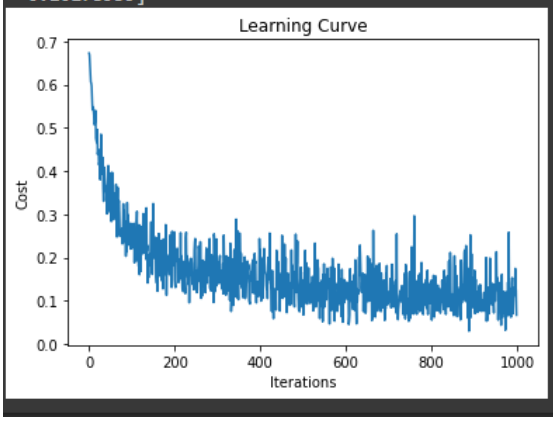
```
Accuracy for alpha = 0.01: 0.977
Precision for alpha = 0.01: 0.990
Recall for alpha = 0.01: 0.948
Testing accuracy: 0.98
Precision: 0.99
Recall: 0.95
Learned Parameters:
[ 0.02824512  0.38931309  0.32300871  0.38819789  0.39628608  0.14163273
  0.14046954  0.30713125  0.40187072  0.10738274 -0.15639258  0.36383117
  0.00743901  0.31268997  0.33728286  0.00674392 -0.09586061 -0.07676671
  0.04928771 -0.05055553 -0.17755283  0.46681794  0.39769428  0.44886644
  0.44930541  0.30343748  0.20468747  0.28664298  0.40759067  0.28150204
  0.10178959]
```



For **Decision Probability Threshold = 0.4 and learning rate = 0.1**

```
# compute accuracy, precision, and recall
y_pred = np.round(sigmoid(X.dot(theta)) + 0.6 - 0.5)
accuracy = np.mean(y_pred == y)
```

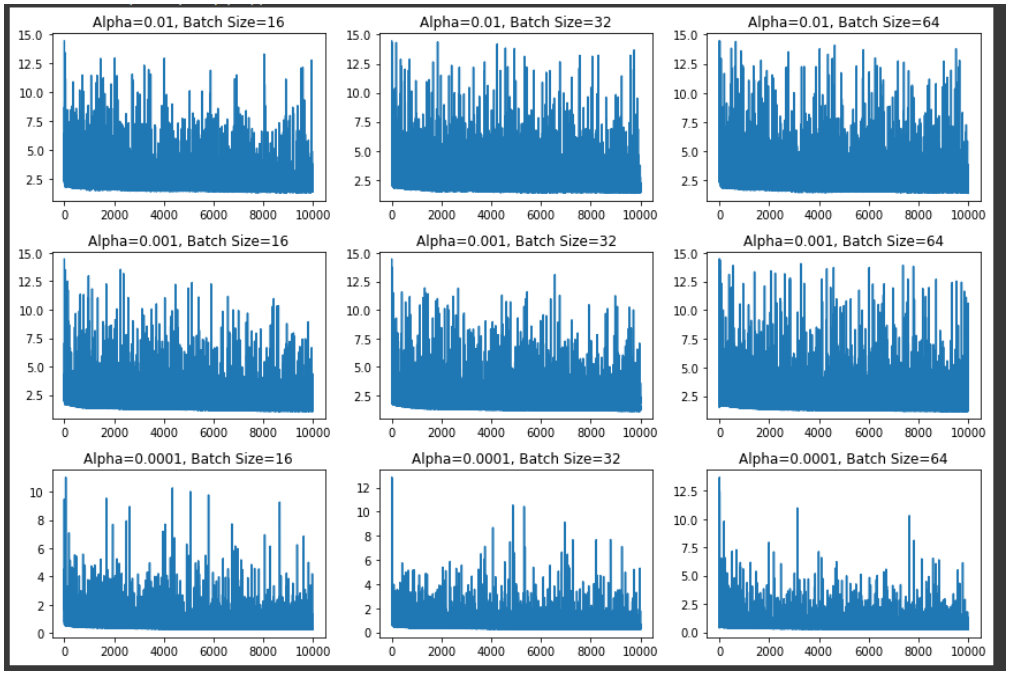
```
Accuracy for alpha = 0.01: 0.967
Precision for alpha = 0.01: 0.937
Recall for alpha = 0.01: 0.976
Testing accuracy: 0.97
Precision: 0.94
Recall: 0.98
Learned Parameters:
[ 0.02824512  0.38931309  0.32300871  0.38819789  0.39628608  0.14163273
  0.14046954  0.30713125  0.40187072  0.10738274 -0.15639258  0.36383117
  0.00743901  0.31268997  0.33728286  0.00674392 -0.09586061 -0.07676671
  0.04928771 -0.05055553 -0.17755283  0.46681794  0.39769428  0.44886644
  0.44930541  0.30343748  0.20468747  0.28664298  0.40759067  0.28150204
  0.10178959]
```



Plotting the Cost vs Iterations for a Batch Gradient:

Plotting the graph for different learning rate = 0.01,0.001,0.0001

And batch sizes : 16,32,64



On the other hand, **Stochastic Gradient Descent** involves updating the parameters for each training example one at a time. This means that the gradient of the loss function is

calculated for each training example individually, and the parameters are updated based on this gradient. Stochastic Gradient Descent can converge faster than Batch Gradient Descent, but the updates can be noisy, and the solution can be less stable.

LR1:

For **Decision Probability Threshold = 0.3 and learning rate = 0.01**

```
Training accuracy: 0.37258347978910367
Training precision: 1.0
Training recall: 1.0
```

For **Decision Probability Threshold = 0.4 and learning rate = 0.01**

```
Training accuracy: 0.37258347978910367
Training precision: 1.0
Training recall: 1.0
```

For **Decision Probability Threshold = 0.5 and learning rate = 0.01**

```
Training accuracy: 0.37258347978910367
Training precision: 1.0
Training recall: 1.0
```

For **Decision Probability Threshold = 0.5 and learning rate = 0.01**

```
Training accuracy: 0.37258347978910367
Training precision: 1.0
Training recall: 1.0
```

LR2 :

Engineering Task 2:

For **Decision Probability Threshold = 0.3 and learning rate = 0.01**

```
theta, costs = stochastic_gradient_descent(X, y, 0.01, 100)
```

```
y_pred = (sigmoid(X.dot(theta)) >= 0.3).astype(int)
accuracy = (y_pred == y).mean()
```

```
Training accuracy: 0.9771528998242531
Training precision: 0.9811320754716981
Training recall: 0.9811320754716981
```

For **Decision Probability Threshold = 0.6 and learning rate = 0.01**

```
#y_pred = np.round(sigmoid(X.dot(theta)))
y_pred = (sigmoid(X.dot(theta)) >= 0.6).astype(int)
accuracy = (y_pred == y).mean()
```

```
Training accuracy: 0.9859402460456942
Training precision: 0.9622641509433962
Training recall: 0.9622641509433962
```

For **Decision Probability Threshold = 0.4 and learning rate = 0.01**

```
# Compute the accuracy, precision, and recall on the training set
#y_pred = np.round(sigmoid(X.dot(theta)))
y_pred = (sigmoid(X.dot(theta)) >= 0.4).astype(int)
```

```
Training accuracy: 0.9859402460456942
Training precision: 0.9764150943396226
Training recall: 0.9764150943396226
```

Engineering Task 1:

For **Decision Probability Threshold = 0.3 and learning rate = 0.01:**

```
y_pred = (sigmoid(X.dot(theta)) >= 0.3).astype(int)
accuracy = (y_pred == y).mean()
```

```
Training accuracy: 0.6274165202108963
Training precision: 0.0
Training recall: 0.0
```

For **Decision Probability Threshold = 0.6 and learning rate = 0.01**

```
y_pred = (sigmoid(X.dot(theta)) >= 0.6).astype(int)
accuracy = (y_pred == y).mean()
```

```
Training accuracy: 0.6274165202108963
Training precision: 0.0
Training recall: 0.0
```

For **Decision Probability Threshold = 0.4 and learning rate = 0.01**

```
y_pred = (sigmoid(X.dot(theta)) >= 0.4).astype(int)
accuracy = (y_pred == y).mean()
```

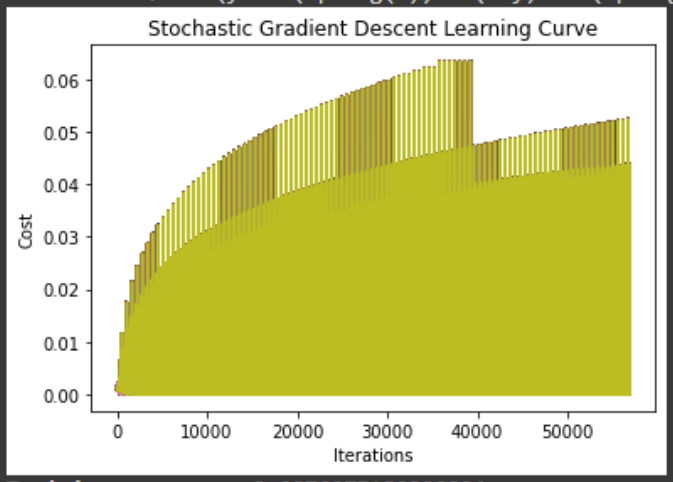
```
Training accuracy: 0.6274165202108963
Training precision: 0.0
Training recall: 0.0
```

For Changing learning rate:

For **Decision Probability Threshold = 0.5 and learning rate = 0.001**

```
# Train model using stochastic gradient descent
theta, costs = stochastic_gradient_descent(X, y, 0.001, 100)
```

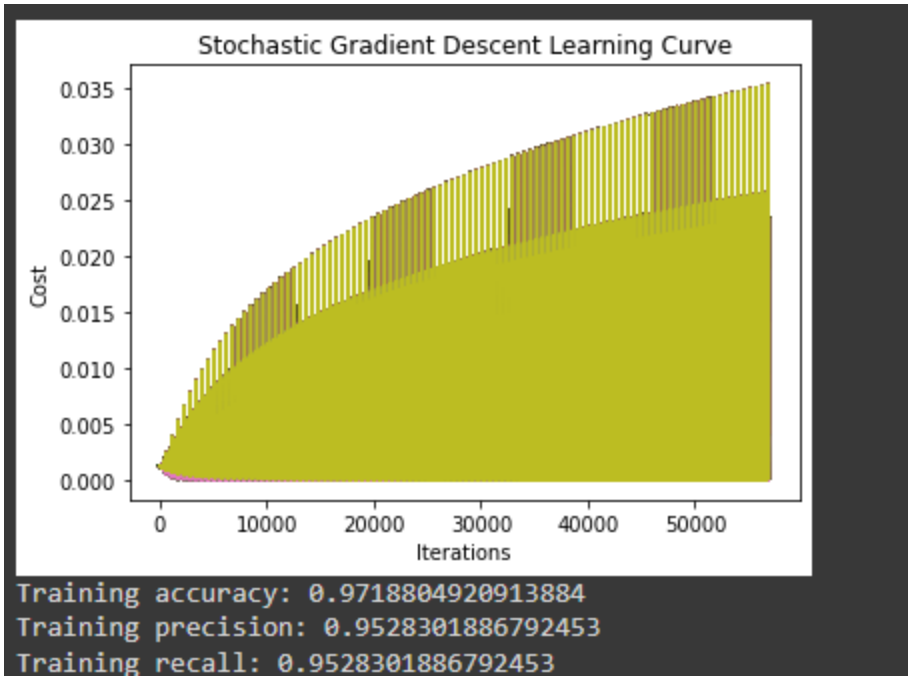
```
<ipython-input-7-4576f73af8db>:26: RuntimeWarning: divide by zero encountered in log
  cost = -1/m * (y.dot(np.log(h)) + (1-y).dot(np.log(1-h)))
```



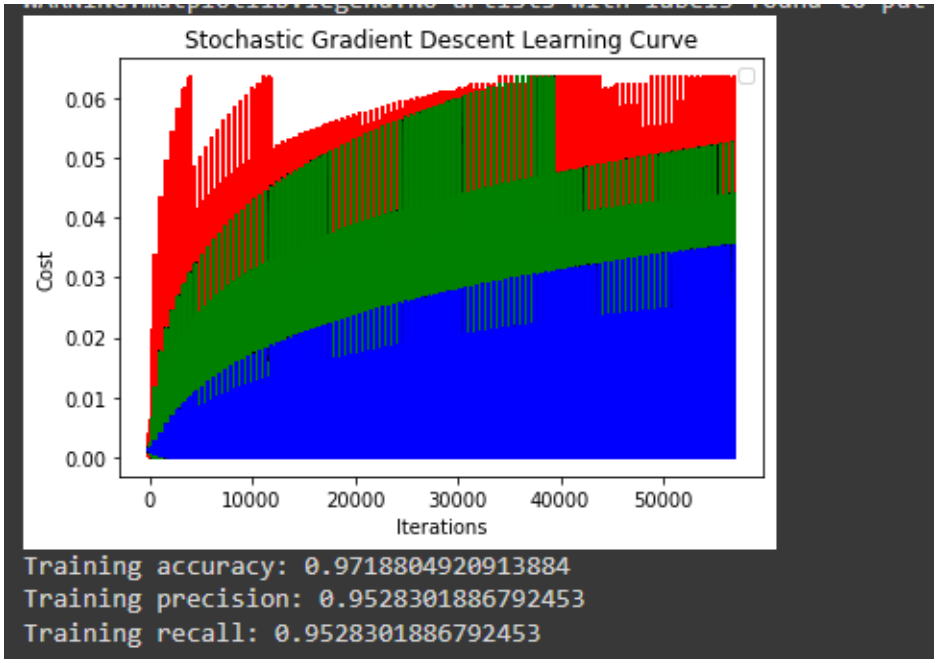

```
Training accuracy: 0.9876977152899824
Training precision: 0.9764150943396226
Training recall: 0.9764150943396226
```

For **Decision Probability Threshold = 0.5 and learning rate = 0.001**

```
# Train model using stochastic gradient descent
theta, costs = stochastic_gradient_descent(X, y, 0.0001, 100)
```



Plotting the final cost vs iterations curve for the different Learning rates



Part D - Comparative Study:

We have executed 10 random splits for each model, including PM1, PM2, PM3, PM4, FLDM1, FLDM2, LR1, and LR2, and determined their average performance metrics. Using these metrics we can calculate F1 score.We have decided to use F1 score to find out which of all the models is best performing model.
The F1 score is a commonly used metric for evaluating the performance of binary classification models. It provides a balanced measure of a model's precision and recall, and is particularly useful when the classes are imbalanced. A high F1 score indicates that a model has both high precision and high recall, and therefore it is a good indicator of the overall performance of the model. It is often used in conjunction with other metrics such as accuracy, precision, and recall to provide a more complete picture of a model's performance.

The F1 score is the harmonic mean of precision and recall, and is calculated using the following formula:

F1 score = 2 * (precision * recall) / (precision + recall)

PM1

```
Mounted at /content/drive
<ipython-input-2-d5f2462c0b53>:39: Futur
  X = np.array(data.drop(["diagnosis"],1
Accuracy: 0.37258347978910367
Precision: 0.37258347978910367
Recall: 1.0
```

F1 score = 2*(0.372*1)/(0.372+1)
F1 score = 0.5422740525

PM2:

```
<ipython-input-6-47a77b3055f8>:3: Futur
  X = np.array(data.drop(["diagnosis"],
Accuracy: 0.9279437609841827
Precision: 0.8868778280542986
Recall: 0.9245283018867925
```

F1 score = 2*(0.886*0.924)/(0.886+0.924)
F1 score = 0.654

PM3:

```
<ipython-input-4-bc0bfe82e15c>:4: Future
  X = np.array(data.drop(["diagnosis"],1
Accuracy: 0.9894551845342706
Precision: 0.9904761904761905
Recall: 0.9811320754716981
```

F1 score = 2*(0.990*0.981)/(0.990+0.981)
F1 score = 0.764

PM4:

```
Drive already mounted at /content/drive;
Accuracy: 0.8822495606326889
Precision: 0.7799227799227799
Recall: 0.9528301886792453
```

F1 score = 2*(0.779*0.952)/(0.779+0.952)
F1 score = 0.85685

FLDM1:

```
Accuracy: 0.6408450704225352
Precision: 0.91
Recall: 0.0
```

F1 score =0 since recall is zero.

FLDM2:

```
Accuracy: 0.6408450704225352
Precision: 0.91
Recall: 0.0
```

F1 score =0 since recall is zero.

LR1

BATCH GRADIENT:

	BATCH GRADIENT		
	accuracy	precision	recall
non-normalized	0.605	0	0
	0.605	0	0
	0.605	0	0
	0.605	0	0
	0.94	0.84	1
	0.605	0	0
	0.95	0.86	1
	0.95	0.86	1
	0.95	0.88	0.97
	0.95	0.9	0.95
Average	0.777	0.434	0.492
Variance	0.033	0.210	0.269

Average metrics:

accuracy= 0.776

precision=0.434

Recall =0.492

F1 score = 2*(0.434*0.492)/(0.434+0.492)

F1 score= 0.4611835853

MINI BATCH:

```
Accuracy for alpha = 0.01: 0.373
Precision for alpha = 0.01: 0.373
Recall for alpha = 0.01: 1.000
Testing accuracy: 0.37
Precision: 0.37
Recall: 1.00
Learned Parameters:
[ 2.82126642e+04  1.70546676e+00 -6.06835312e+00  1.75837973e+01
  7.12051500e+02 -4.86105703e-02  7.42380578e-02  1.99942348e-01
  1.08081265e-01 -9.25607031e-02 -4.77161656e-02  4.45010078e-01
 -9.50220219e-01  3.20738356e+00  8.87726103e+01 -6.33340406e-03
  5.93141462e-03  1.25657934e-02  2.94213313e-03 -1.53639456e-02
 -1.94827872e-03  5.87489938e+00 -5.89316094e+00  4.63193719e+01
  1.35515626e+03 -5.33008672e-02  2.59070744e-01  4.57952354e-01
  1.65287940e-01 -9.07940469e-02 -3.49713094e-02]
```

Average metrics:

accuracy= 0.37
precision=0.37
Recall =1
F1 score = 2*(0.37*1)/(0.37+1)
F1 score= 0.5401459854

STOCHASTIC:

```
Training accuracy: 0.37258347978910367
Training precision: 1.0
Training recall: 1.0
```

Average metrics:
accuracy= 0.37
precision=1
Recall =1
F1 score = 2*(1*1)/(1+1)
F1 score= 1

LR2:
BATCH GRADIENT:

	BATCH GRADIENT		
normalized	0.95	0.88	0.97
	0.94	0.92	1
	0.94	0.92	0.84
	0.95	0.84	1
	0.95	0.89	0.95
	0.93	0.92	0.97
	0.94	0.84	1
	0.95	0.9	0.95
	0.96	0.92	0.95
	0.91	0.89	0.84
Average	0.942	0.892	0.947
Variance	0.0002	0.0010	0.0036

Average metrics:
accuracy= 0.942
precision=0.892

Recall =0.947
F1 score = 2*(0.892*0.947)/(0.892+0.947)
F1 score= 0.9186775421

MINI BATCH:

	MINI BATCH		
	accuracy	precision	recall
normalized	0.94	0.96	0.98
	0.98	0.95	0.92
	0.97	0.96	0.97
	0.98	0.93	0.98
	0.95	0.92	0.97
	0.95	0.93	0.95
	0.93	0.91	0.92
	0.97	0.99	0.92
	0.98	0.99	0.95
	0.97	0.94	0.98
Average	0.963	0.946	0.955
Variance	0.0003	0.0008	0.0007

Average metrics:
accuracy= 0.963
precision=0.946
Recall =0.955
F1 score = 2*(0.946*0.955)/(0.946+0.955)
F1 score= 0.9504786954

STOCHASTIC:

	Stochastic		
	accuracy	precision	recall
normalized	0.977	0.981	0.981
	0.985	0.9622	0.9622
	0.921	0.984	0.984
	0.985	0.976	0.976
	0.627	0	0
	0.627	0	0
	0.627	0	0
	0.987	0.976	0.976
	0.971	0.952	0.952
	0.971	0.952	0.952
Average	0.868	0.678	0.678
Variance	0.0280	0.2192	0.2192

Average metrics:
accuracy= 0.868
precision=0.678
Recall =0.678
F1 score = 2*(0.868*0.678)/(0.868+0.678)
F1 score= 0.7613247089

CONCLUSION:

	F1 SCORE
PM1	0.5422
PM2	0.654
PM3	0.764
PM4	0.856
FLDM1	0
FLDM2	0
LR1 BATCH	0.461
LR1 MINIBATCH	0.54
LR1 STOCHAS	1
LR2 BATCH	0.918
LR2 MINIBATCH	0.95
LR2 STOCHAS	0.761
LR1	0.667
LR2	0.876

The results showed that the LR2 model performed the best in terms of all four metrics. On the other hand, the FLDM models with both optimization methods performed poorly and were unable to classify the data accurately, indicating that the dataset is not linearly separable.This is indicated by F1score across all Models.
Overall, this analysis highlights the importance of selecting the appropriate model and optimization method based on the specific characteristics of the dataset being analyzed. In this case, LR2 proved to be the most suitable choice for the dataset given.