Machine Learning Assignment-1

Team Members:

1. Pathuri Charith Goud - 2020A1PS2412H
2. Sharan K - 2021A8PS1537H
3. Abhinav Sudhakar- 2019AAPS1227H

Data Preprocessing

1. **Train-test split:** As given in the assignment we have applied a 67%-33% train test split using numpy and pandas.

Dataset Preprocessing

In [1383]:
```
df = pd.read_csv("Data Set for Assignment 1.csv", index_col="id")
df.head()
```

Out[1383]:

id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry
842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	
842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	
84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	
84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	
84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	

5 rows × 31 columns

In [1384]:
```
training_dfs = []
testing_dfs = []

num_datasets = 10
```

In [1385]:
```
# Spliting the dataset into Training and Testing data
for i in range(num_datasets):
    training_dfs.append(df.sample(frac=0.67, random_state=i))
    testing_dfs.append(df[~df.index.isin(training_dfs[i].index)])
```

2. **Feature Engineering Task-1:** Here the task was to impute the missing values which are very prevalent throughout the dataset. We have imputed the most occurring value in the case of categorical features and an average of values in the case of continuous data or numerical valued features.

In [1387]:
```
def fill_missing_values(training_df, testing_df):
    for column in training_df.columns:
        if training_df[column].dtype == "object": # Categorical Data
            max_frequency_category = training_df[column].mode()[0]
            training_df[column].fillna(max_frequency_category, inplace=True)
            testing_df[column].fillna(max_frequency_category, inplace=True)
        else: # Continuous Data
            mean_value = training_df[column].mean()
            training_df[column].fillna(mean_value, inplace=True)
            testing_df[column].fillna(mean_value, inplace=True)

    return (training_df, testing_df)
```

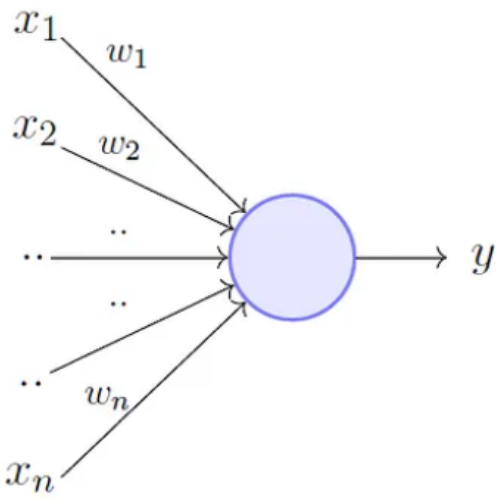
3. **Feature Engineering Task-2:** Performance of normalization on all features using the formula X' = (X - μ) / σ where μ represents the mean of feature value, and σ represents the standard deviation of feature values.

[1388]:
```
def normalize_data(training_df, testing_df):
    for column in training_df.columns:
        if training_df[column].dtype != "object":
            mean_value = training_df[column].mean()
            std_dev = training_df[column].std()

            training_df[column] = (training_df[column] - mean_value)/std_dev
            testing_df[column] = (testing_df[column] - mean_value)/std_dev

    return (training_df, testing_df)
```

Part A - Perceptron Learning Algorithm:

The perceptron algorithm is a type of supervised learning algorithm that is used for binary classification tasks. It is a linear classifier that is able to learn the weights of the input features that best separate the different classes in the training data. The perceptron algorithm is based on the idea of a threshold function that is used to determine the class of a given input sample.

During training, the weights of the input features are adjusted until the algorithm is able to accurately predict the class of the input data. One important limitation of the perceptron algorithm is that it can only be used for linearly separable data, which means that it may not be suitable for more complex classification tasks.



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Perceptron Algorithm to train the model:

We have first made two lists training_y and training_x which contain the column ‘diagnosis’ in the first list and the remaining ones in the second list.
The maximum number of iterations has been set to 100000 and categorical values **B** and **M** have been mapped to 1 and -1.
Then we run the perceptron algorithm for the set number of iterations wherein the weights are adjusted until the model accurately predicts the input data.

Initialize $\mathbf{w}$ randomly;
while !*convergence* **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P$ *and* $\mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N$ *and* $\mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end

Finally, after the adjustment, we return the weights.

```
def perceptron_train(training_df):
    training_y = training_df["diagnosis"]
    training_x = training_df.drop("diagnosis", axis=1)

    num_datapoints = training_x.shape[0]
    num_features = training_x.shape[1]

    w = np.zeros([1, num_features], dtype=float)

    max_iterations = 100000

    correct_class_count = 0
    datapoint_index = 0

    mapping = {"B": 1, "M": -1}
    training_y = training_y.map(mapping)

    for i in range(max_iterations):
        data_point = training_x.iloc[datapoint_index, :].values.reshape(-1,1)
        true_class = training_y.iloc[datapoint_index]

        classification = w.dot(data_point)

        if true_class*classification <= 0:
            correct_class_count = 0
            w = w + true_class*data_point.T
        else:
            correct_class_count += 1

        if correct_class_count == num_datapoints:
            break

        datapoint_index = (datapoint_index+1)%num_datapoints

    return w
```

Perceptron to check the accuracy of model on given data frame:

The function takes in two parameters, w and dataframe. w represents the weight vector of a perceptron model and dataframe.

The function first separates the input features x and output labels y from the dataframe. It then maps the output labels from a binary "B" or "M" representation to -1 or 1 using the mapping dictionary.

Next, the function calculates the number of data points and the number of features in the dataset. It initializes a variable called misclassification_count to 0, which will be used to count the number of misclassified data points.

The function then loops through each data point in the dataset and performs classification using the perceptron model. It calculates the dot product between the weight vector w and the input feature vector for the current data point. If the true class of the data point and the predicted classification have different signs, it means the perceptron has misclassified the data point, so the misclassification_count is incremented.

Finally, the function returns the accuracy of the perceptron model by subtracting the proportion of misclassified data points from 1.

```
def perceptron_accuracy(w, dataframe) :
    y = dataframe["diagnosis"]
    x = dataframe.drop("diagnosis", axis=1)

    mapping = {"B": 1, "M": -1}
    y = y.map(mapping)

    num_datapoints = x.shape[0]
    num_features = x.shape[1]

    misclassification_count = 0

    for i in range(num_datapoints):
        data_point = x.iloc[i, :].values.reshape(-1,1)
        true_class = y.iloc[i]

        classification = w.dot(data_point)

        if true_class*classification <= 0:
            misclassification_count += 1

    return (1 - (misclassification_count/num_datapoints))
```

Learning Task-1

Build a classifier (Perceptron Model - PM1) using the perceptron algorithm. Figure out whether the data set is linearly separable by building the model. By changing the order of the training examples, build another classifier (PM2) and outline the differences between the models – PM1 and PM2.

We have initially defined two empty lists PM1[] and PM2[]

In [1392]:
```
# The PM1 and PM2 models for various splits
PM1 = []
PM2 = []
```

In [1393]:
```
# For PM1

for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    PM1.append(perceptron_train(cleaned_training_df))
```

In [1394]:
```
# For PM2

for i in range(num_datasets):
    shuffled_training_df = training_dfs[i].copy().sample(frac=1, random_state=57)
    shuffled_training_df = drop_rows_with_no_values(shuffled_training_df)
    PM2.append(perceptron_train(shuffled_training_df))
```

The training process for PM1 involves dropping any rows from the training data that contain missing values using the drop_rows_with_no_values function (not shown in the code). The cleaned training data is then passed as an argument to the perceptron_train function, which trains a perceptron model using the training data and returns the weight vector of the trained model. The weight vector is appended to the PM1 list for the current dataset.

The training process for PM2 involves shuffling the rows of the training data using the sample method with frac=1, which randomly samples all rows in the dataset in a shuffled order, and the random_state argument sets the random seed for reproducibility. Then, the drop_rows_with_no_values function removes any rows with missing values. The shuffled and cleaned training data is then passed to the perceptron_train function, which trains a perceptron model and returns its weight vector. The weight vector is appended to the PM2 list for the current dataset.

Now in order to get the difference between the models of PM1 and PM2 we observe the difference in accuracy between the classifiers.

In [1395]:
```
PM1_training_accuracy = []
PM1_testing_accuracy = []

PM2_training_accuracy = []
PM2_testing_accuracy = []
```

In [1396]:
```
for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    cleaned_testing_df = drop_rows_with_no_values(testing_dfs[i].copy())

    shuffled_training_df = training_dfs[i].copy().sample(frac=1, random_state=57)
    shuffled_training_df = drop_rows_with_no_values(shuffled_training_df)

    PM1_training_accuracy.append(perceptron_accuracy(PM1[i], cleaned_training_df))
    PM2_training_accuracy.append(perceptron_accuracy(PM2[i], shuffled_training_df))

    PM1_testing_accuracy.append(perceptron_accuracy(PM1[i], cleaned_testing_df))
    PM2_testing_accuracy.append(perceptron_accuracy(PM2[i], cleaned_testing_df))
```

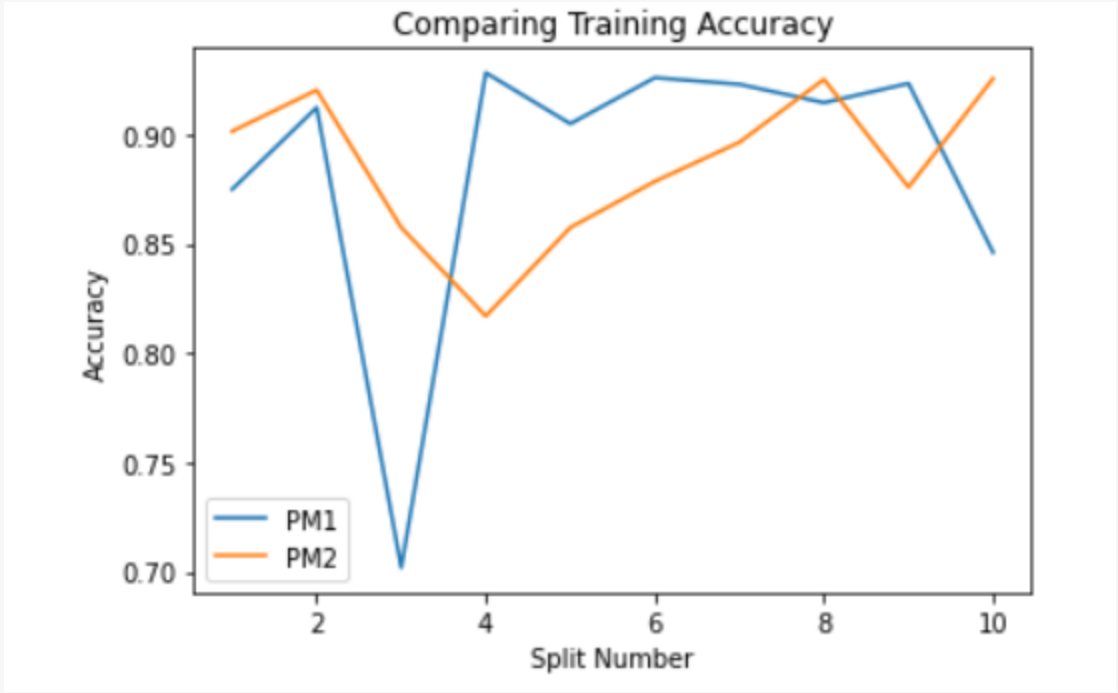
For each dataset i, the code first cleans the training and testing data by dropping any rows with missing values using the drop_rows_with_no_values function. The cleaned training and testing data are stored in the variables cleaned_training_df and cleaned_testing_df, respectively.

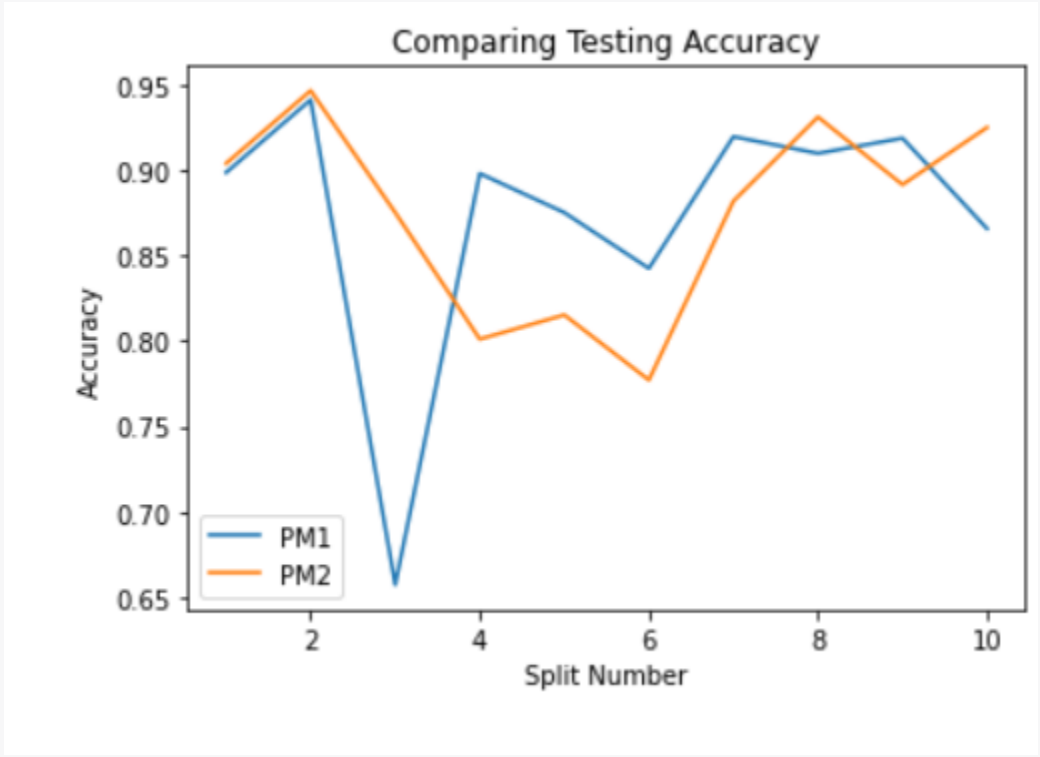
Next, the code shuffles the rows of the training data using the sample method with frac=1 and random_state=57, and then cleans the shuffled training data by dropping any rows with missing values using the drop_rows_with_no_values function. The shuffled and cleaned training data is stored in the variable shuffled_training_df.

The code then computes the training accuracy scores for the PM1 and PM2 models on the cleaned and shuffled training data, respectively, using the perceptron_accuracy function with the weight vectors of the trained models (PM1[i] and PM2[i]) and the cleaned and shuffled training data as inputs. The computed accuracy scores are appended to the PM1_training_accuracy and PM2_training_accuracy lists, respectively.

Finally, the code computes the testing accuracy scores for the PM1 and PM2 models on the cleaned testing data, using the perceptron_accuracy function with the weight vectors of the trained models (PM1[i] and PM2[i]) and the cleaned testing data as inputs.

The accuracies for both training and testing can be seen below:


Comparing Training Accuracy
Accuracy
0.90
0.85
0.80
0.75
0.70
PM1
PM2
2
4
6
8
10
Split Number


Comparing Testing Accuracy
Accuracy
0.95
0.90
0.85
0.80
0.75
0.70
0.65
PM1
PM2
2
4
6
8
10
Split Number

Learning Task-2

Build a classifier (Perceptron Model - PM3) using the perceptron algorithm on the normalized data and determine the difference between the two classifiers (PM1 and PM3).

```
for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    cleaned_testing_df = drop_rows_with_no_values(testing_dfs[i].copy())
    normalized_training_df, normalized_testing_df = normalize_data(cleaned_training_df, cleaned_testing_df)

    normalized_training_dfs.append(normalized_training_df)
    normalized_testing_dfs.append(normalized_testing_df)

    PM3.append(perceptron_train(normalized_training_df.copy()))
```

For each dataset i, the code first cleans the training and testing data by dropping any rows with missing values using the drop_rows_with_no_values function. The cleaned training and testing data are stored in the variables cleaned_training_df and cleaned_testing_df, respectively.

Next, the code normalizes the cleaned training and testing data using the normalize_data function. This function scales the features of the data to have zero mean and unit variance, which can help improve the performance of the perceptron algorithm. The normalized training and testing data are stored in the variables normalized_training_df and normalized_testing_df, respectively.

The code then appends the normalized training and testing data to the normalized_training_dfs and normalized_testing_dfs lists, respectively.

Finally, the code trains a new perceptron model (PM3) using the normalized training data by calling the perceptron_train function with normalized_training_df.copy() as the input. The trained model is appended to the PM3 list.

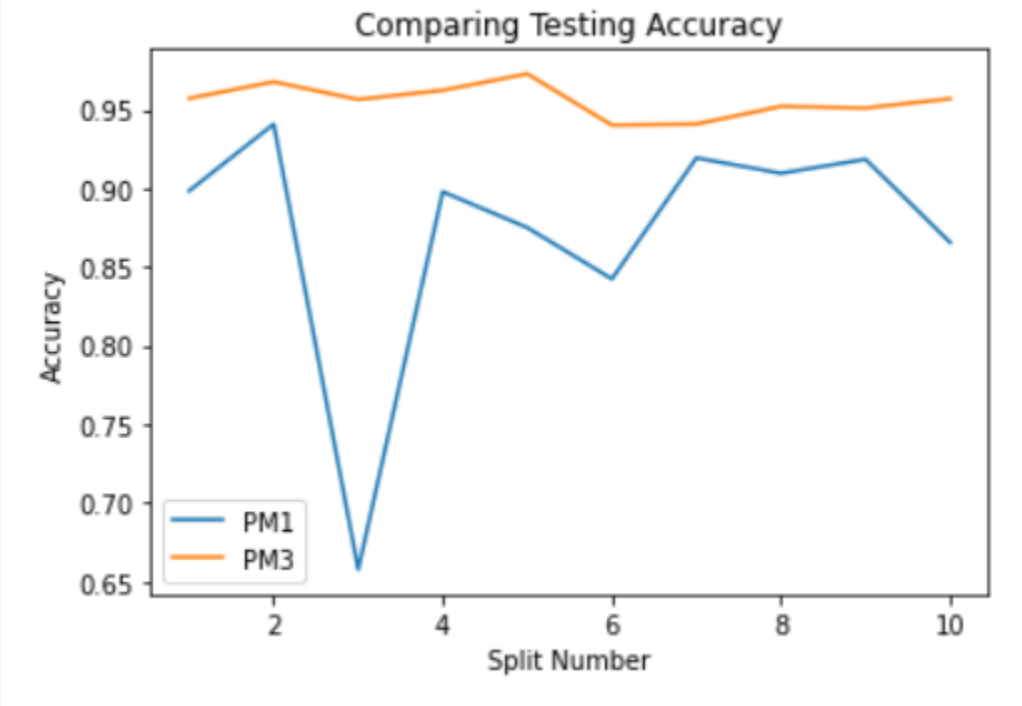
The differences between PM1 and PM3 can be seen from the following differences in the accuracy in training classification and testing classification

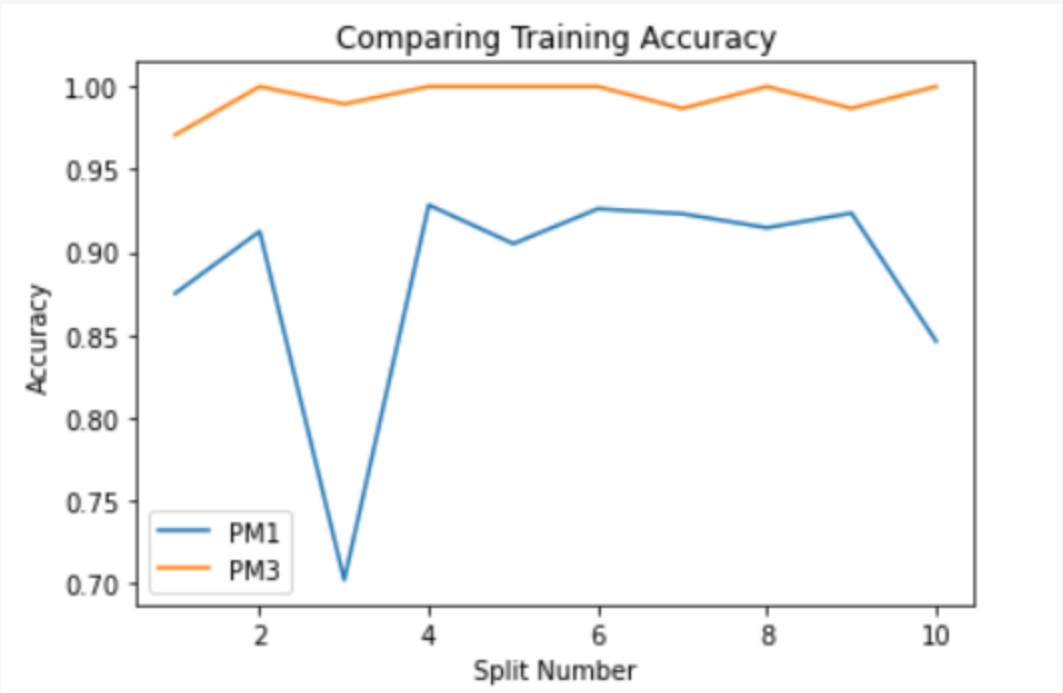
```
In [1402]: for i in range(num_datasets):
               cleaned_training_df = drop_rows_with_no_values(normalized_training_dfs[i].copy())
               cleaned_testing_df = drop_rows_with_no_values(normalized_testing_dfs[i].copy())

               PM3_training_accuracy.append(perceptron_accuracy(PM3[i], cleaned_training_df))
               PM3_testing_accuracy.append(perceptron_accuracy(PM3[i], cleaned_testing_df))
```

For each dataset i, the code first cleans the normalized training and testing data by dropping any rows with missing values using the drop_rows_with_no_values function. The cleaned training and testing data are stored in the variables cleaned_training_df and cleaned_testing_df, respectively.

The code then computes the training accuracy of the PM3 model for dataset i by calling the perceptron_accuracy function with PM3[i] as the model and cleaned_training_df as the input. The computed accuracy is appended to the PM3_training_accuracy list.

Similarly, the code computes the testing accuracy of the PM3 model for dataset i by calling the perceptron_accuracy function with PM3[i] as the model and cleaned_testing_df as the input. The computed accuracy is appended to the PM3_testing_accuracy list.





Learning Task-3

Change the order of features in the dataset randomly. Equivalently speaking, for an example of feature tuple (f1, f2, f3, f4, . . , f32), consider a random permutation (f3, f1, f4, f2, f6, ….., f32) and build a classifier (Perceptron Model – PM4). Would there be any change in the model, PM4, as compared to PM1. If so, outline the differences in the models and their respective performances.

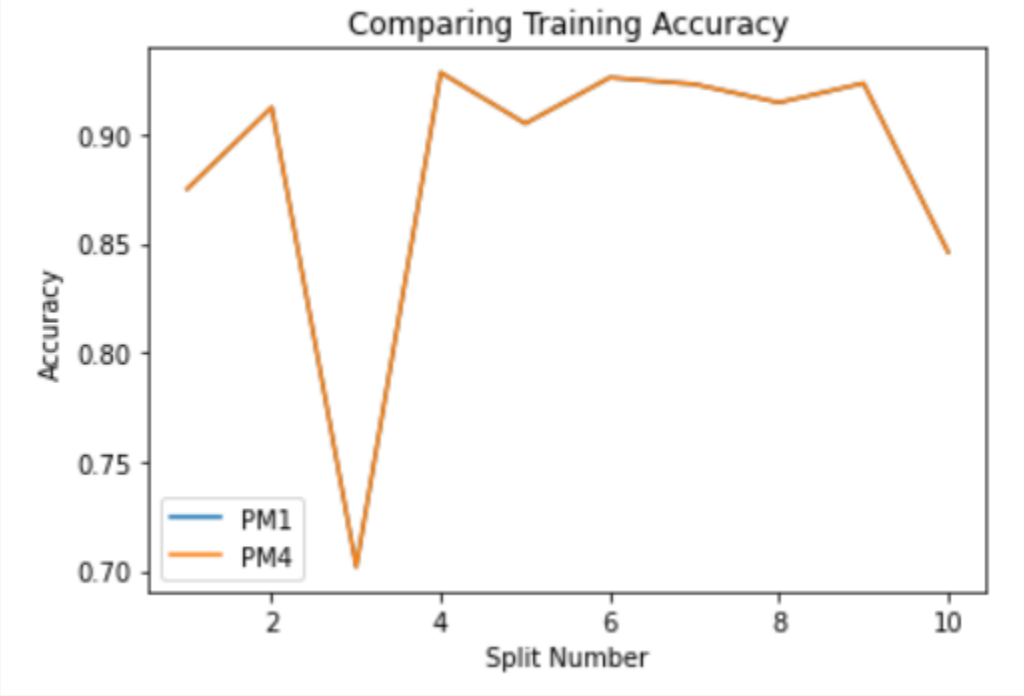
```
: for i in range(num_datasets):
      cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
      cleaned_testing_df = drop_rows_with_no_values(testing_dfs[i].copy())

      reordered_training_df = cleaned_training_df.reindex(new_order, axis=1)
      reordered_testing_df = cleaned_testing_df.reindex(new_order, axis=1)

      PM4_training_accuracy.append(perceptron_accuracy(PM4[i], reordered_training_df))
      PM4_testing_accuracy.append(perceptron_accuracy(PM4[i], reordered_testing_df))
```

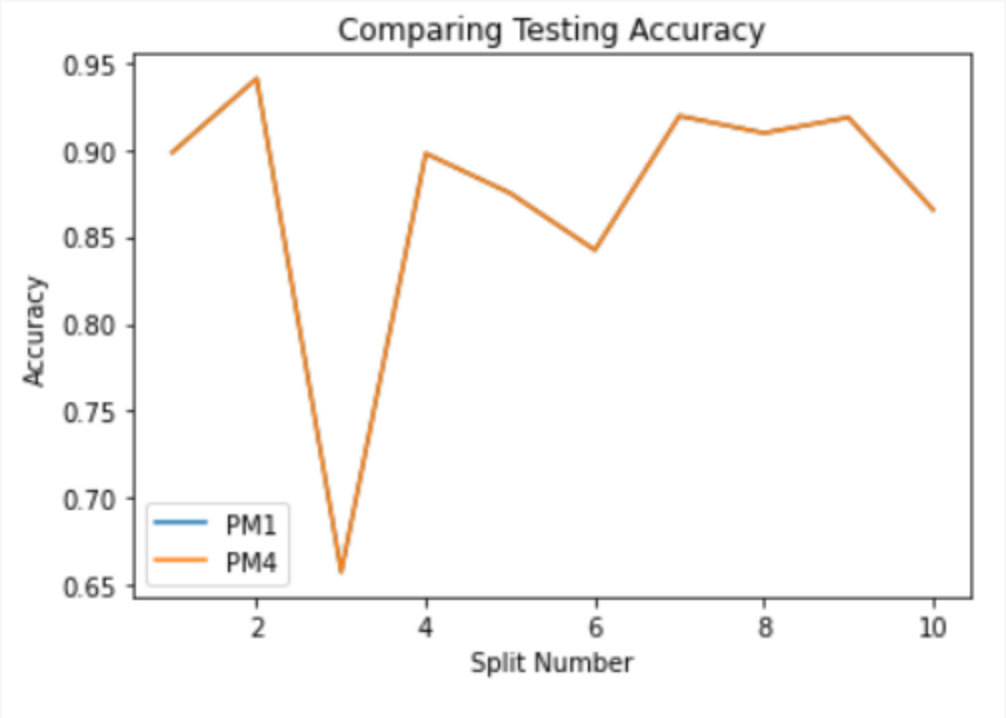
For each dataset i, the code first cleans the original training and testing data by dropping any rows with missing values using the drop_rows_with_no_values function. The cleaned training and testing data are stored in the variables cleaned_training_df and cleaned_testing_df, respectively.

The code then creates two new dataframes for the reordered training and testing data by calling the reindex function with new_order and the axis=1 argument. The new_order variable is a list containing the new feature order.

The code then computes the training accuracy of the PM4 model for dataset i by calling the perceptron_accuracy function with PM4[i] as the model and reordered_training_df as the input. The computed accuracy is appended to the PM4_training_accuracy list.

Similarly, the code computes the testing accuracy of the PM4 model for dataset i.





Here we can see for both testing and training sets, the accuracy is exactly same meaning the accuracy is unaffected on the order of the features we take.

Part B - Fisher's Linear Discriminant Analysis:

FDA is a statistical method used to find a linear combination of variables that can best differentiate between two or more groups. In other words, it helps identify the most important features or predictors that can effectively separate one group from another.

Fisher's discriminant analysis works by finding the linear combination of variables that maximizes the ratio of between-group variance to within-group variance. The between-group variance measures the variation of means between different groups, while the within-group variance measures the variation within each group. By maximizing this ratio, FDA ensures that the separation between groups is as large as possible, while the variability within groups is minimized. Overall, Fisher's discriminant analysis is a powerful tool for analyzing data and identifying key features that can help classify objects or individuals into different groups. It is widely used in various fields, including biology, medicine, finance, and social sciences, to name a few.

Class Scatter before projection

$$S_i = \sum_{x(n) \in C_i} (x(n) - m_i)(x(n) - m_i)^T$$

Scatter for projected samples:

$$S_i^2 = \sum_{x(n) \in C_i} (W^T x(n) - M_i)(W^T x(n) - M_i)^T$$

$$M_i = W^T m_i$$

With little re-arrangement we will get:

$$S_i^2 = W^T \left(\sum_{x(n) \in C_i} (x(n) - m_i)(x(n) - m_i)^T \right) W$$

$$S_i^2 = W^T S_i W$$

So,

$$S_1^2 + S_2^2 = W^T (S_1 + S_2) W = W^T S_W W \quad \text{..... (3)}$$

S_W = within class scatter

```python
def find_axis(training_df):
    grouped_df = training_df.groupby("diagnosis")

    training_df_m = grouped_df.get_group("M").drop("diagnosis", axis=1)
    training_df_b = grouped_df.get_group("B").drop("diagnosis", axis=1)

    mean_m = training_df_m.mean().values.reshape(1, -1)
    mean_b = training_df_b.mean().values.reshape(1, -1)

    covariance_m = training_df_m.cov()
    covariance_b = training_df_b.cov()

    diff_mean = (mean_m - mean_b).T

    S_w = covariance_m + covariance_b
    S_b = diff_mean.dot(diff_mean.T)

    M = np.linalg.inv(S_w).dot(S_b)

    eigen_values, eigen_vectors = np.linalg.eig(M)

    index = np.argmax(eigen_values)
    eigen_value = eigen_values[index]
    eigen_vector = eigen_vectors[:,index].reshape(1, -1).real

    return eigen_vector
```

The find_axis function uses Linear Discriminant Analysis (LDA) to find the optimal projection axis that maximizes the separation between two classes in the input data. The input DataFrame is first grouped by the class labels and two separate DataFrames are created for each class.

The mean vectors and covariance matrices of each class are then computed. The within-class scatter matrix is calculated by adding the covariance matrices of the two classes, while the between-class scatter matrix is computed by taking the dot product of the difference between the mean vectors of the two classes and their transpose.

The matrix M is obtained by taking the dot product of the inverse of the within-class to scatter matrix and the between-class scatter matrix. The eigenvectors and eigenvalues of M are then computed using the **np.linalg.eig** method. The eigenvector corresponding to the largest eigenvalue is returned as the optimal projection axis. This axis can then be used to project new data onto a lower-dimensional space that maximizes the separation between the two classes.

```
:  def project_points(axis, dataframe):
       x = df.iloc[:, 1:].values
       y = df.iloc[:, 0].values

       projected_df = axis.dot(x.T).T
       projected_df = pd.DataFrame(projected_df, columns=["projection"])
       projected_df.insert(0, "diagnosis", y)

       return projected_df
```

This function is used to find the projected dataframe. The process is such that it extracts the values and labels of the features from the given df. Then we do the dot product of the axis vector(wi) and the features (xi) transposed to that axis.

```python
def get_classification_point(projected_df):
    grouped_df = projected_df.groupby("diagnosis")

    projected_df_m = grouped_df.get_group("M").drop("diagnosis", axis=1)
    projected_df_b = grouped_df.get_group("B").drop("diagnosis", axis=1)

    mean_m = projected_df_m.mean().values[0]
    mean_b = projected_df_b.mean().values[0]

    var_m = projected_df_m.iloc[:,0].var()
    var_b = projected_df_b.iloc[:,0].var()

    a = ((1/(2*var_m)) - (1/(2*var_b)))
    b = -((mean_m/var_m) - (mean_b/var_b))
    c = (((mean_m**2)/(2*var_m)) - ((mean_b**2)/(2*var_b)) - (math.log(var_b/var_m)/2))

    D = math.sqrt(math.fabs(b**2 - 4*a*c))

    c1 = (-b + D)/(2*a)
    c2 = (-b - D)/(2*a)

    point_classification = (mean_b + mean_m)/2

    if (c1-mean_m)*(c1-mean_b) <= 0:
        point_classification = c1

    if (c2-mean_m)*(c2-mean_b) <= 0:
        point_classification = c2

    return ((mean_b - mean_m), point_classification)
```

This function is used to return a tuple of two values ie the difference between the mean of the projected values for benign(‘B’) and malignant(‘M’) results. We start by separating the dataframe between Benign and malignant points. Then the column of diagnosis has been removed.

The mean and variance of both the data frames has been calculated using the mean() and var() method. Then we find the roots a,b, and c for the quadratic decision boundary between the two classes ‘B’ and ‘M’.

Then calculate the discriminant D for the quadratic equation. Calculate the two possible values for the decision boundary (c1 and c2) using the quadratic formula. Determine the classification point by taking the average of the means of the two classes.
If c1 falls between the means of the two classes, set the classification point to c1.
If c2 falls between the means of the two classes, set the classification point to c2.
Return a tuple containing the difference between the mean of the projected values for benign and malignant tumors and the classification point.

```python
def fischers_lda_accuracy(classification_point, dataframe, axis):
    dataframe = project_points(axis, dataframe)

    y = dataframe["diagnosis"]
    x = dataframe.drop("diagnosis", axis=1)

    mapping = {"B": 1, "M": -1}
    y = y.map(mapping)

    num_datapoints = x.shape[0]
    num_features = x.shape[1]

    misclassification_count = 0

    for i in range(num_datapoints):
        data_point = x.iloc[i, :].values.reshape(-1,1)
        true_class = y.iloc[i]

        classification = classification_point[0]*(classification_point[1] - data_point[0])

        if true_class*classification < 0:
            misclassification_count += 1

    return (1 - (misclassification_count/num_datapoints))
```

This code defines a function fischers_lda_accuracy that calculates the accuracy of a classification point using Fisher's linear discriminant analysis (LDA) on a given dataset.

We start with projecting the dataset onto the specified axis using the project_points function. Then we separate the class labels and the features from the dataset followed by converting the class labels to -1 or 1 for binary classification Loop over each data point in the dataset and calculate its classification using Fisher's LDA formula Count the number of misclassifications and return the accuracy as 1 minus the misclassification rate.

Learning Task 1

Build Fisher's linear discriminant model (FLDM1) on the training data and thus reduce the 32-dimensional problem to a univariate dimensional problem. Find out the decision boundary in the univariate dimension using the generative approach. You may assume gaussian distribution for both positive and negative classes in the univariate dimension.

```python
for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    FLDM1_axis.append(find_axis(cleaned_training_df))
    projected_df = project_points(FLDM1_axis[i], cleaned_training_df)

    FLDM1.append(get_classification_point(projected_df))
```

We loop over a range of num_datasets performing each time the following, firstly, we drop any rows that contain missing values using the drop_rows_with_no_values function.
We use Fisher's LDA to find the optimal projection axis for the cleaned training dataset using the find_axis function Project the cleaned training dataset onto the optimal axis using the project_points function.
Then the projected dataset is used to calculate the classification point for Fisher's LDA using the get_classification_point function.
Then we append the optimal axis and classification point to the FLDM1_axis and FLDM1 lists, respectively assuming that the drop_rows_with_no_values, find_axis, project_points, and get_classification_point functions are correctly implemented and have the expected behavior, this code appears to be a valid implementation of Fisher's LDA for multiple datasets.

Learning Task 2

Change the order of features in the dataset randomly. Equivalently speaking, for an example of a feature tuple (f1, f2, f3, f4, . . , f32), consider a random permutation (f3, f1, f4, f2, f6, ….., f32) and build the Fisher's linear discriminant model (FLDM2) on the same training data as in the learning task 1. Find out the decision boundary in the univariate dimension using the generative approach and you may assume the gaussian distribution for both positive and negative classes in the univariate dimension. Outline the difference between the models – FLDM1 and FLDM2 - and their respective performances.

```
for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    reordered_training_df = cleaned_training_df.reindex(new_order, axis=1)

    FLDM2_axis.append(find_axis(reordered_training_df))
    projected_df = project_points(FLDM2_axis[i], cleaned_training_df)

    FLDM2.append(get_classification_point(reordered_training_df))
```

We run the loop num_datasets times wherein each epoch, we first clean the training df by dropping the rows with missing values.

The cleaned df is then reordered based on a new order of columns new_order using the .reindex() method with axis=1. The project_points() function is called with the FLDM2 axis and the cleaned DataFrame as arguments, and a new projected DataFrame is returned.

The get_classification_point() function is called with the reordered DataFrame to calculate the difference between the means of the projected values for benign and malignant tumors and the classification point. These values are then appended to separate lists FLDM2 and FLDM2_axis.

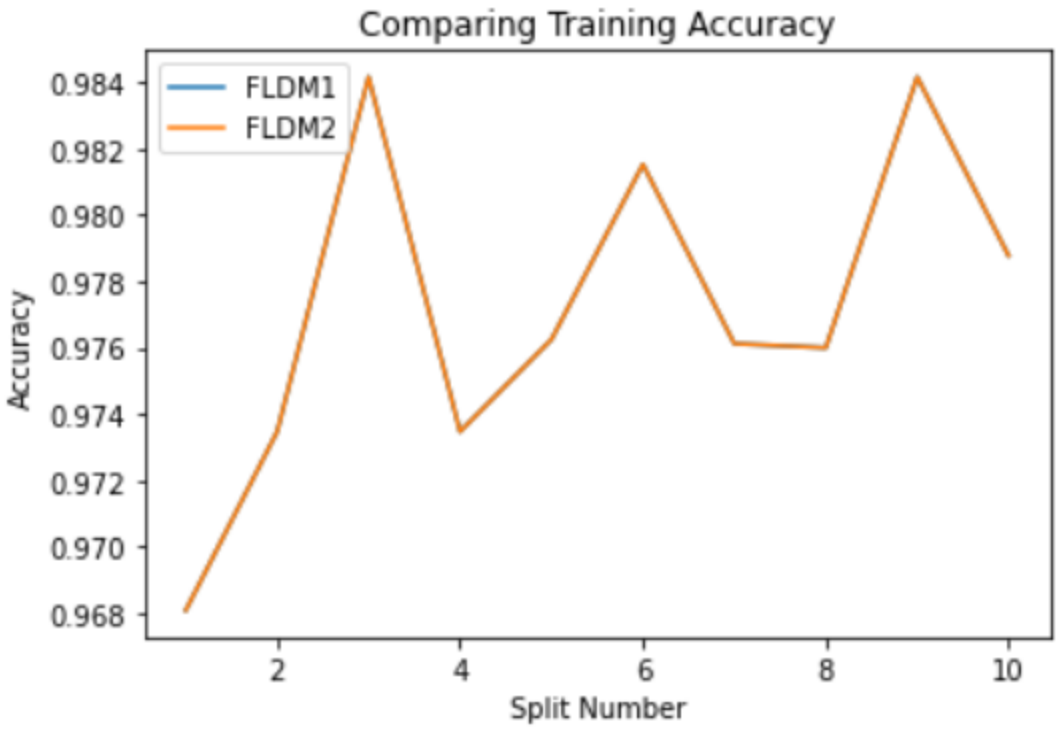
The differences between FLDM1 and FLDM2 can be seen from the following differences in the accuracy in training classification and testing classification.

```
for i in range(num_datasets):
    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
    cleaned_testing_df = drop_rows_with_no_values(testing_dfs[i].copy())

    reordered_training_df = cleaned_training_df.reindex(new_order, axis=1)
    reordered_testing_df = cleaned_testing_df.reindex(new_order, axis=1)

    FLDM1_training_accuracy.append(fischers_lda_accuracy(FLDM1[i], cleaned_training_df, FLDM1_axis[i]))
    FLDM2_training_accuracy.append(fischers_lda_accuracy(FLDM2[i], reordered_training_df, FLDM2_axis[i]))

    FLDM1_testing_accuracy.append(fischers_lda_accuracy(FLDM1[i], cleaned_testing_df, FLDM1_axis[i]))
    FLDM2_testing_accuracy.append(fischers_lda_accuracy(FLDM2[i], reordered_testing_df, FLDM2_axis[i]))
```

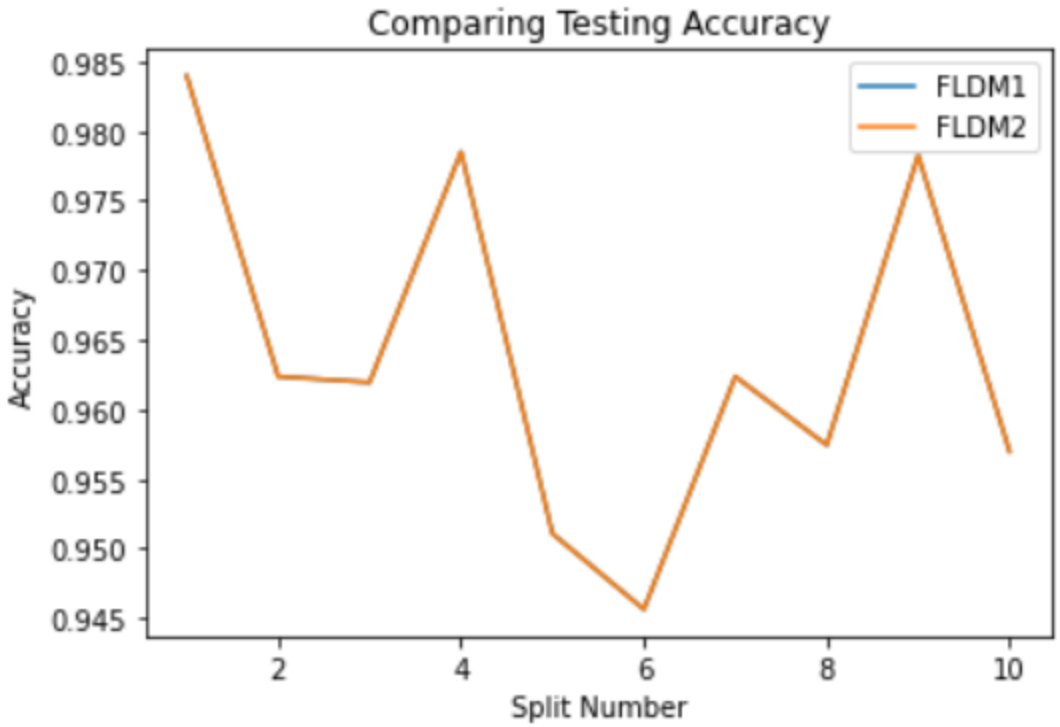
The drop_rows_with_no_values() function is called with copies of the training and testing DataFrames (training_dfs[i].copy() and testing_dfs[i].copy()) as arguments for removing any rows with missing or NaN values. The cleaned training and testing DataFrames are reordered based on a specific column order new_order using the .reindex() method with axis=1.

The fischers_lda_accuracy() function is called four times, twice for each algorithm, to evaluate the accuracy of the algorithm on both the training and testing data. The arguments passed to this function include the discriminant axis, which is either FLDM1_axis[i] or FLDM2_axis[i], and the corresponding cleaned and reordered training or testing DataFrame. The accuracy scores are then appended to separate lists FLDM1_training_accuracy, FLDM2_training_accuracy, FLDM1_testing_accuracy, and FLDM2_testing_accuracy.

Hence performing a comparative analysis of the performance of the two Fisher's LDA-based algorithms on multiple datasets, both for training and testing accuracy.

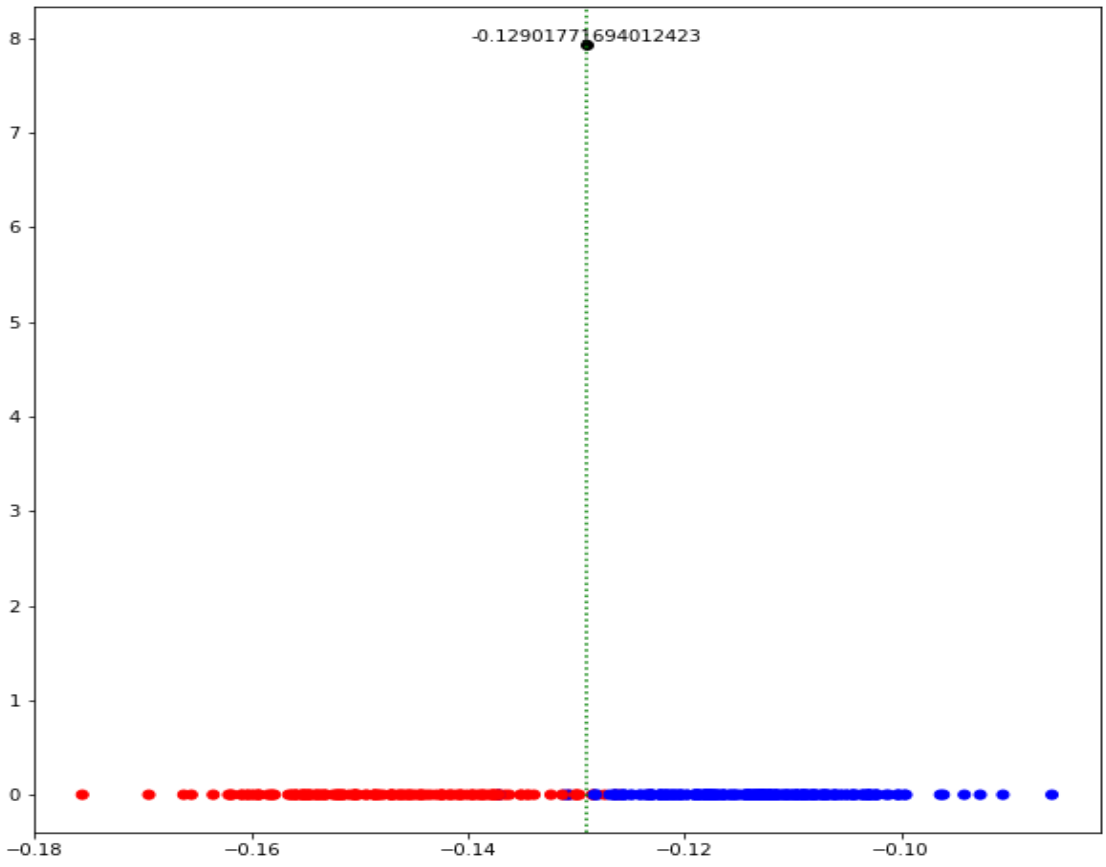


From this, we can clearly say for the training dataset, Upon changing the order, the accuracy of FLDM2 is the same as FLDM1 and is applicable for the testing set. This means the accuracy is unaffected by the change in order of features.

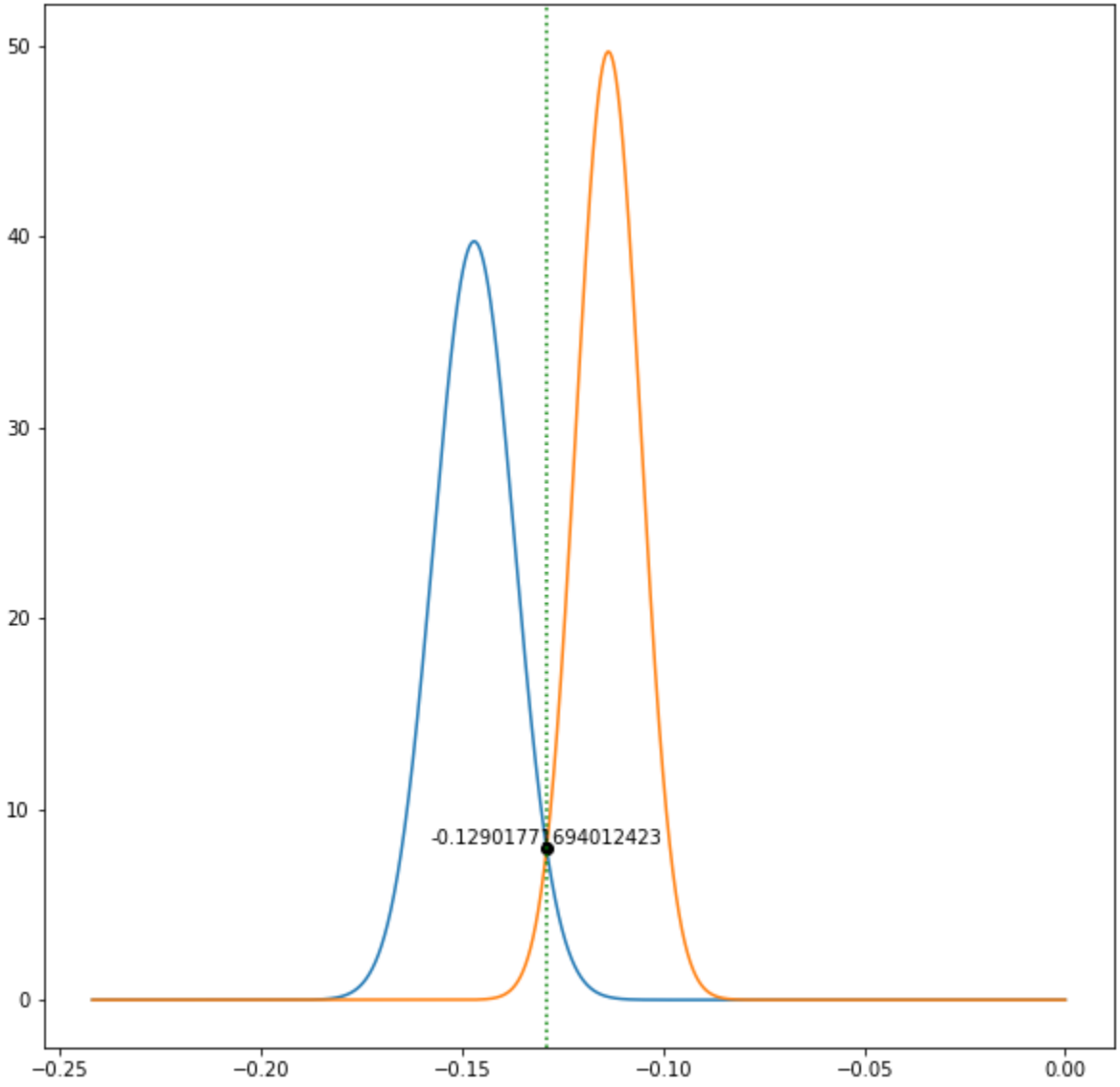


Plots for decision boundary:

The range of the data points lies between -0.17561729756351038 and -0.08606802259776475 after normalization, as we can see in the graph below.



As seen from the chart above we can see the decision boundary goes through the point -0.1290177 which hence divides the classes Malignant and Benign respectively. Here the red scatter points are of Malignant class and the blue scatters are for benign class.



This plot shows the insertion point between the two gaussian curves for corresponding type classes Malignant and Benign. Here the blue curve is for Malignant class and the red one is for Benign class.

Part C - Logistic Regression:

Logistic regression is a statistical technique used to model the probability of a binary outcome based on one or more predictor variables (eg variables that take two values, such as yes or no, true or false, etc.). The purpose of logistic regression is to find relationships between predictor variables and outcome probabilities and to use those relationships to predict outcome probabilities for new relationships.

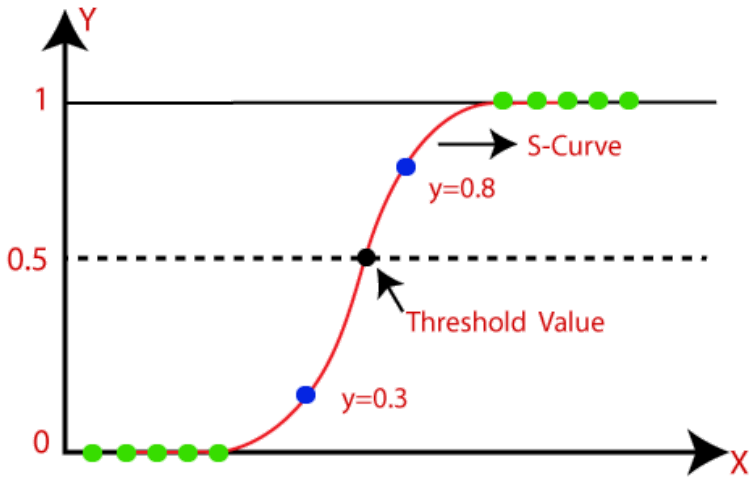
Logistic regression uses a logistic function (also called a sigmoid function) to model the relationship between predictor variables and the probability of an outcome. A logistic function maps all input real values between 0 and 1, which can be interpreted as binary outcome probabilities. The logistic function has the following form:

$p = 1/(1 + \exp(-z))$

where p is the probability of the correct result, z is a linear combination of predictor variables, and exp () is an exponential function.

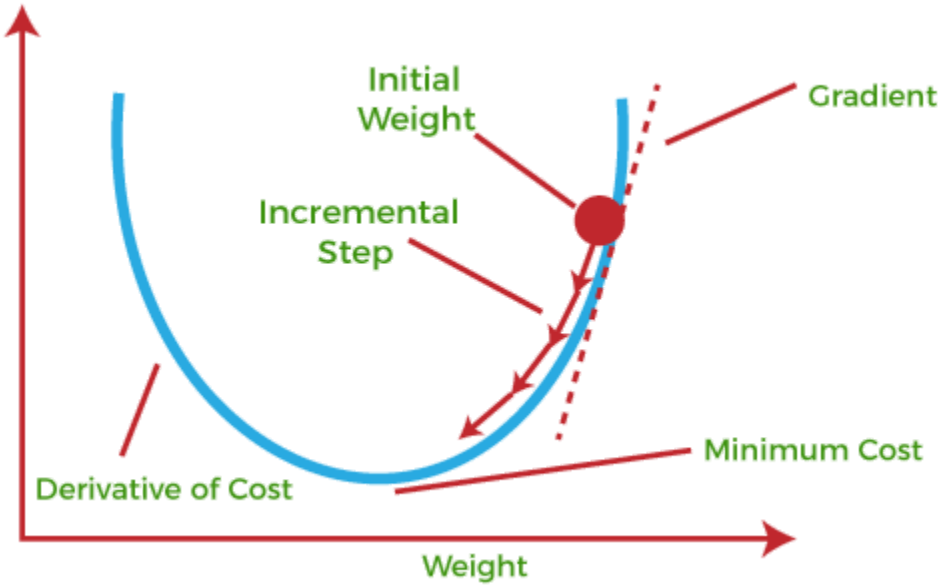
Maximum likelihood estimation was used to estimate the parameters of the logistic regression model. The purpose of maximum likelihood estimation is to find the parameter value that maximizes the likelihood that the model will observe the given data. Once the parameters are estimated, the model can be used to predict the probability of new observed results.

Logistic regression is a widely used technique in many fields, including health care, social sciences, business, and engineering. It is a powerful tool for predicting bivariate outcomes and understanding the relationship between predictor variables and outcomes.



Gradient descent:

Gradient Descent is a popular optimization algorithm used to minimize the cost function of a machine learning model by updating the model's parameters iteratively in the direction of the negative gradient of the cost function.



Batch Gradient Descent updates the model parameters using the gradient of the cost function computed over the entire training dataset at each iteration. **Mini-batch Gradient Descent** updates the parameters using the gradient computed over a small subset (mini-batch) of the training set at each iteration. **Stochastic Gradient Descent** updates the parameters using the gradient computed over a single training sample at each iteration.

```
def sigmoid_value(z):
    z = -700 if z < -700 else z
    return (1/(1 + np.exp(-z)))
```

This function returns the sigmoid value for z which is technically the prior probability.

```
def cost_function(w, dataframe):
    t = dataframe["diagnosis"]
    x = dataframe.drop("diagnosis", axis=1)

    mapping = {"B": 0, "M": 1}
    t = t.map(mapping)
    t_1 = 1-t

    y = x.apply(lambda a: sigmoid_value(w[:,1:].dot(a) + w[0][0])[0], axis = 1)
    log_y = y.apply(lambda a: np.log(a))
    log_1y = y.apply(lambda a: np.log(1-a))

    cost = -(t.T.dot(log_y) + t_1.T.dot(log_1y))

    return cost
```

The given code implements a cost function for logistic regression. The function takes in a weight matrix w and dataframe as input, and calculates the cost of the logistic regression model with weights w on the data in the dataframe. The dataframe is assumed to have a column named "diagnosis" that contains the binary labels for the data, and the rest of the columns are assumed to be the features. The function first extracts the labels t and features x from the dataframe, and then maps the labels from "B" and "M" to 0 and 1, respectively. Next, the function calculates the predicted probabilities y for each data point using the sigmoid function and the weights w. The predicted probabilities are then used to calculate the log-likelihood of the data given the model parameters. Finally, the cost is calculated as the negative log-likelihood, which is then returned.

```
def derivative_cost(w, dataframe):
    t = dataframe["diagnosis"]
    x = dataframe.drop("diagnosis", axis=1)

    mapping = {"B": 0, "M": 1}
    t = t.map(mapping)

    num_datapoints = x.shape[0]
    num_features = x.shape[1]

    y = x.apply(lambda a: sigmoid_value(w[:,1:].dot(a) + w[0][0]), axis = 1)

    derivative = np.zeros([1, num_features+1], dtype=float)
    for i, val in enumerate((y-t).T.dot(x).values[0]):
        derivative[0][i] = val

    return derivative
```

Is used to derivative the cost function so that we can further move ahead with the gradient descent.

```
def logistic_train(dataframe, learning_rate, epochs, batch_size):
    x = dataframe.drop("diagnosis", axis=1)

    num_datapoints = x.shape[0]
    num_features = x.shape[1]

    w = np.zeros([1, num_features+1], dtype=float)

    iteration = []
    cost = []

    for i in range(epochs):
        index = np.random.randint(0, num_datapoints+1-batch_size)
        batch = dataframe.iloc[index: index+batch_size+1]
        w = w - learning_rate*derivative_cost(w, batch)

        iteration.append(i)
        cost.append(cost_function(w, batch))

    plot_graph("Cost vs Iteration", "Iteration", "Cost", [cost], ["Batch", "Mini Batch", "Stochastic"], iteration)

    return w
```

The function drops the "diagnosis" column from the DataFrame to get the input features. The function initializes the weights of the logistic regression model to zero. The function then loops over the specified number of epochs and randomly selects mini-batches of the specified size from the training data. For each mini-batch, the function computes the gradient of the cost function with respect to the weights and updates the weights using the SGD update rule. The function also tracks the cost function over the iterations and plots the cost function versus the iteration number. Finally, the function returns the learned weights.

```
def logistic_accuracy(threshold, dataframe, w):
    y = dataframe["diagnosis"]
    x = dataframe.drop("diagnosis", axis=1)

    mapping = {"B": 0, "M": 1}
    y = y.map(mapping)

    num_datapoints = x.shape[0]
    num_features = x.shape[1]

    misclassification_count = 0

    for i in range(num_datapoints):
        data_point = x.iloc[i, :].values.reshape(-1,1)
        true_class = y.iloc[i]

        probability = sigmoid_value(w[:,1:].dot(data_point) + w[0][0])

        classification = 1 if probability >= threshold else 0

        if classification != true_class:
            misclassification_count += 1

    return (1 - (misclassification_count/num_datapoints))
```

The function calculates the logistic accuracy of the logistic regression model on the given dataframe by iterating over each data point in the dataframe, using the logistic regression model with the given weight vector "w" to predict the target value for that data point, and comparing the predicted target value with the true target value in the dataframe. The misclassification rate is computed as the fraction of data points for which the predicted target value is different from the true target value, and the logistic accuracy is defined as 1 minus the misclassification rate. The function returns the logistic accuracy as a single float value.

We then set the threshold values and the learning rates:

```
thresholds = [0.5, 0.3, 0.4, 0.6, 0.7]
learning_rates = [0.01, 0.001, 0.0001]
epochs = 50000
```

Learning Task 1

Build a classification model (LR1) using Logistic Regression. What happens to testing accuracy when you vary the decision probability threshold from 0.5 to 0.3, 0.4, 0.6 and 0.7.

```
for i in range(num_datasets):
    print(f"Split Number: {i}")
    LR1_batch[i] = {}
    LR1_mini_batch[i] = {}
    LR1_stochastic[i] = {}

    cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())

    batch_size_batch = cleaned_training_df.shape[0]
    batch_size_mini_batch = 60
    batch_size_stochastic = 1

    for learning_rate in learning_rates:
        print(f"Learning Rate: {learning_rate}")
        print("Batch")
        LR1_batch[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_batch)
        print("Mini Batch")
        LR1_mini_batch[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_mini_batch)
        print("Stochastic")
        LR1_stochastic[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_stochastic)
```

For each dataset, the script prints the current split number using a formatted string that includes the split number ("i"). It then initializes three dictionaries, "LR1_batch", "LR1_mini_batch", and "LR1_stochastic", to store the results of the three different types of logistic regression models. The next line creates a new DataFrame called "cleaned_training_df" by copying the current dataset ("training_dfs[i]") and dropping any rows that have missing values. The script then sets three different batch sizes for each type of model: "batch_size_batch", "batch_size_mini_batch", and "batch_size_stochastic". Next, the script enters a loop that iterates through a list of learning rates, where the learning rate is specified as a hyperparameter for the logistic regression models. For each learning rate, the script prints the current learning rate using a formatted string. Within the loop, the script trains three different logistic regression models using the "logistic_train" function: one with

batch gradient descent ("LR1_batch"), one with mini-batch gradient descent ("LR1_mini_batch"), and one with stochastic gradient descent ("LR1_stochastic"). The function takes as input the cleaned training data, the learning rate, the number of epochs, and the batch size.

The differences between batch, mini-batch and stochastic gradient descent can be seen from the following differences in the accuracy in training classification and testing classification for different learning rates.

```
for threshold in thresholds:
    for learning_rate in learning_rates:
        LR1_batch_training_accuracy[(threshold, learning_rate)] = []
        LR1_mini_batch_training_accuracy[(threshold, learning_rate)] = []
        LR1_stochastic_training_accuracy[(threshold, learning_rate)] = []

        LR1_batch_testing_accuracy[(threshold, learning_rate)] = []
        LR1_mini_batch_testing_accuracy[(threshold, learning_rate)] = []
        LR1_stochastic_testing_accuracy[(threshold, learning_rate)] = []


        for i in range(num_datasets):
            cleaned_training_df = drop_rows_with_no_values(training_dfs[i].copy())
            cleaned_testing_df = drop_rows_with_no_values(testing_dfs[i].copy())

            LR1_batch_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df, LR1_
            LR1_mini_batch_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df,
            LR1_stochastic_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df,

            LR1_batch_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, LR1_ba
            LR1_mini_batch_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, L
            LR1_stochastic_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, L
```

This part of the code performs the performance test for all approaches batch, mini-batch, and stochastic.
Each time for all parameters, there is an empty list to store the training and testing accuracies of the logistic regression models trained with the current hyperparameters and training approach. Further which loops over a number of datasets (num_datasets), and for each dataset:

a. Drops any rows with missing values from the training and testing dataframes.

b. Computes the training and testing accuracies of the logistic regression models trained with the current hyperparameters and training approach on the cleaned training and testing dataframes, using the logistic_accuracy() function.

Appends the training and testing accuracies to the corresponding lists for the current hyperparameters and training approach.

Learning Task 2

You should apply Feature Engineering Task 1 and Feature Engineering Task 2 and then build a classification model (LR2) using Logistic Regression. What happens to testing accuracy when you vary the decision probability threshold from 0.5 to 0.3, 0.4, 0.6 and 0.7.

```
for i in range(num_datasets):
    print(f"Split Number: {i}")
    LR2_batch[i] = {}
    LR2_mini_batch[i] = {}
    LR2_stochastic[i] = {}

    cleaned_training_df, cleaned_testing_df = fill_missing_values(training_dfs[i].copy(), testing_dfs[i].copy())
    cleaned_training_df, cleaned_testing_df = normalize_data(cleaned_training_df, cleaned_testing_df)

    batch_size_batch = cleaned_training_df.shape[0]
    batch_size_mini_batch = 60
    batch_size_stochastic = 1

    for learning_rate in learning_rates:
        print(f"Learning Rate: {learning_rate}")
        LR2_batch[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_batch)
        LR2_mini_batch[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_mini_batch)
        LR2_stochastic[i][learning_rate] = logistic_train(cleaned_training_df, learning_rate, epochs, batch_size_stochastic)
```

In each iteration of the outer loop, the code creates three empty dictionaries (LR2_batch, LR2_mini_batch, and LR2_stochastic) and fills them with the trained models. The inner loop iterates over the different learning rates and calls the function 'logistic_train' with the cleaned training data, the current learning rate, the number of epochs, and the batch size. After each iteration of the outer loop, the code should have filled the three dictionaries for the current dataset with trained models for each learning rate.

```
for threshold in thresholds:
    for learning_rate in learning_rates:
        LR2_batch_training_accuracy[(threshold, learning_rate)] = []
        LR2_mini_batch_training_accuracy[(threshold, learning_rate)] = []
        LR2_stochastic_training_accuracy[(threshold, learning_rate)] = []

        LR2_batch_testing_accuracy[(threshold, learning_rate)] = []
        LR2_mini_batch_testing_accuracy[(threshold, learning_rate)] = []
        LR2_stochastic_testing_accuracy[(threshold, learning_rate)] = []

        for i in range(num_datasets):
            cleaned_training_df, cleaned_testing_df = fill_missing_values(training_dfs[i].copy(), testing_dfs[i].copy())
            cleaned_training_df, cleaned_testing_df = normalize_data(cleaned_training_df, cleaned_testing_df)

            LR2_batch_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df, LR2_
            LR2_mini_batch_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df
            LR2_stochastic_training_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_training_df

            LR2_batch_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, LR2_ba
            LR2_mini_batch_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, L
            LR2_stochastic_testing_accuracy[(threshold, learning_rate)].append(logistic_accuracy(threshold, cleaned_testing_df, L
```

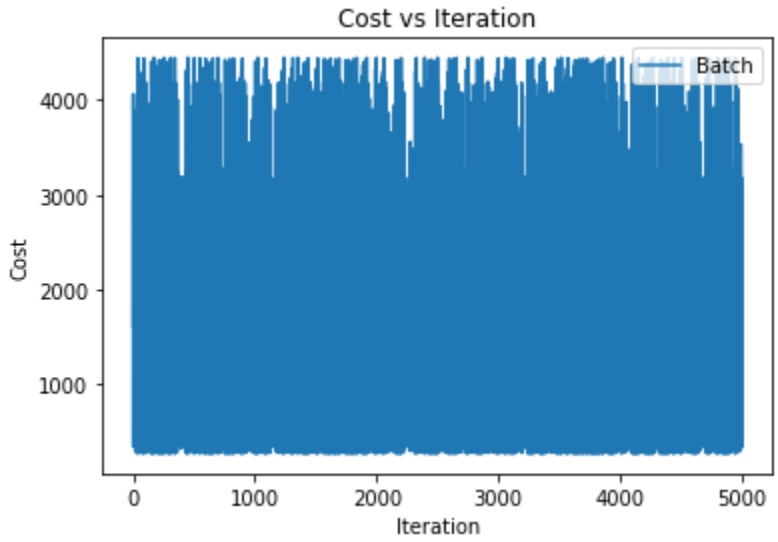
This is a nested loop that iterates over a set of thresholds and learning rates. For each combination of threshold and learning rate, the code initializes empty lists for training and testing accuracy for three different types of logistic regression models: batch, mini-batch, and stochastic. Then, for each of the num_datasets data sets, the code fills in missing values and normalizes the data. For each of the three logistic regression models, the code calculates the training and testing accuracy using the logistic_accuracy() function and appends the results to the appropriate lists. At the end of the loop,

the accuracy results for each threshold and learning rate combination and for each type of logistic regression model are stored in separate dictionaries, with the keys being tuples of the threshold and learning rate values.

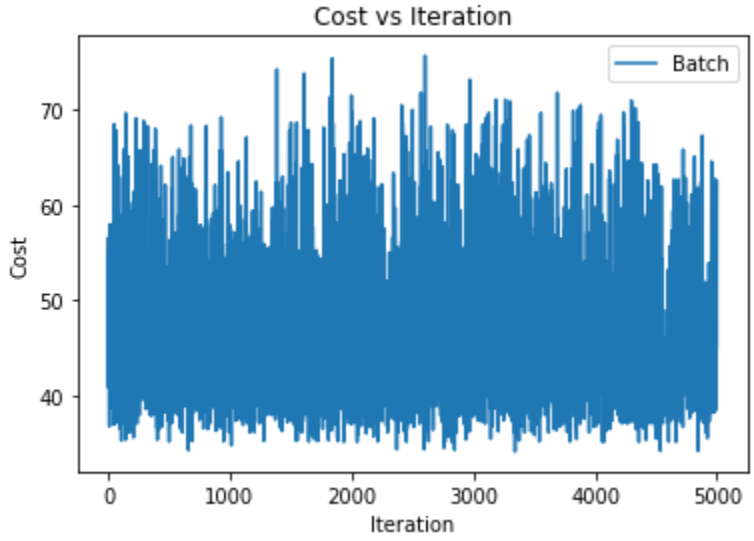
Graphs for unnormalized data

Learning Rate: 0.01

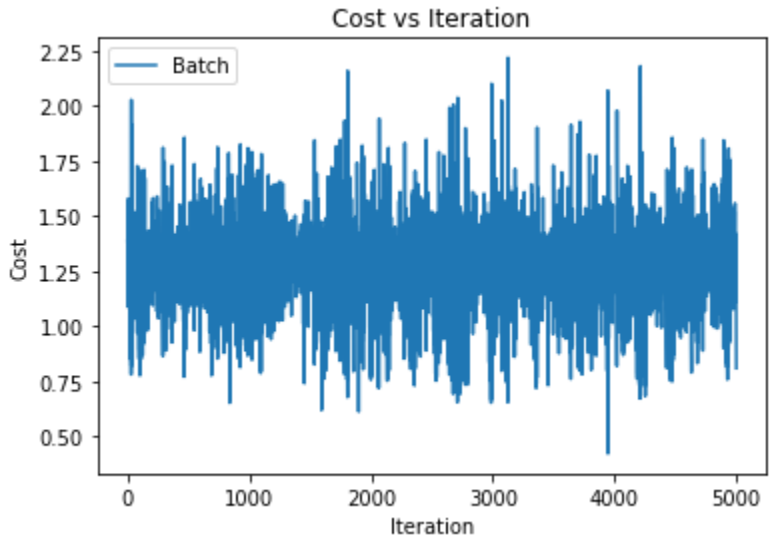
Batch



Mini Batch

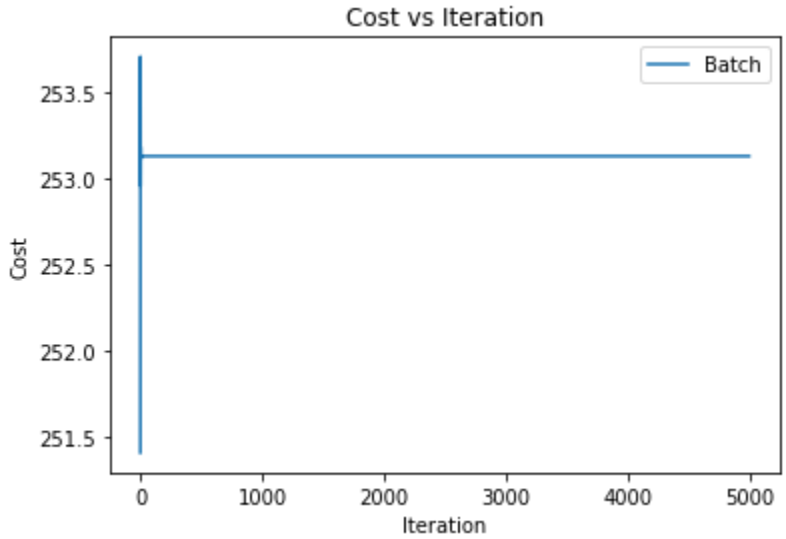


Stochastic

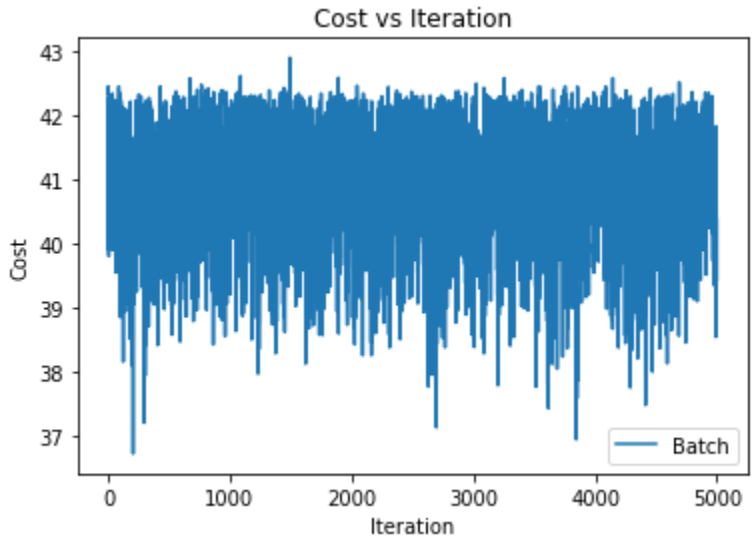


Learning Rate: 0.001

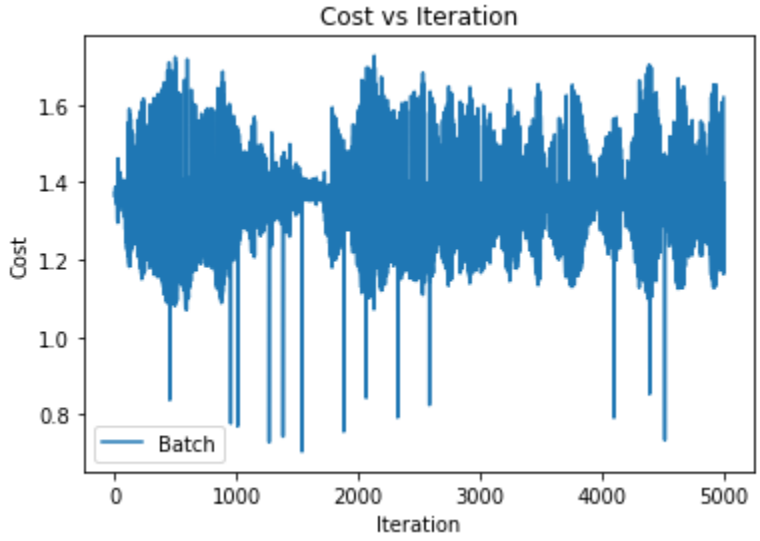
Batch



Mini Batch

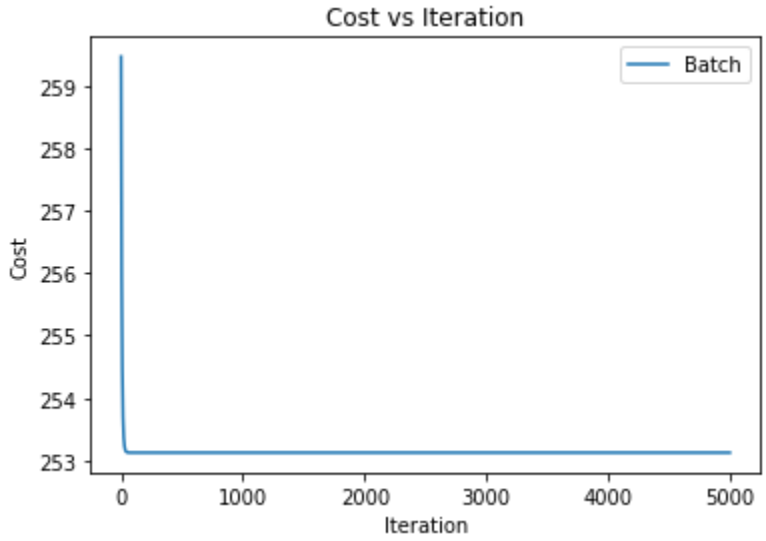


Stochastic

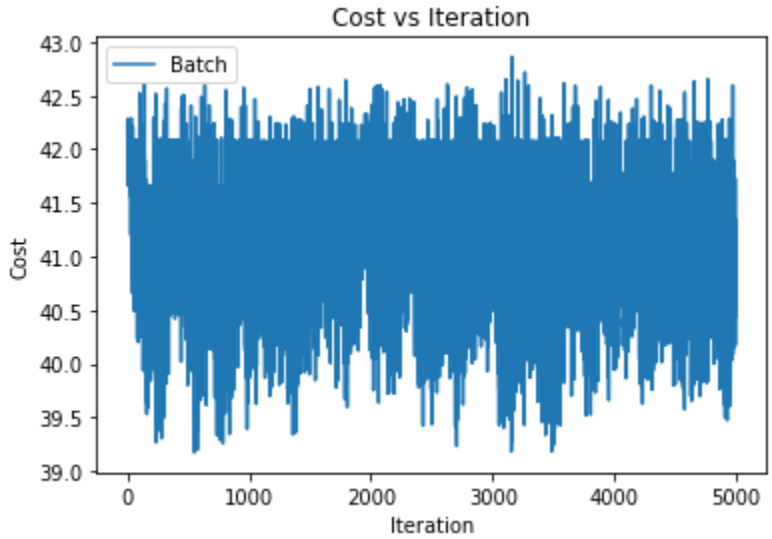


Learning Rate: 0.0001

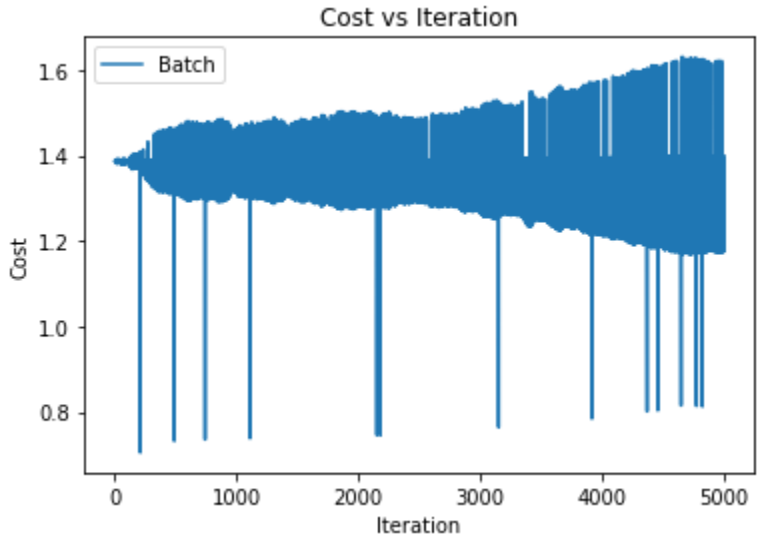
Batch



Mini Batch



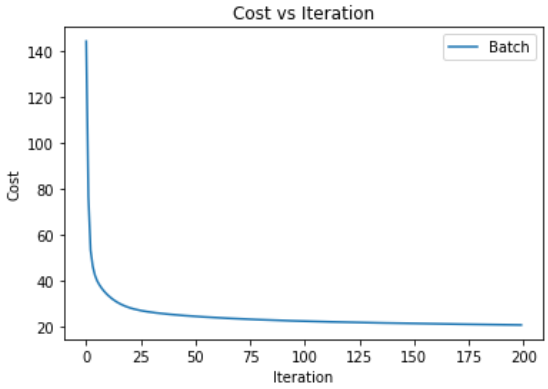
Stochastic



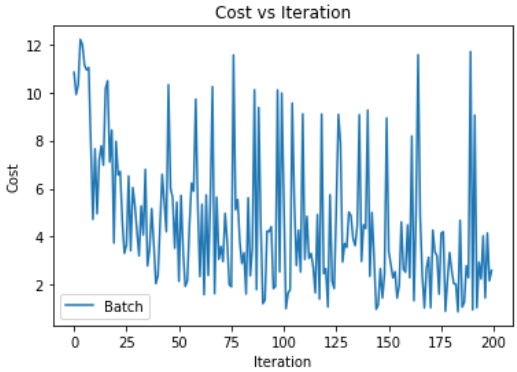
For the non normalized models in Logistic regression, the cost varies highly with an increase in the number of iterations. This is due to the data being unnormalized. The value of cost upon calculating reaches extremely high values without bound, thus causing a big change in cost after every iteration irrespective of having a low learning rate. The results will differ highly after normalization since that will give a bound to the range of values thus producing a better cost function.

Learning Rate: 0.01

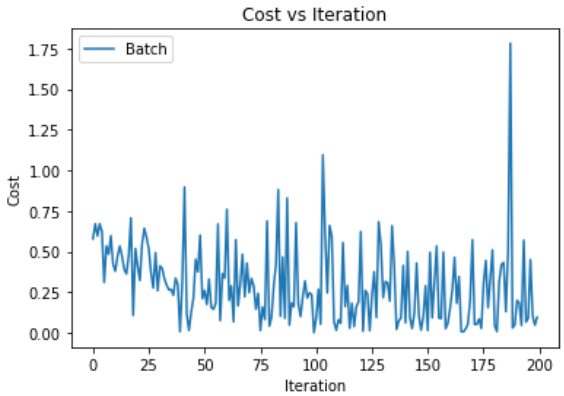
Batch



Mini Batch

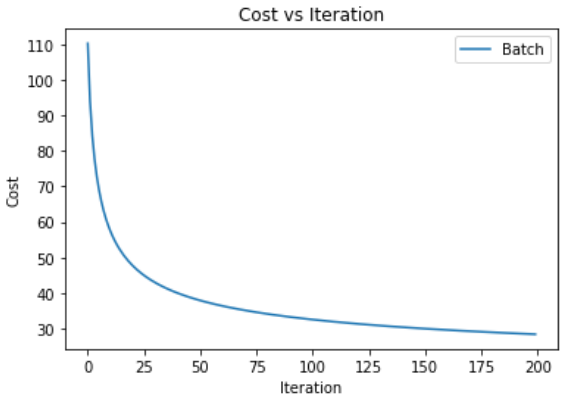


Stochastic

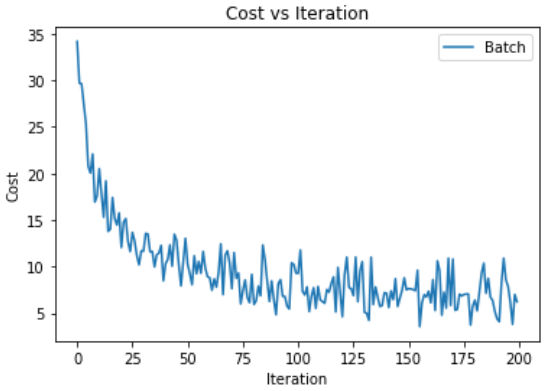


Learning Rate 0.001

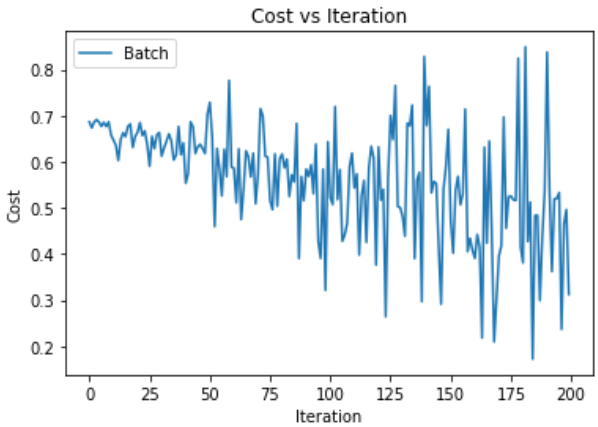
Batch



Mini Batch

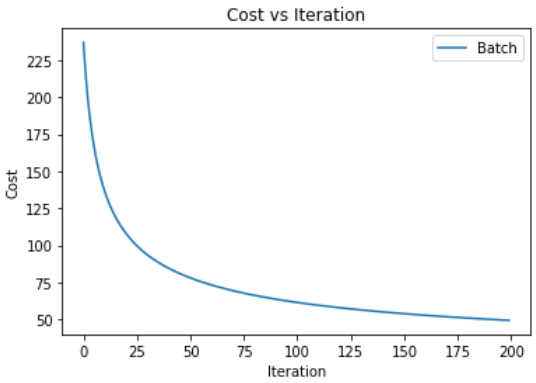


Stochastic

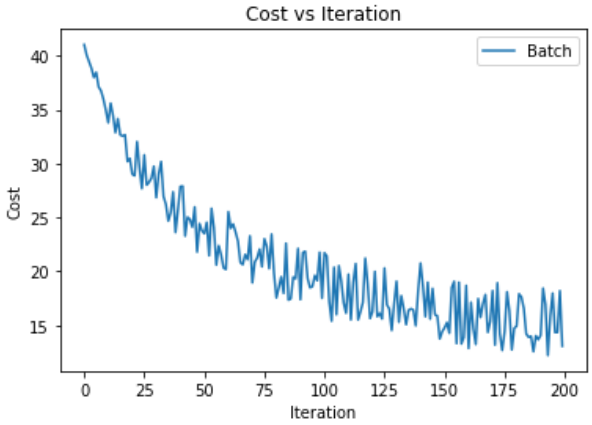


Learning rate 0.0001

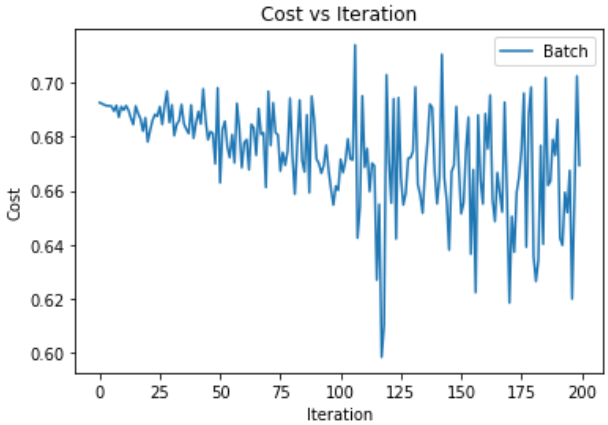
Batch



Mini Batch



Stochastic

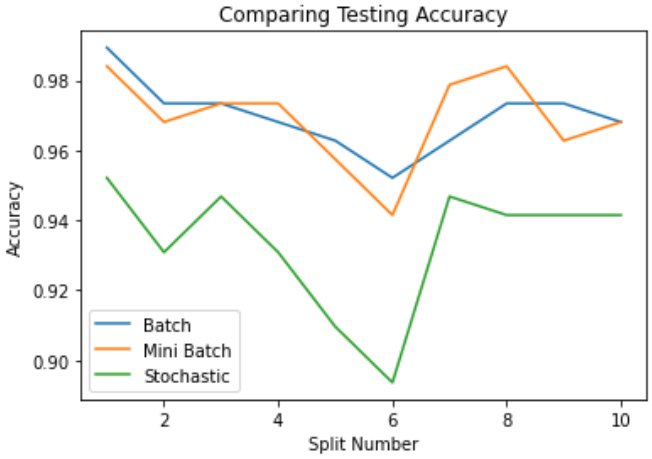


For the normalized data we can explain each of the graphs with explanation as it is clearly visible the error or cost decreases after every iteration. Consider the batch gradient descent example. Since we are working with the complete data in batch gradient descent, we can see that the cost function decreases after every iteration since all training examples are considered in this case. Eventually, after a set number of iterations we get to the “w” giving the lowest cost. This is the w used in our example for logistic regression. Similarly the error decreases for mini batch as well as stochastic gradient descent too although since we are picking a random sample each time, the cost even increases in some iterations.

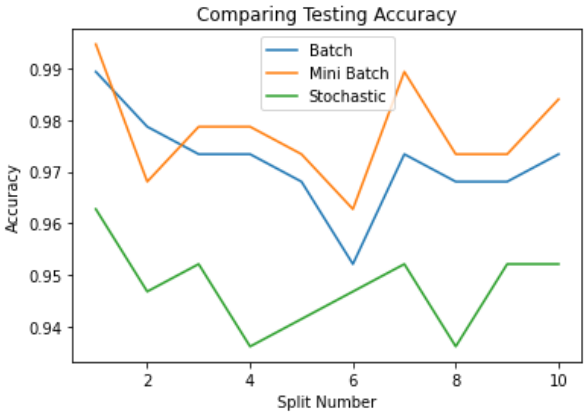
Outlining difference on changing threshold (Normalised)

For different values of threshold on same learning rate:

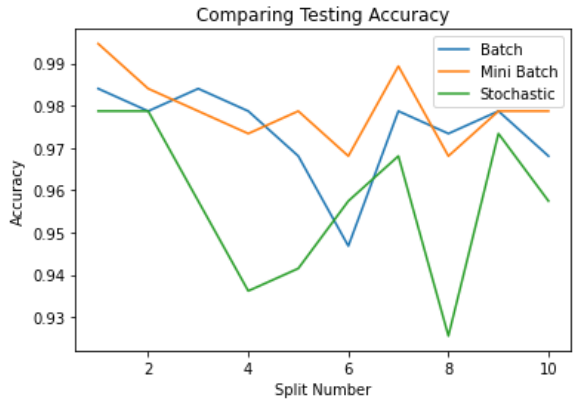
Threshold = 0.7



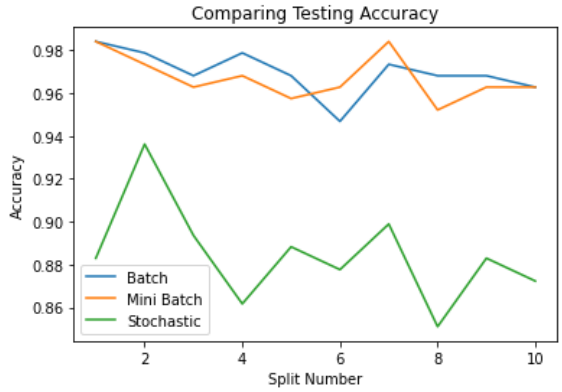
Threshold = 0.6



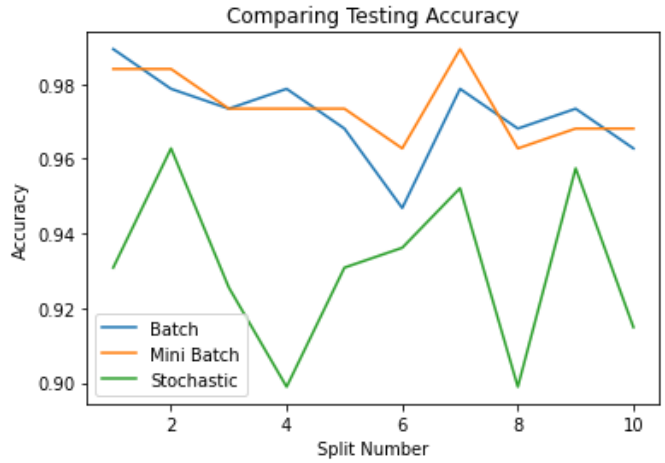
Threshold = 0.5



Threshold = 0.3



Threshold = 0.4



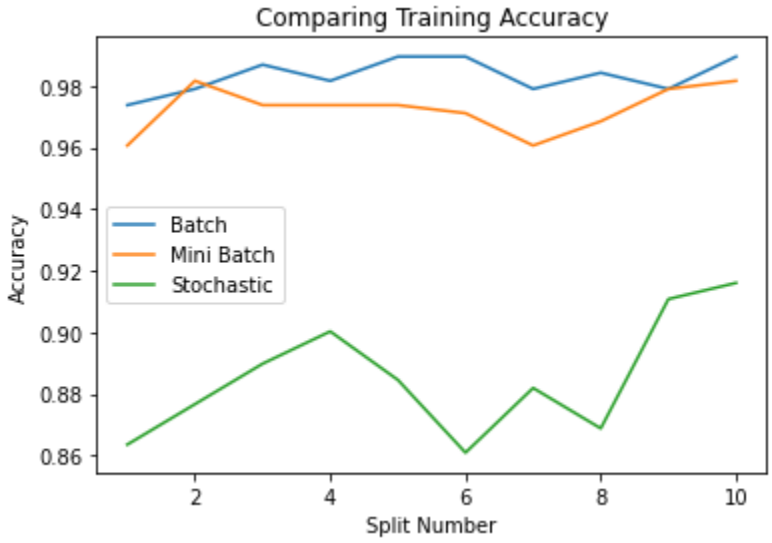
Comparing accuracy at different thresholds, we see that threshold at 0.5 and 0.6 yields good results as compared to 0.3 and 0.4. This indicates that the boundary is linear and around 55% Rather than exactly at the center. Thus there is around 45% probability of malignant class, since at 0.6 we get the highest accuracy.

Outlining difference on changing threshold (Not Normalised)

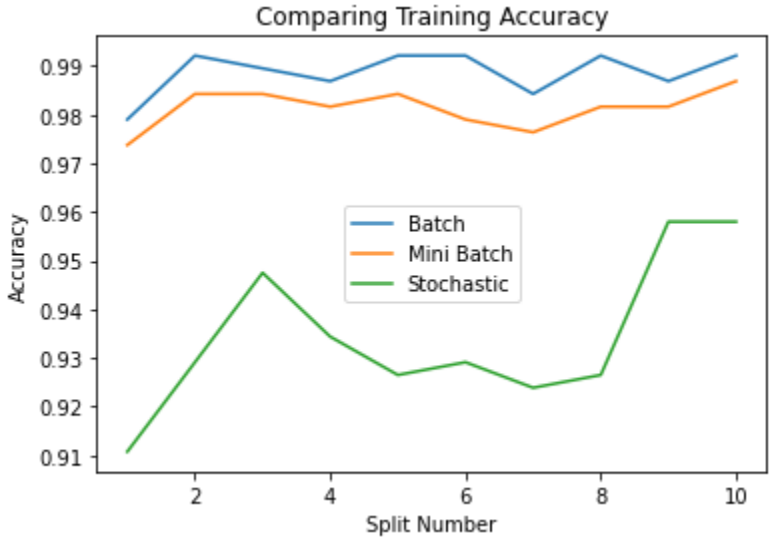
Threshold 0.7

Threshold 0.6

Threshold 0.3



Threshold 0.5



Threshold 0.4

Similar results are yielded here too. We get the best accuracy at threshold 0.6. Thus 60% probability of being in class B and 40% class M at this decision boundary.

Part D – Comparative Study:

$$\text{Accuracy} = \frac{(TP + TN)}{(TP + FP + TN + FN)}$$

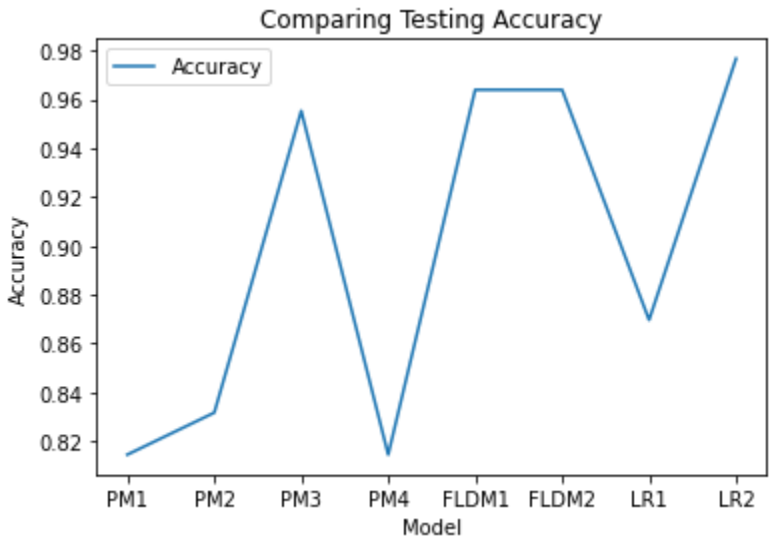
$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

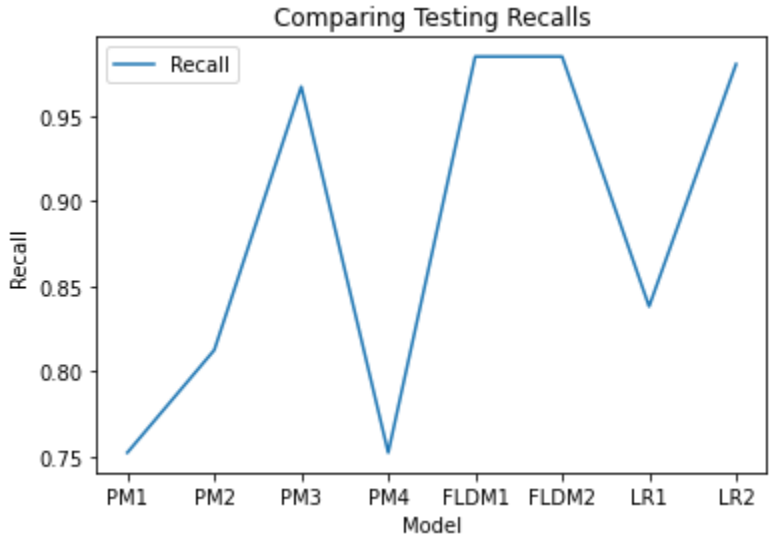
We can see that after calculating the average recalls and precisions in all the above cases and also plotting the accuracy for all models in the above photos, FLDM1 and FLDM2 have the same precision and recall values since we already discussed that their graphical plots are the same.
Also note, we get the highest precision and recall for these values as compared to all our other models. The models generated by FLDM are generative models, thus we have plotted the gaussian distributions in accordance with the provided dataset to find the intersection point or line of the two classes.
FLDM1 is performing better than most of the PM models since PM models are not linearly separable in cases where data is not normally distributed. Thus the model does not find the decision boundary accurately in the end giving lower recall values which suggest a high number of False negatives in the separation.

Serial Number	PM1	PM2	PM3	PM4	FLDM1	FLDM2	LR1	LR2
Average Accuracy	0.9322	0.9023	0.9544	0.9322	0.97117	0.97117	0.891	0.983
Average Precision	0.9333	0.8027	0.9621	0.9333	0.95742	0.95742	0.828	0.976
Average Recall	0.7520	0.8124	0.9674	0.7520	0.9851	0.9851	0.883	0.981

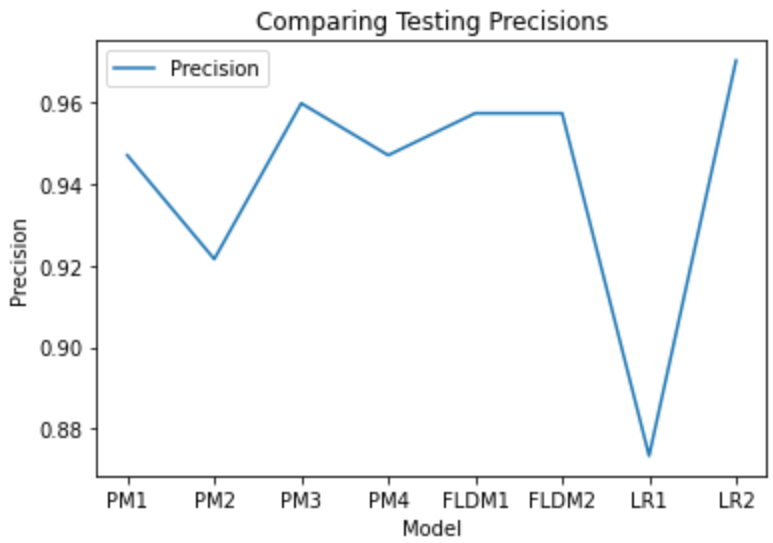
From the table seen above we can clearly see that the Logistic Regression Model provides the best results overall.



Even from the graph for accuracy we can see that Logstic Regression 2 is most accurate amongst all.



Here we can see recalls have high values for FLDM1 and 2 and also for LR2



Here we can see that LR2 is most precise.

Why so?

The reason for such result is that Logistic regression can handle non-linear relationships between features and target variable. In general generative models are worse than discriminative models as they are less sensitive to outliers. It also handles imbalenced datasets more effectively compared to others.