Classification Algorithms

Anirudh Singh, 2019B5A70948H
Bharath Variar, 2019B5A70930H

Assignment-1 Report
BITS F266: Machine Learning
Department of Computer Science and Information
BITS Pilani, Hyderabad Campus
March 2023

Contents

1 Perceptron Learning Algorithm

1.1 Data Cleaning

For the perceptron algorithm to work efficiently, we first remove the `id` column since it consists of large values that can disturb the learning process while not contributing to it semantically, as the id assigned to any sample does not determine the type of the tumour. We then map the target attribute, `diagnosis`, to numeric values using the map function. We assign value 1 to Malignant and -1 to Benign tumours.

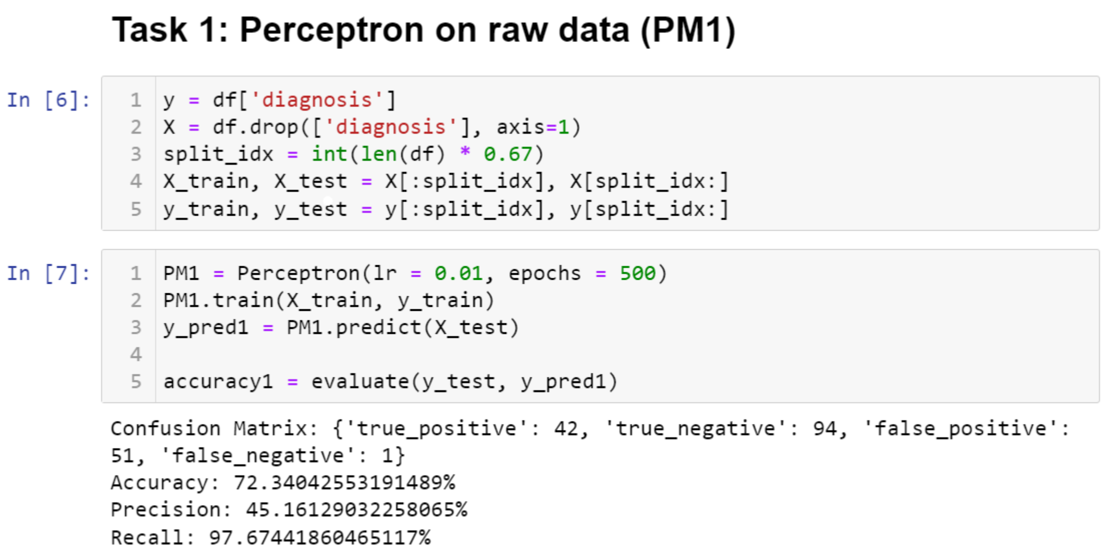
1.2 Perceptron Algorithm

The Perceptron algorithm is implemented using a `Perceptron` class which is implemented as follows:

```python
class Perceptron:
    def __init__(self, lr=0.001, epochs=10):
        self.lr = lr
        self.epochs = epochs
        self.weights = None

    def train(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)

        for _ in range(self.epochs):
            y_h=np.dot(X,self.weights)
            for idx, x_i in X.iterrows():
                y_hat = np.dot(x_i, self.weights)
                if y_hat*y[idx]<=0:
                    self.weights += x_i*y[idx]

    def predict(self, X):
        y_hat=np.dot(X, self.weights)
        return np.where(y_hat >= 0, 1, -1)
```

1.3 Task 1

In this task, we train the perceptron algorithm on the raw data (`PM1`) and on shuffled data (`PM2`). We can see that `PM1` gives an accuracy of 72.3% while s`PM2` gives an accuracy of 79.3%, suggesting that the data is linearly separable and shuffling the data improves the learning of the perceptron algorithm.



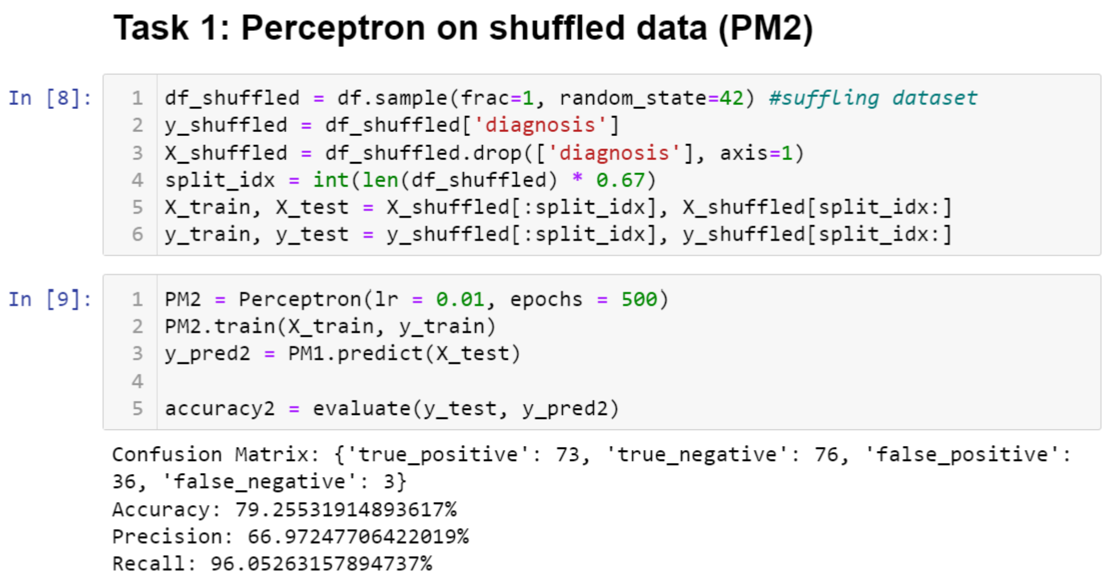
Task 1: Perceptron on raw data (PM1)

In [6]:
```python
y = df['diagnosis']
X = df.drop(['diagnosis'], axis=1)
split_idx = int(len(df) * 0.67)
X_train, X_test = X[:split_idx], X[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]
```

In [7]:
```python
PM1 = Perceptron(lr = 0.01, epochs = 500)
PM1.train(X_train, y_train)
y_pred1 = PM1.predict(X_test)

accuracy1 = evaluate(y_test, y_pred1)
```

```
Confusion Matrix: {'true_positive': 42, 'true_negative': 94, 'false_positive':
51, 'false_negative': 1}
Accuracy: 72.34042553191489%
Precision: 45.16129032258065%
Recall: 97.67441860465117%
```

Figure 1: PM1



Task 1: Perceptron on shuffled data (PM2)

In [8]:
```python
df_shuffled = df.sample(frac=1, random_state=42) #suffling dataset
y_shuffled = df_shuffled['diagnosis']
X_shuffled = df_shuffled.drop(['diagnosis'], axis=1)
split_idx = int(len(df_shuffled) * 0.67)
X_train, X_test = X_shuffled[:split_idx], X_shuffled[split_idx:]
y_train, y_test = y_shuffled[:split_idx], y_shuffled[split_idx:]
```

In [9]:
```python
PM2 = Perceptron(lr = 0.01, epochs = 500)
PM2.train(X_train, y_train)
y_pred2 = PM1.predict(X_test)

accuracy2 = evaluate(y_test, y_pred2)
```

```
Confusion Matrix: {'true_positive': 73, 'true_negative': 76, 'false_positive':
36, 'false_negative': 3}
Accuracy: 79.25531914893617%
Precision: 66.97247706422019%
Recall: 96.05263157894737%
```

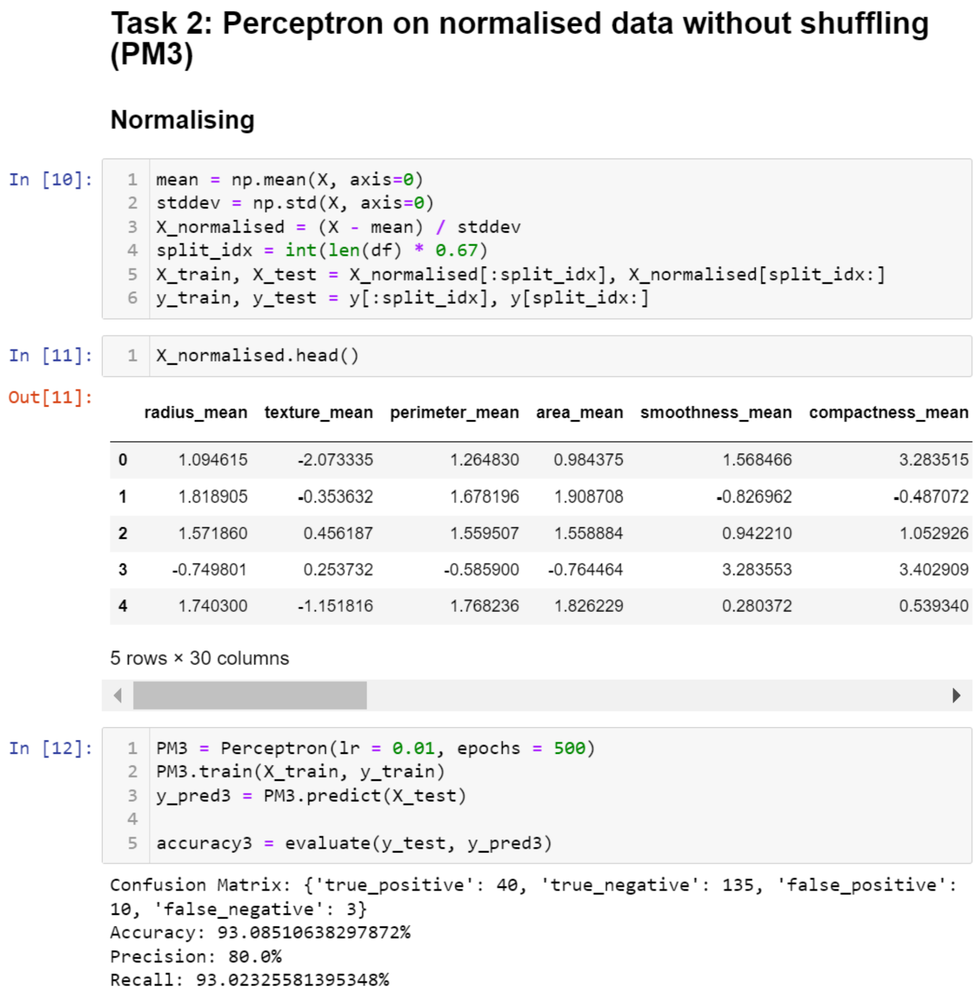
Figure 2: PM2

1.4 Task 2

In this task, we attempt to improve the learning by normalising the data based on the formula:

$$X = \frac{X - \mu}{\sigma} \tag{1}$$

where μ, σ are the mean and standard deviation of X respectively.
We call the model that trains on normalised data `PM3`, and it classifies with an accuracy of 93.1% as shown below. Comparing this model with `PM1`, we see that normalising the data greatly improves the accuracy of classification, and this is because the change in weights in each iteration is not large since the data is small due to normalisation.



Task 2: Perceptron on normalised data without shuffling (PM3)

Normalising

In [10]:
```
mean = np.mean(X, axis=0)
stddev = np.std(X, axis=0)
X_normalised = (X - mean) / stddev
split_idx = int(len(df) * 0.67)
X_train, X_test = X_normalised[:split_idx], X_normalised[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]
```

In [11]:
```
X_normalised.head()
```

Out[11]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
0	1.094615	-2.073335	1.264830	0.984375	1.568466	3.283515
1	1.818905	-0.353632	1.678196	1.908708	-0.826962	-0.487072
2	1.571860	0.456187	1.559507	1.558884	0.942210	1.052926
3	-0.749801	0.253732	-0.585900	-0.764464	3.283553	3.402909
4	1.740300	-1.151816	1.768236	1.826229	0.280372	0.539340

5 rows × 30 columns

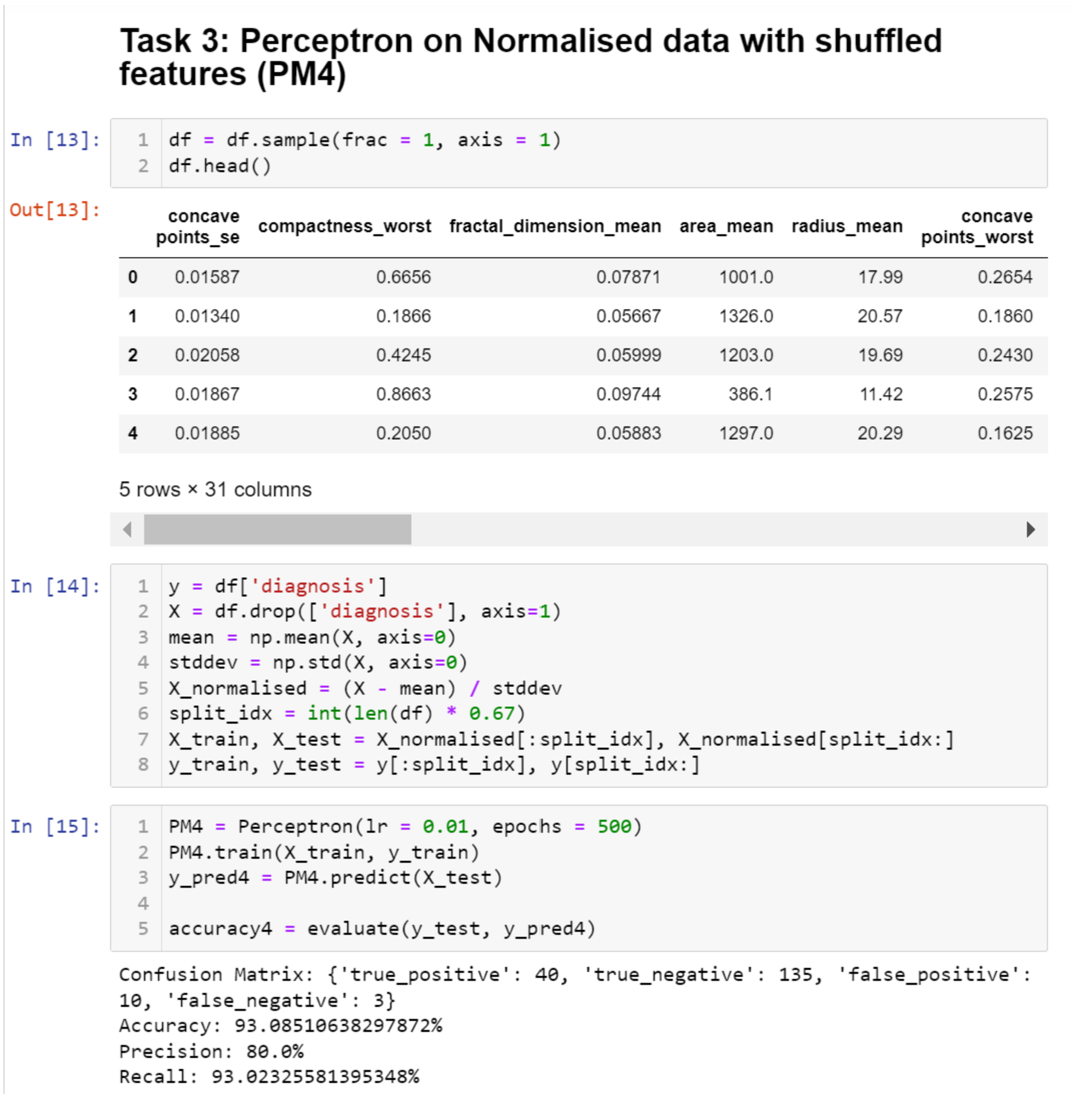
In [12]:
```
PM3 = Perceptron(lr = 0.01, epochs = 500)
PM3.train(X_train, y_train)
y_pred3 = PM3.predict(X_test)

accuracy3 = evaluate(y_test, y_pred3)
```

```
Confusion Matrix: {'true_positive': 40, 'true_negative': 135, 'false_positive':
10, 'false_negative': 3}
Accuracy: 93.08510638297872%
Precision: 80.0%
Recall: 93.02325581395348%
```

Figure 3: PM3

1.5 Task 3

In the final task for the perceptron algorithm, we train the model, `PM4` on normalised data with the features shuffled. `PM4` classifies with an accuracy of 93.1%, exactly the same as `PM3`, which implies that while shuffling data improves accuracy, shuffling the feature tuple does not. This still performs much better than `PM1`, but this is due to normalisation and not the shuffled feature tuple, since the accuracies of both `PM3` and `PM4` are identical.



Task 3: Perceptron on Normalised data with shuffled features (PM4)

In [13]:
```
df = df.sample(frac = 1, axis = 1)
df.head()
```

Out[13]:

	concave points_se	compactness_worst	fractal_dimension_mean	area_mean	radius_mean	concave points_worst
0	0.01587	0.6656	0.07871	1001.0	17.99	0.2654
1	0.01340	0.1866	0.05667	1326.0	20.57	0.1860
2	0.02058	0.4245	0.05999	1203.0	19.69	0.2430
3	0.01867	0.8663	0.09744	386.1	11.42	0.2575
4	0.01885	0.2050	0.05883	1297.0	20.29	0.1625

5 rows × 31 columns

In [14]:
```
y = df['diagnosis']
X = df.drop(['diagnosis'], axis=1)
mean = np.mean(X, axis=0)
stddev = np.std(X, axis=0)
X_normalised = (X - mean) / stddev
split_idx = int(len(df) * 0.67)
X_train, X_test = X_normalised[:split_idx], X_normalised[split_idx:]
y_train, y_test = y[:split_idx], y[split_idx:]
```

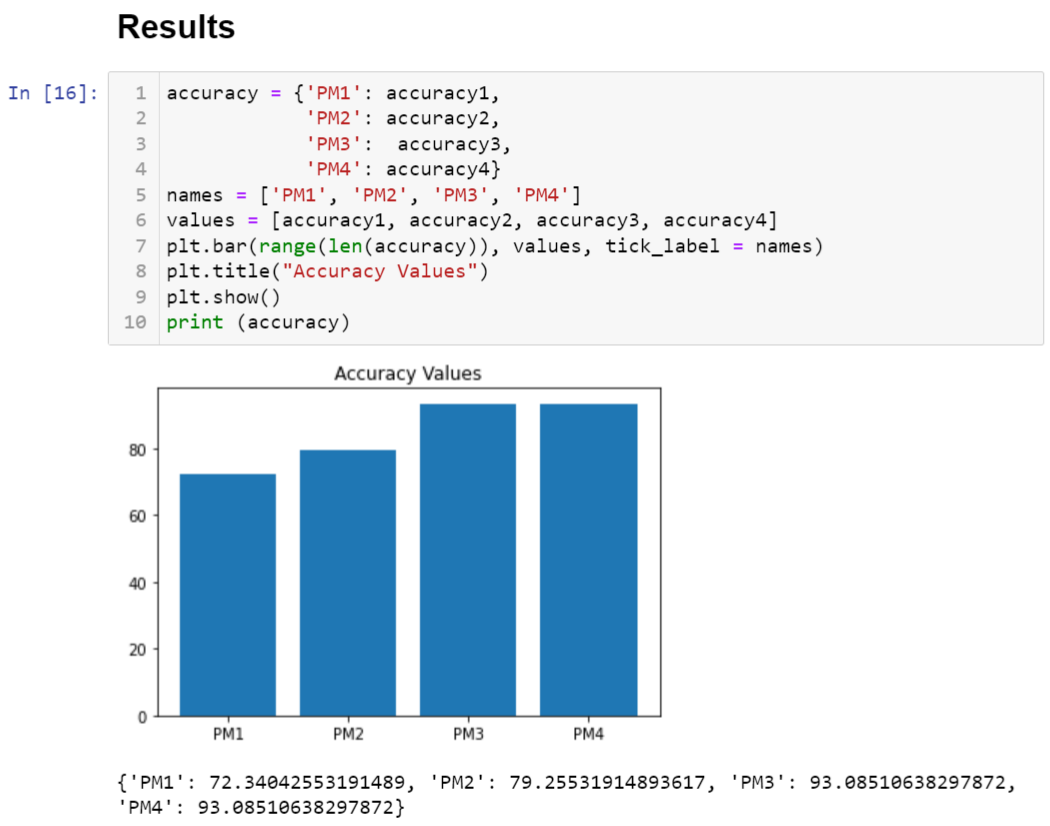
In [15]:
```
PM4 = Perceptron(lr = 0.01, epochs = 500)
PM4.train(X_train, y_train)
y_pred4 = PM4.predict(X_test)

accuracy4 = evaluate(y_test, y_pred4)
```

```
Confusion Matrix: {'true_positive': 40, 'true_negative': 135, 'false_positive':
10, 'false_negative': 3}
Accuracy: 93.08510638297872%
Precision: 80.0%
Recall: 93.02325581395348%
```

Figure 4: PM4

1.6 Results

We finally plot the accuracies of all the models in a bar chart for a comparative study as shown:



Results

In [16]:
```python
accuracy = {'PM1': accuracy1,
            'PM2': accuracy2,
            'PM3':  accuracy3,
            'PM4': accuracy4}
names = ['PM1', 'PM2', 'PM3', 'PM4']
values = [accuracy1, accuracy2, accuracy3, accuracy4]
plt.bar(range(len(accuracy)), values, tick_label = names)
plt.title("Accuracy Values")
plt.show()
print (accuracy)
```

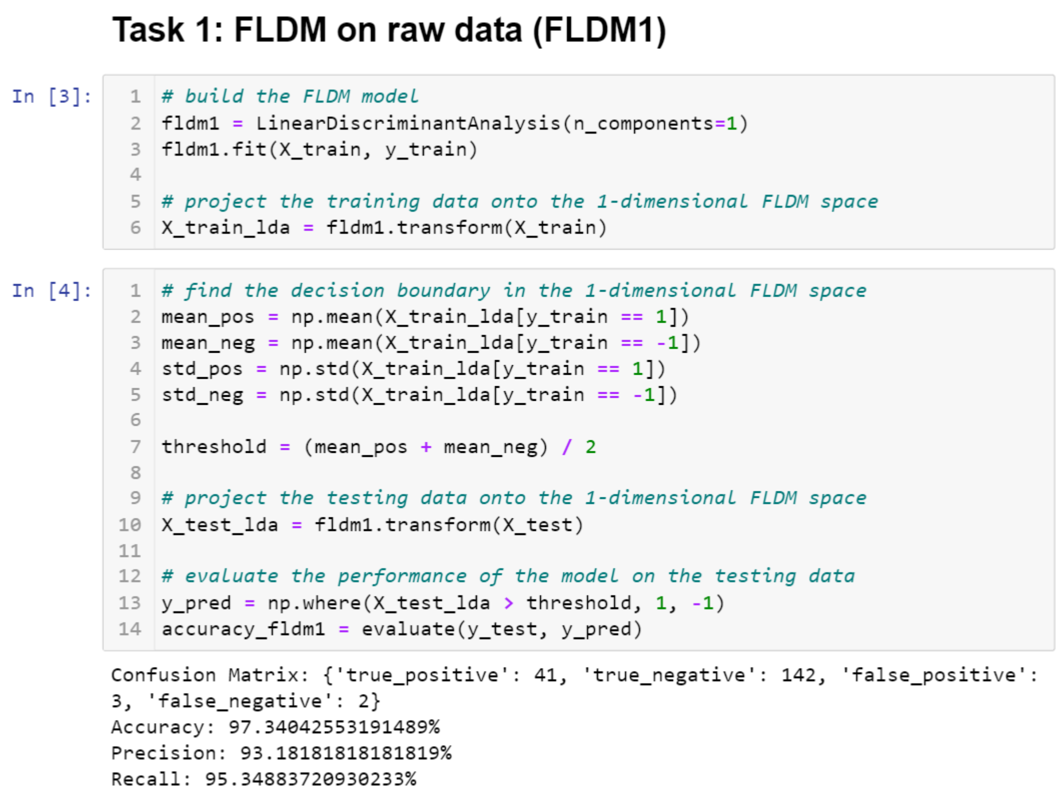
```
{'PM1': 72.34042553191489, 'PM2': 79.25531914893617, 'PM3': 93.08510638297872,
'PM4': 93.08510638297872}
```

Figure 5: Results

2 Fisher's Linear Discriminant Analysis

2.1 Task 1

We use the linear discriminant model from `scikitlearn` and call it `FLDM1`. Implementation is done as shown below:


Task 1: FLDM on raw data (FLDM1)

In [3]:
```
# build the FLDM model
fldm1 = LinearDiscriminantAnalysis(n_components=1)
fldm1.fit(X_train, y_train)

# project the training data onto the 1-dimensional FLDM space
X_train_lda = fldm1.transform(X_train)
```

In [4]:
```
# find the decision boundary in the 1-dimensional FLDM space
mean_pos = np.mean(X_train_lda[y_train == 1])
mean_neg = np.mean(X_train_lda[y_train == -1])
std_pos = np.std(X_train_lda[y_train == 1])
std_neg = np.std(X_train_lda[y_train == -1])

threshold = (mean_pos + mean_neg) / 2

# project the testing data onto the 1-dimensional FLDM space
X_test_lda = fldm1.transform(X_test)

# evaluate the performance of the model on the testing data
y_pred = np.where(X_test_lda > threshold, 1, -1)
accuracy_fldm1 = evaluate(y_test, y_pred)
```

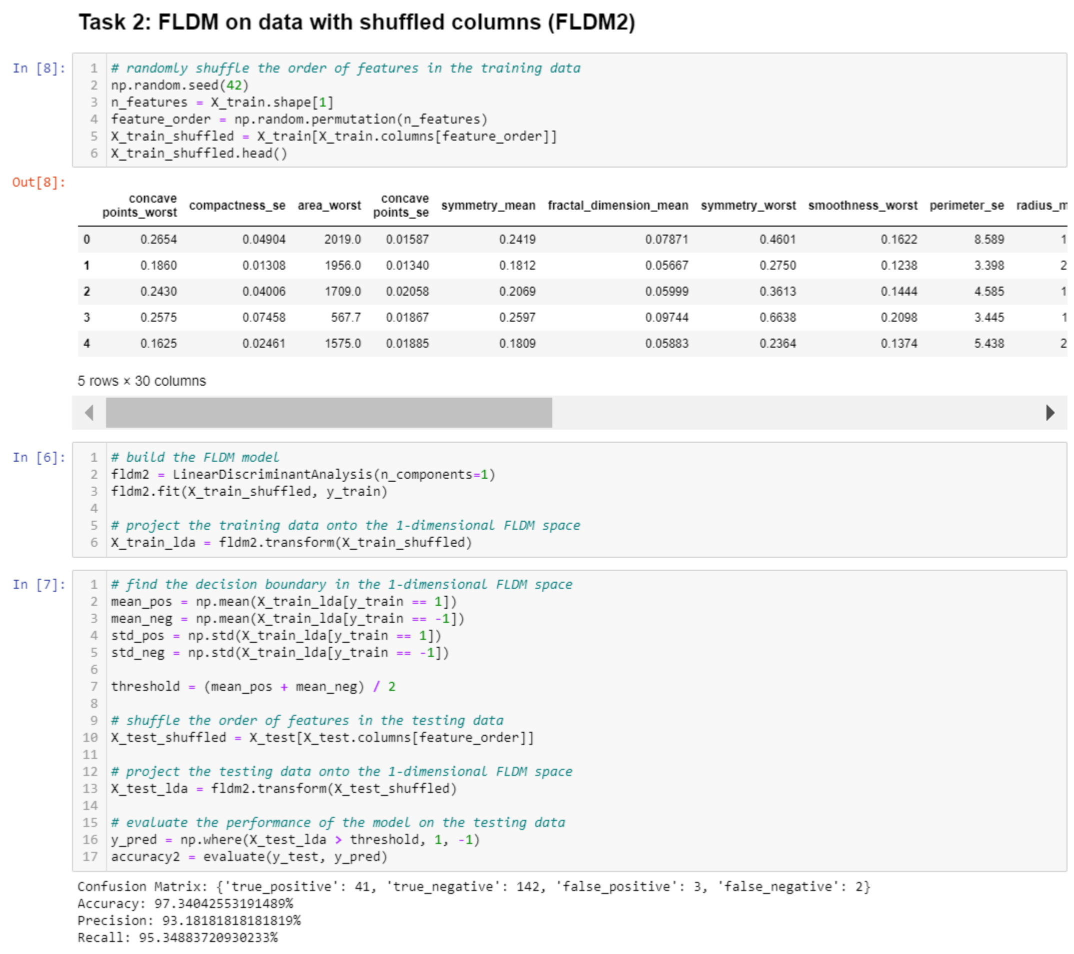
```
Confusion Matrix: {'true_positive': 41, 'true_negative': 142, 'false_positive':
3, 'false_negative': 2}
Accuracy: 97.34042553191489%
Precision: 93.18181818181819%
Recall: 95.34883720930233%
```

Figure 6: FLDM1

We obtain an accuracy of 97.3%.

2.2 Task 2

In this task, we shuffle the feature tuple and then call this model `FLDM2`. The implementation is shown below. As we can see, the accuracy is 97.3%, identical to that produced by `FLDM1`, and this implies that shuffling the features does not change the learning for this algorithm.



Task 2: FLDM on data with shuffled columns (FLDM2)

In [8]:
```
# randomly shuffle the order of features in the training data
np.random.seed(42)
n_features = X_train.shape[1]
feature_order = np.random.permutation(n_features)
X_train_shuffled = X_train[X_train.columns[feature_order]]
X_train_shuffled.head()
```

Out[8]:

	concave points_worst	compactness_se	area_worst	concave points_se	symmetry_mean	fractal_dimension_mean	symmetry_worst	smoothness_worst	perimeter_se	radius_m
0	0.2654	0.04904	2019.0	0.01587	0.2419	0.07871	0.4601	0.1622	8.589	1
1	0.1860	0.01308	1956.0	0.01340	0.1812	0.05667	0.2750	0.1238	3.398	2
2	0.2430	0.04006	1709.0	0.02058	0.2069	0.05999	0.3613	0.1444	4.585	1
3	0.2575	0.07458	567.7	0.01867	0.2597	0.09744	0.6638	0.2098	3.445	1
4	0.1625	0.02461	1575.0	0.01885	0.1809	0.05883	0.2364	0.1374	5.438	2

5 rows × 30 columns

In [6]:
```
# build the FLDM model
fldm2 = LinearDiscriminantAnalysis(n_components=1)
fldm2.fit(X_train_shuffled, y_train)

# project the training data onto the 1-dimensional FLDM space
X_train_lda = fldm2.transform(X_train_shuffled)
```

In [7]:
```
# find the decision boundary in the 1-dimensional FLDM space
mean_pos = np.mean(X_train_lda[y_train == 1])
mean_neg = np.mean(X_train_lda[y_train == -1])
std_pos = np.std(X_train_lda[y_train == 1])
std_neg = np.std(X_train_lda[y_train == -1])

threshold = (mean_pos + mean_neg) / 2

# shuffle the order of features in the testing data
X_test_shuffled = X_test[X_test.columns[feature_order]]

# project the testing data onto the 1-dimensional FLDM space
X_test_lda = fldm2.transform(X_test_shuffled)

# evaluate the performance of the model on the testing data
y_pred = np.where(X_test_lda > threshold, 1, -1)
accuracy2 = evaluate(y_test, y_pred)
```

```
Confusion Matrix: {'true_positive': 41, 'true_negative': 142, 'false_positive': 3, 'false_negative': 2}
Accuracy: 97.34042553191489%
Precision: 93.18181818181819%
Recall: 95.34883720930233%
```

Figure 7: FLDM2

3 Logistic Regression

3.1 Implementation

We have implemented vanilla, stochastic and minibatch gradient descent in the object-oriented paradigm by defining various classes as shown below:

3.1.1 Vanilla Gradient Descent

LogisiticRegressionGD as follows:

```
class LogisticRegressionGD:
```

```python
def __init__(self, learning_rate=0.01, n_iters=1000, random_state=None,threshold=0.5):
        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.random_state = random_state
        self.threshold=threshold

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
        self.costs = []

        # set random seed for reproducibility
        if self.random_state is not None:
            np.random.seed(self.random_state)

        # gradient descent
        for i in range(self.n_iters):
            # calculate predicted probabilities and gradients
            linear_model = np.dot(X, self.weights) + self.bias
            y_pred = self.sigmoid(linear_model)
            dw = np.dot(X.T, (y_pred - y)) / n_samples
            db = np.sum(y_pred - y) / n_samples

            # update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

            # calculate cost and add to list for graphing
            y_pred = self.sigmoid(np.dot(X, self.weights) + self.bias)
            y_pred[y_pred == 0] = 1e-15  # add small constant value to avoid NaN in cost
            y_pred[y_pred == 1] = 1 - 1e-15  # add small constant value to avoid NaN in cost
            cost = -1/n_samples * np.sum(y * np.log(y_pred) + (1-y) * np.log(1-y_pred))
            self.costs.append(cost)

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = self.sigmoid(linear_model)
        y_pred_class = [1 if i > self.threshold else 0 for i in y_pred]
        return y_pred_class

    def plot_cost(self):
        fig, ax = plt.subplots(figsize=(10, 8))
        plt.plot(np.arange(1, len(self.costs)+1), self.costs)
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.title('Gradient Descent Cost Graph')
        plt.show()
        fig.savefig('unormalizedLR_graphs/'+"GD"+str(len(str(self.
    learning_rate)))+str(self.threshold)[-1]+".png")
```

3.1.2 Stochastic Gradient Descent

```python
class LogisticRegressionSGD:
    def __init__(self, learning_rate=0.01, n_iters=1000, batch_size
    =1, random_state=None,threshold=0.5):
        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.batch_size = batch_size
        self.random_state = random_state
        self.threshold=threshold

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
        self.costs = []

        # set random seed for reproducibility
        if self.random_state is not None:
            np.random.seed(self.random_state)

        # stochastic gradient descent
        for i in range(self.n_iters):
            # shuffle data
            idx = np.arange(n_samples)
            np.random.shuffle(idx)
            X_shuffled = X[idx]
            y_shuffled = y[idx]

            # loop over batches
            for j in range(0, n_samples, self.batch_size):
                # get mini-batch
                X_batch = X_shuffled[j:j+self.batch_size]
                y_batch = y_shuffled[j:j+self.batch_size]

                # calculate predicted probabilities and gradients
                linear_model = np.dot(X_batch, self.weights) + self.bias
                y_pred = self.sigmoid(linear_model)
                dw = np.dot(X_batch.T, (y_pred - y_batch)) / self.batch_size
                db = np.sum(y_pred - y_batch) / self.batch_size

                # update weights and bias
                self.weights -= self.learning_rate * dw
                self.bias -= self.learning_rate * db

            # calculate cost and add to list for graphing
            y_pred = self.sigmoid(np.dot(X, self.weights) + self.bias)
            y_pred[y_pred == 0] = 1e-15  # add small constant value to avoid NaN in cost
            y_pred[y_pred == 1] = 1 - 1e-15  # add small constant value to avoid NaN in cost
            cost = -1/n_samples * np.sum(y * np.log(y_pred) + (1-y) * np.log(1-y_pred))
            self.costs.append(cost)

    def predict_proba(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_pred = self.sigmoid(linear_model)
        return y_pred

    def predict(self, X):
        y_pred_proba = self.predict_proba(X)
        y_pred_class = [1 if i > self.threshold else 0 for i in y_pred_proba]
        return y_pred_class

    def plot_cost(self):
        fig, ax = plt.subplots(figsize=(10, 8))
        plt.plot(np.arange(1, len(self.costs)+1), self.costs)
        plt.xlabel('Iterations')
        plt.ylabel('Cost')
        plt.title('Stochastic Gradient Descent Cost Graph')
        plt.show()
        fig.savefig('unormalizedLR_graphs/'+"SGD"+str(len(str(self.learning_rate)))+str(self.threshold)[-1]+".png")
```

3.1.3 MiniBatch Gradient Descent

```python
class LogisticRegressionMiniBatchGD:
    def __init__(self, learning_rate=0.01, n_iters=1000, batch_size=32, random_state=None,threshold=0.5):
        self.learning_rate = learning_rate
        self.n_iters = n_iters
        self.batch_size = batch_size
        self.random_state = random_state
        self.threshold=threshold

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

    def fit(self, X, y):
        # initialize weights and bias to zero
        self.weights = np.zeros(X.shape[1])
        self.bias = 0

        # initialize costs list for storing costs at each iteration
        self.costs = []

        # set random seed for reproducibility
        n_samples, n_features = X.shape
        if self.random_state is not None:
            np.random.seed(self.random_state)

        # minibatch gradient descent
        for i in range(self.n_iters):
            # shuffle data
            idx = np.arange(X.shape[0])
            np.random.shuffle(idx)
            X = X[idx]
            y = y[idx]

            # loop over batches
            for j in range(0, X.shape[0], self.batch_size):
                # get minibatch
                X_batch = X[j:j+self.batch_size]
                y_batch = y[j:j+self.batch_size]

                # calculate predicted probabilities and gradients
                y_pred = self.sigmoid(np.dot(X_batch, self.weights) + self.bias)
                dw = np.dot(X_batch.T, (y_pred - y_batch)) / self.batch_size
                db = np.sum(y_pred - y_batch) / self.batch_size

```

```python
             # update weights and bias
             self.weights -= self.learning_rate * dw
             self.bias -= self.learning_rate * db

         # calculate cost and add to list for graphing
         y_pred = self.sigmoid(np.dot(X, self.weights) + self.bias)
         y_pred[y_pred == 0] = 1e-15  # add small constant value to avoid NaN in cost
         y_pred[y_pred == 1] = 1 - 1e-15  # add small constant value to avoid NaN in cost
         cost = -1/n_samples * np.sum(y * np.log(y_pred) + (1-y) * np.log(1-y_pred))
         self.costs.append(cost)

     return self

 def predict_proba(self, X):
     return self.sigmoid(np.dot(X, self.weights) + self.bias)

 def predict(self, X):
     return np.where(self.predict_proba(X) > self.threshold, 1, 0)

 def plot_cost(self):
     fig, ax = plt.subplots(figsize=(10, 8))
     plt.plot(range(1, len(self.costs) + 1), self.costs)
     plt.xlabel('Iteration')
     plt.ylabel('Cost')
     plt.title('Logistic Regression Cost Graph')
     plt.show()
     fig.savefig('unormalizedLR_graphs/'+"minibatchGD"+str(len(str(self.learning_rate)))+str(self.threshold)[-1]+".png")
```

3.2 Task 1: Raw Data

Varying thresholds did not change the accuracies much, but we got the best accuracy of 92.6% when the threshold was 0.5 and the learning rate of 0.001 using vanilla gradient descent, while we got the worst accuracy of 32.9% when the threshold was 0.6 and the learning rate 0.01.

3.3 Task 2: Normalised Data

In this task, we run the same algorithms after normalising data using the code shown below:

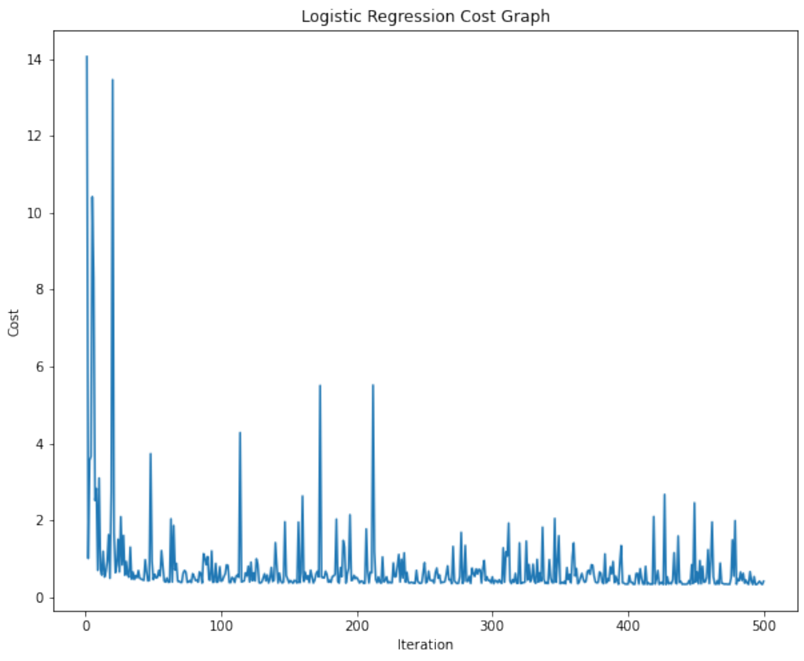


Figure 8: Mini-batch GD on raw Data

```
    y = df["diagnosis"]
    X = df.drop(["diagnosis"], axis=1)
    mean = np.mean(X, axis=0)
    stddev = np.std(X, axis=0)
    X = (X - mean) / stddev
    df = pd.concat([X, y], axis=1)
```

After normalising, while accuracy remained relatively small, from 92.6% to 98.9%, we noticed that the graphs for cost curves were significantly more stable, and this is because there are no large changes in weights in each iteration.
We obtained the highest accuracy of 98.9% with a learning rate of 0.001 and a threshold of 0.5 using the minibatch gradient descent. While the worst was an accuracy of 96.2% with a learning rate, of 0.01, and the threshold of 0.4 using stochastic gradient descent.

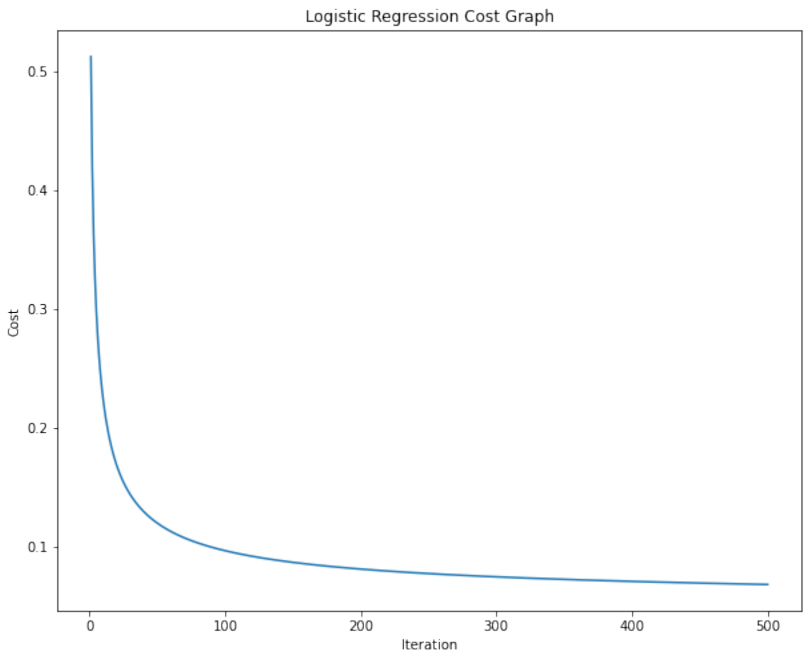


Figure 9: Mini-batch GD on normalised Data

Note: It, however, should be noted that with extremely low learning rates (0.0001), the gradient descent does not converge to the global minima in $>$ 500 iterations.

References

[1] GitHub Repository