In [ ]:
```
from sklearn.datasets import load_breast_cancer
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import sklearn
import sklearn.datasets
import numpy as np
import pandas as pd
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

%matplotlib inline
```

COGS 118A - Assignment 2: Perceptron & Logistic Regression (30 points)

The goal of this assignment is to practice and learn how to use Python to build a perceptron classifer and a logistic regression classifer for a given binary classification task.

Part 1 Perceptron

In this part of the assignment, we will be building a perceptron class.

Introduction:

W and b are the weight(s) and the bias for the perceptron classifier. W is an n-dimensional vector where n is the number of features the dataset has and the bias b is a scaler. The process of fitting is to optimize them using gradient descent. To predict whether the label y_i (our target) is 1 or -1, we calculate $sign(wx_i + b)$. The sign function we are using here is:

$$sign(x) = \begin{cases} 1 & x > 0 \\ -1 & x \leq 0 \end{cases}$$

In other words, if the sample x_i projected by w plus the bias b is larger than 0, we predict y_i to be 1, otherwise, we predict y_i to be -1.

Hence, the decision boundary of the perceptron classifier is:

$$Wx + b = 0$$

A decisioin boundary is a hyperplane (or a line if data is 2d) where samples that are classifed as one class lay on one side and the samples on the other side are classified as the other class.

The error is zero if our prediction matches the target, and it is 1 if our prediction is different from the target. Since our target can be either -1 or 1, we can formulate our error function as:

$$\epsilon_i = \frac{1}{2}(y_i - sign(wx_i + b))^2 = \frac{1}{2}(y_i - sign(\sum_j \{w_j x_{ij}\} + b))^2$$

, where j is the jth feature.

In each epoch, we loop through the entire training dataset. For each sample, we make a prediction using our current w and b. We update our weights and bias only when the prediction is incorrect.

The gradient of our error function is used to update our weights and the bias. The gradient of ϵ_i:

$$\frac{\partial \epsilon_i}{\partial w} = \begin{cases} x_i & \text{predicted} = 1 \ \& \text{ target} = \text{-}1 \\ -x_i & \text{predicted} = \text{-}1 \ \& \text{ target} = 1 \end{cases}$$

$$\frac{\partial \epsilon_i}{\partial b} = \begin{cases} y_i & \text{predicted} = 1 \ \& \text{ target} = \text{-}1 \\ -y_i & \text{predicted} = \text{-}1 \ \& \text{ target} = 1 \end{cases}$$

Another way of writing the derivatives is :

$$\frac{\partial \epsilon_i}{\partial w} = -y_i \cdot x_i$$

$$\frac{\partial \epsilon_i}{\partial b} = -y_i$$

For each incorrectly predicted sample, we update w and b :

$$w = w - \alpha \frac{\partial \epsilon_i}{\partial w}$$

$$b = b - \alpha \frac{\partial \epsilon_i}{\partial b}$$

,where α is the learning rate.

Notice that since the derivative is negative, this means you will be adding either $y_i \cdot x_i$ or y_i (times the learning rate) .

The following are the necessary components of the perceptron class:

1. Instance attribute `self.w` and `self.b` are the weights and the bias term. We randomly initialize them between -1 and 1. (ex. 0.452)

2. Instance attribute `self.learning_rate` is $\alpha, \alpha \in (0, 1]$.

3. Instance attribute `self.epoch` is the number of times we want to loop through the entire training dataset.

4. Instance attribute `self.x` stores the training data. Instance attribute `self.y` stores the training labels.

5. Instance attribute `self.x_test` stores the test data. Instance attribute `self.y_test` stores the test labels.

6. Instance attribute `self.train_accuracies` stores the training accuracy after each epoch.

7. Instance attribute `self.test_accuracies` stores the test accuracy after each epoch.

1.1 (10 points)

To-do:

You will need the write all of the following instance methods. Some of them might need to call the instance attributes and make modifications to them. You are free to modify the input parameters for the methods but do use the class object.

8. Instance method `predict` takes in a data sample (in the first dataset, the input is all the data from one patient), which would be a vector, returns the predicted label as the output, either -1 or 1. If you would like to optimize your classifier, you can also enable your `predict` method to process data in batches.
9. Instance method `accuracy` takes in a test dataset and a test label set. It returns the percentage of the samples that are correctly predicted using the current w and b.
10. Instance method `train` is where we apply gradient descent. We need to loop through the training dataset `self.epoch` number of times and for each sample that has been incorrectly classfied we need to update the weights and the bias of our classfier. At the end of each epoch, please store the training accuracy and test accuracy into the corresponding instance atributes.

In [ ]:
```
class perceptron:
    def __init__(self, x_train, y_train, x_test, y_test, learning_rate, epochs): #
        rng = np.random.default_rng()
        self.w = (2 * rng.random(size = x_train.shape[1]) - 1)
        self.b = 2 * rng.random() - 1
        if 0 < learning_rate <= 1:
            self.learning_rate = learning_rate
        else:
            raise ValueError('learning rate should be in (0, 1]')
        self.x_test = x_test
        self.y_test = y_test
        self.y_train = y_train
        self.x_train = x_train
        self.epochs = epochs
        self.train_accuracies = []
        self.test_accuracies = []
```

```
    def predict(self, x): # 2 point
        linear_output = np.dot(x, self.w) + self.b
        predictions = np.where(linear_output >= 0, 1, -1)
        return predictions

    def accuracy(self, x, y):    # 1 point
        predictions = self.predict(x)
        correct_predictions = np.sum(predictions == y)
        accuracy = correct_predictions / len(y)
        return accuracy


    def train(self):  # 5 point
        for epoch in range(self.epochs):
            predictions = self.predict(self.x_train)
            misclassified = predictions != self.y_train

            if np.any(misclassified):
                xi_misclassified = self.x_train[misclassified]
                yi_misclassified = self.y_train[misclassified]

                for xi, yi in zip(xi_misclassified, yi_misclassified):
                    error = yi - (np.dot(xi, self.w) + self.b)
                    self.w += self.learning_rate * error * xi
                    self.b += self.learning_rate * error

            train_accuracy = self.accuracy(self.x_train, self.y_train)
            self.train_accuracies.append(train_accuracy)

            test_accuracy = self.accuracy(self.x_test, self.y_test)
            self.test_accuracies.append(test_accuracy)
```

The dataset:

The first dataset we will be using is the UCI ML Breast Cancer Wisconsin (Diagnostic) dataset. The cell below is to help you understand the data. Each row i is a data sample recorded from the patient i, we denote as x_i. The label y_i is the diagnosis of that patient, either malignant (-1) or benign (1).

In [ ]:
```
breast_cancer = sklearn.datasets.load_breast_cancer()
data = pd.DataFrame(breast_cancer.data, columns = breast_cancer.feature_names)
print("The shape of our data:", data.shape)
data.head()
```

```
The shape of our data: (569, 30)
```

Out[ ]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	symm
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0

5 rows × 30 columns

1.2 (3 points)

To-do

1. Normalize the dataset using the equation from Assignment 1. (1 point)
2. Convert the labels of the data set into -1 and 1 so that it will work with our perceptron algorithm. (1 point)
3. Split into training and test set using `train_test_split` that has been imported for you. One of the parameters in `train_test_split` is `random_state`. Please is set it to 123. Just making sure we are not tuning `random_state`. Also, set `test_size` to 0.2. (1 point)

In [ ]:

```
data['class'] = breast_cancer.target
X = data.drop('class', axis = 1)
Y = data['class']

X = X.to_numpy(dtype = float)
Y = Y.to_numpy(dtype = float)


# YOUR CODE HERE
def normalize_data(data):
    mean = np.mean(data, axis = 0)
    std_dev = np.std(data, axis = 0)

    normalized_data = (data - mean) / std_dev

    return normalized_data

def change_target(Y):
    return np.where(Y==0, -1, Y)\

X_normalized = normalize_data(X)
Y_corrected = change_target(Y)
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X_normalized, Y_corrected, rand
```

1.3 (5 points)

To-do

1. Train the perceptron classifer and predict the labels for the training dataset. (no need to print the predicted labels) (1 points)
2. Test the model on the test dataset and print the accuracy score. (1 points)
3. Plot the training accuacy vs epoch graph and test accuracy vs epoch graph using `train_accuracies` and `train_accuracies`. (You can plot the them seperately or on the same graph.) (1 points)
4. What do you notice when you change the learning rate (how does your plot change)? (Make large changes, ex. 1E-2 to 1E-5. How does it effect convergence, weights/bias, etc) (1 points)

Write your Answer here:

With a lower learning rate the convergence is much smoother but it is also much slower. With a really high learning rate the graph is super jagged and almost looks periodic at times. With a low learning rate most of the weights are very close to 0. WIth really large numbers of epochs it is not uncommon to see models where none of the weights have an absolute value greater than one. With lower epochs and higher learning rates the weights will be more widely distributed.The bias term also gets closer to 0. When the learning rate is high (0.1) the model breaks. I asked ChatGPT and it said that this is becuase of numerical instability with trying to change the weights too much at once. 0.03 is the highest learning rate where the model doesn't concsitently break. There is a lot of noise in the graph with this learning rate, and it will frequenlty get a good score and then mess itself up. Higher learning rates conssitently lead to larger weights and biases

5. What happens when `epoch` is set too low or too high? (1 points)

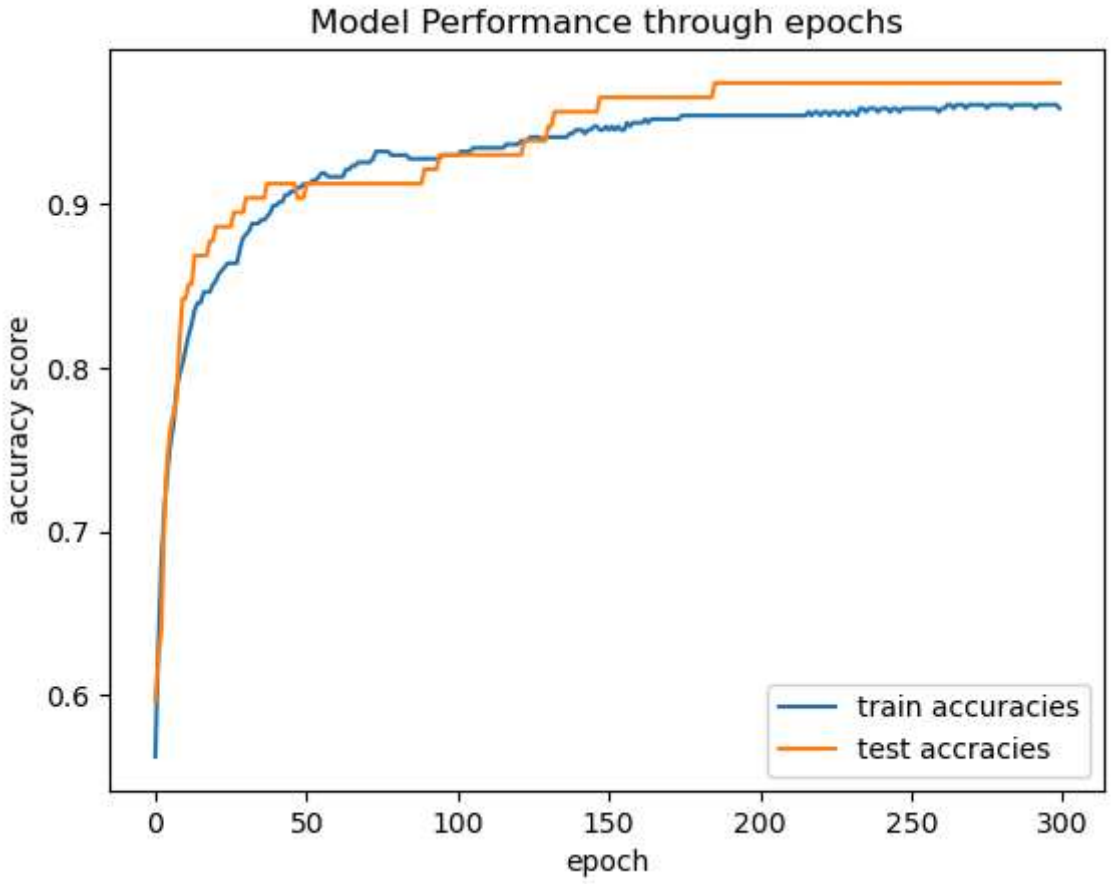
Write your Answer here:

When the learning rate is low enough there doesn't seem to be much of an issue with having too many epochs. After a while the test accuracy stops improving and there is noise in the train accuracies that seems to be gerneally trending upward. Anything beyond 500 epochs with the 1e-4 learning rate seems wholly unnecesary. The model can consistently get to 90% accuracy within 30 epochs. Given how quickly it runs I think that the extra 7% accuracy is well worthwhile. I could add something to my code that would stop training the

model once it stopped improving, but since I did not do that I was able to run for 300000 epoch and found that relatively quickly there was no change in additional epochs. More epochs also result in a lower value for the weights and bias.

In [ ]:
```
# 1. Train and predict
model = perceptron(X_train, Y_train, X_test, Y_test, learning_rate = 1e-4, epochs =

model.train()
model.predict(X_train)


# 2. Test and print the accuracy score.
accuracy = model.accuracy(X_train, Y_train)
print(accuracy)
# 3. Plot the graph.

fig, ax = plt.subplots()
ax.plot(model.train_accuracies, label = 'train accuracies')
ax.plot(model.test_accuracies, label = 'test accracies')
ax.set_xlabel('epoch')
ax.set_ylabel('accuracy score')
ax.set_title('Model Performance through epochs')
ax.legend()

print(model.w, model.b)
```

```
0.9582417582417583
[-0.25854257 -0.14593409  0.13353954 -0.50099443 -0.14507871 -0.37976046
  0.51282603  0.1931209  -0.30001166 -0.34243302  0.35677096 -0.03668949
 -0.4448006  -0.79978513  0.00450693  0.00728057  0.25581448 -0.07803329
  0.45159783 -0.20798138 -0.16678436 -0.42367642 -0.67545767 -0.5511226
 -0.02458509  0.25451659 -0.32769295 -0.52000734 -0.33641607  0.32578449] 0.16186983
738788224
```



1.4 (2 points)

To-do

Train the Sklearn perceptron using the same training dataset and print the test accuracy score after training. You can use accuracy_score from sklearn to help you calculate. We have already imported the function in the first cell.

Here is the documentation for the Sklearn perceptron.

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

Make sure use the same parameters as you used in your perceptron. It is recommended to read through all of the parameters sklearn perceptron has. You might have to change some of the default values.

Sklearn perceptron accuracy should only differ from the accuracy of your perceptron by one or two percent. This difference comes from the random initialization of our weights and biases.

In [ ]:
```
# Train and print out test accruacy score.


sklrn_model = sklearn.linear_model.Perceptron(max_iter = 50, shuffle = False)
```

```
sklrn_model.fit(X_train, Y_train)
predictions = sklrn_model.predict(X_test)


score = sklrn_model.score(X_test, Y_test)

print(score)
```

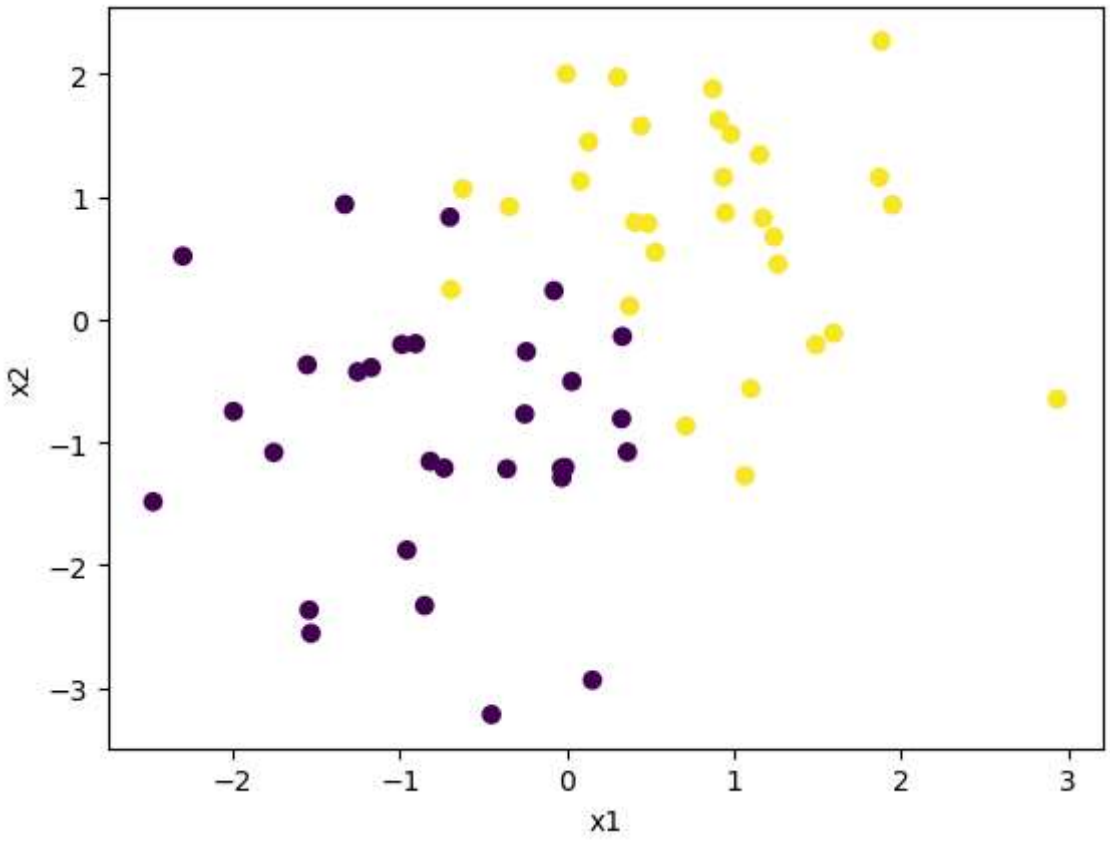
```
0.8947368421052632
```

Part 2

In this section we will be training the our model on a 2 dimensional toy dataset. This way we can visualize its decision boundary.

Below is the scatterplot of the dataset we will be classifying.

In [ ]:
```
SIZE = 30
np.random.seed(seed=1)
toy_data_0 = np.random.multivariate_normal([-0.75, -0.75], cov=np.array([[1, 0.1],
                                                                         [0.1, 1]])
toy_data_1 = np.random.multivariate_normal([0.75, 0.75], cov=np.array([[1, 0],
                                                                       [0, 1]]), si
toy_data = np.vstack((toy_data_0, toy_data_1))
toy_labels = np.hstack((np.ones(SIZE)*-1, np.ones(SIZE)))
plt.scatter(toy_data[:, 0], toy_data[:, 1], c=toy_labels)
plt.xlabel('x1')
plt.ylabel('x2')
X_train, X_test, Y_train, Y_test = train_test_split(toy_data, toy_labels, test_size
```



2.1 (2 points)

To-do

1. First, return the weights and the bias prior to training. Store them in variable `w` and `b`. (1 point)

In [ ]:
```
our_clf= perceptron(X_train, Y_train,X_test,Y_test, learning_rate= 1E-4, epochs=100
w = our_clf.w
b = our_clf.b

print(w)  # Do not remove.
print(b)  # Do not remove.
```

```
[0.64290816 0.50925429]
-0.8036213175997275
```

We denote the data sample (could be any datapoint) as $X = [x_1, x_2]$, weights for our dataset as $W = [w_1, w_2]$, and the bias as b.

To-do

2. Complete the equation for the decision boundary in terms of x_1, x_2, w_1, w_2 and b. If you are plotting a 2-D

x-y plot, x_2 is going to be the value of y. (1 point) (write it below)

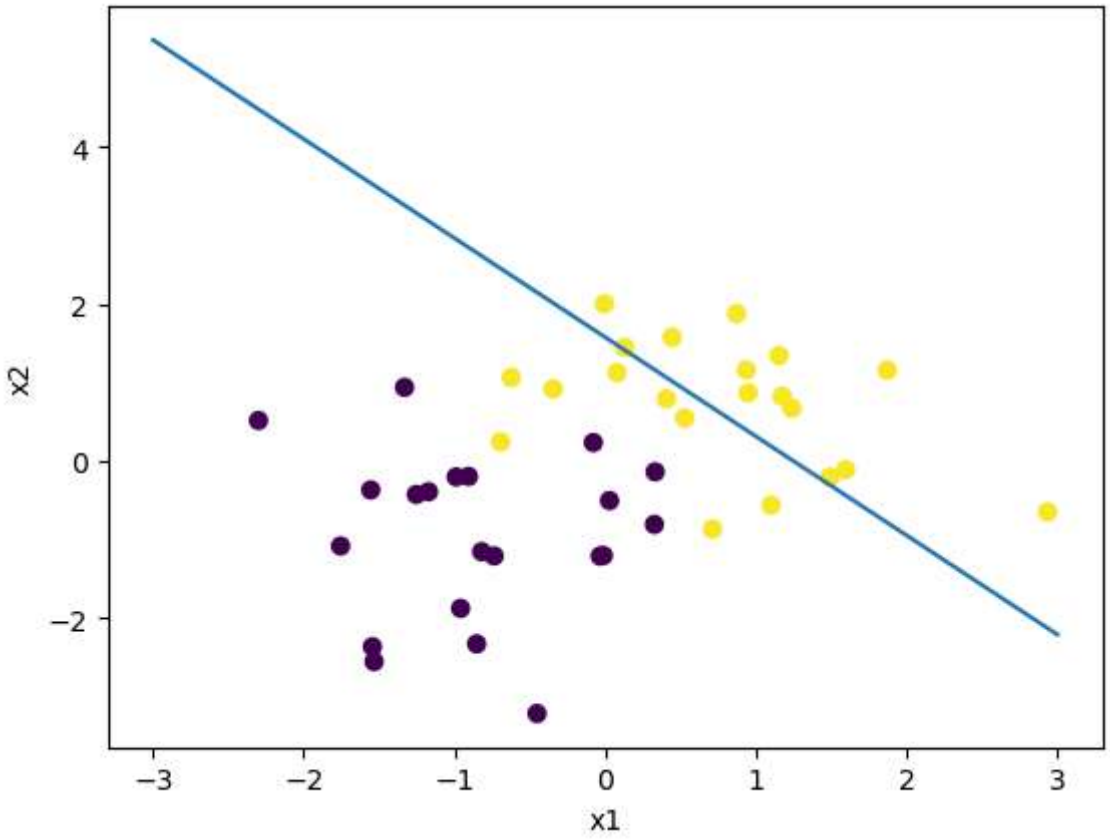
Hint: derive the equation from the decision boundary equation in the part 1 introduction. Or try to understand the provided code.

$x_2 = -(w_1/w_2 * x_1) - b/w_2$

We can visual our initial decision boundary.

In [ ]:
```
plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_train)
x= np.linspace(-3,3, 100)
x2= -(w[0]/w[1])*x - b/w[1]
plt.plot(x, x2)
plt.xlabel('x1')
plt.ylabel('x2')
```

Out[ ]:
```
Text(0, 0.5, 'x2')
```



2.2 (3 points)

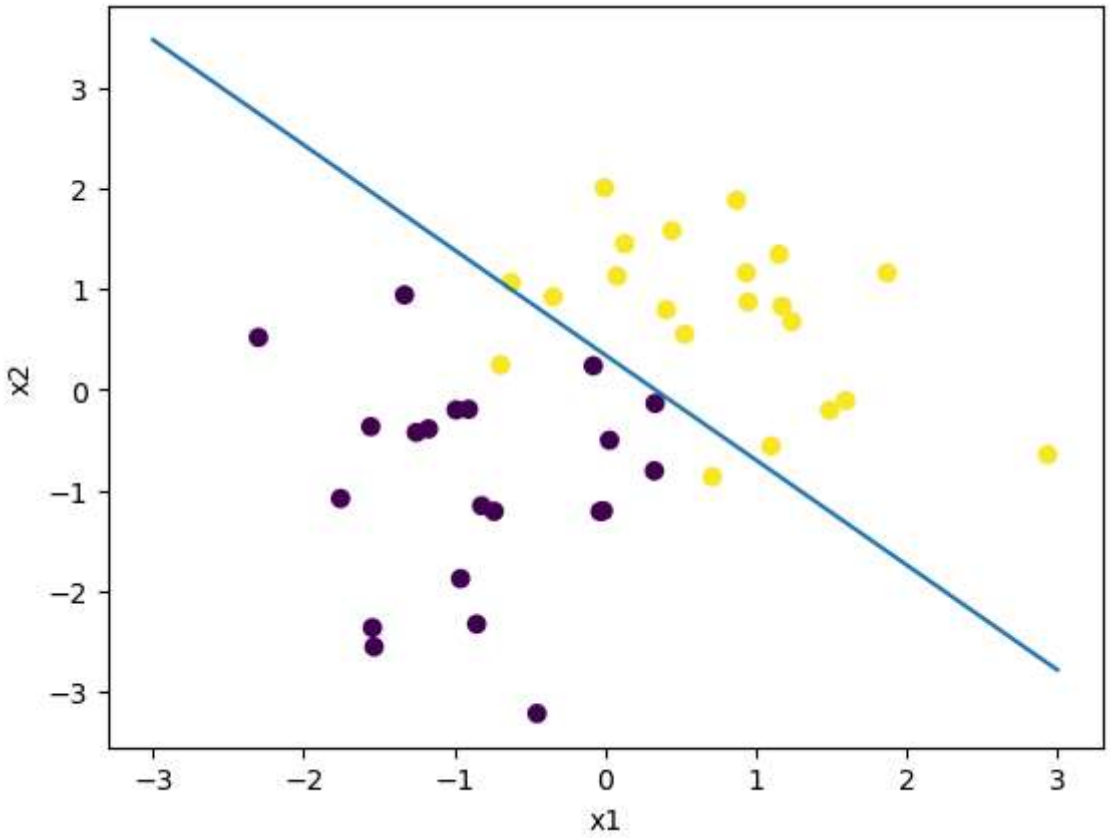
To-do

1. Train your perceptron classifier on the dataset. (2 points)
2. Plot the decision boudary of the trained perceptron. (Feel free to use the provided code from the problem above.) (1 point)

In [ ]:
```
# Write down your code here
```

```
our_clf.train()

w = our_clf.w
b = our_clf.b
plt.scatter(X_train[:, 0], X_train[:, 1], c=Y_train)
x= np.linspace(-3,3, 100)
x2= -(w[0]/w[1])*x - b/w[1]
plt.plot(x, x2)
plt.xlabel('x1')
plt.ylabel('x2')
```

Out[ ]: Text(0, 0.5, 'x2')



Part 3

3.1 (Breast cancer dataset) (2 points)

Logistic regression is another binary classification algorithm.

To-do:

1. Train the Sklearn logistic regression using the breast cancer training dataset. (1 point)
2. Print the test accuracy score after training. (1 point)

Here is the documentation for the Sklearn logistic regression .

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

It is recommended to read through all of the parameters Sklearn logistic regression has.

In [ ]:
```
import sklearn.linear_model # type: ignore


breast_cancer = sklearn.datasets.load_breast_cancer()
data = pd.DataFrame(breast_cancer.data, columns = breast_cancer.feature_names)
data['class'] = breast_cancer.target
X = data.drop('class', axis = 1)
Y = data['class']

X = X.to_numpy(dtype = float)
Y = Y.to_numpy(dtype = float)
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.3, stratify

# Train your model
regressor = sklearn.linear_model.LogisticRegression()
regressor.fit(X_train, Y_train)
regressor.predict(X_test)

# Print the accuracy score
print(regressor.score(X_test, Y_test))
```

```
0.9590643274853801
```

```
c:\Users\SmotP\anaconda3\envs\COGS118A\lib\site-packages\sklearn\linear_model\_logis
tic.py:818: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG,
```

3.2 (Toy dataset created in Part 2) (3 points)

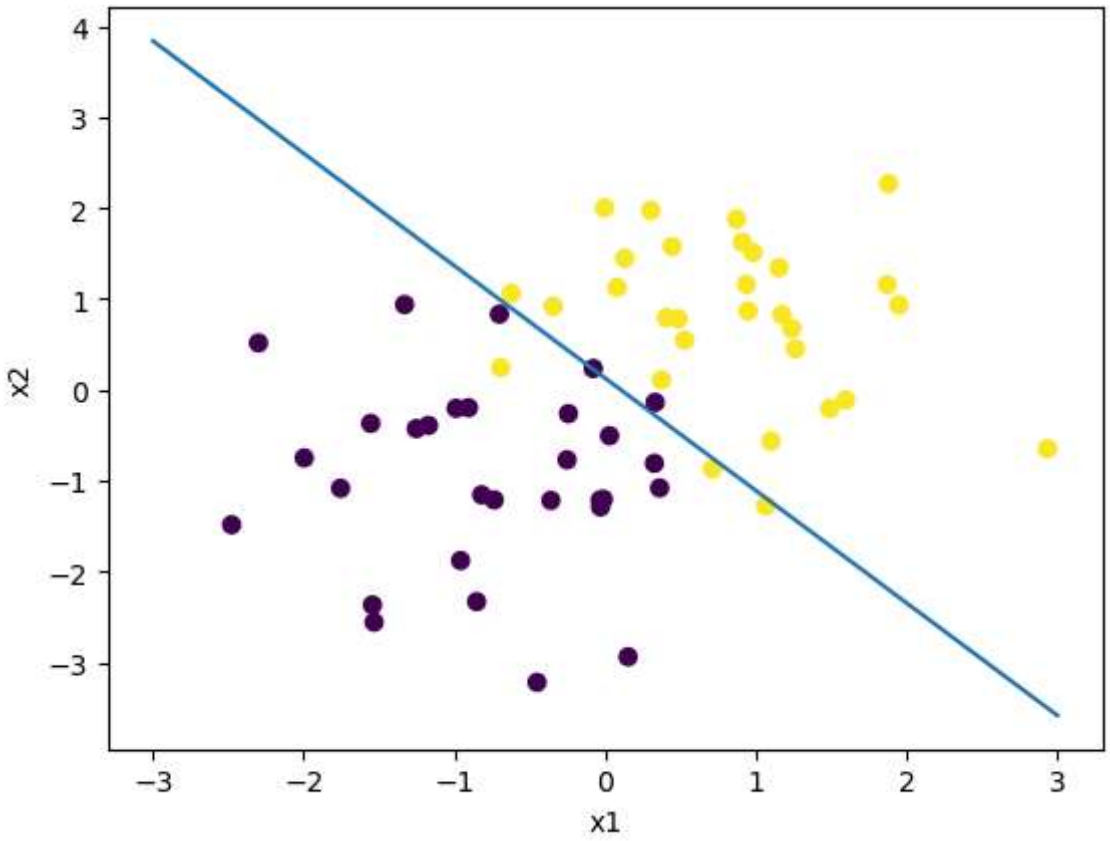
To-do

1. Train the logistic regression on the toy dataset. (1 point)
2. Print the weights and the bias after training. If you are not sure how to obtain them, Google would be a good resource. (1 point)
3. Plot the decision boundary. You can copy the code from problem 2. (1 point)

In [ ]:
```
# Train your model
regressor.fit(toy_data, toy_labels)
# Print out the weights and the bias.
print(regressor.coef_)
# Plot the decision boundary.
plt.scatter(toy_data[:, 0], toy_data[:, 1], c=toy_labels)
x= np.linspace(-3,3, 100)
x2= -(regressor.coef_[0][0]/regressor.coef_[0][1])*x - b/regressor.coef_[0][1]
plt.plot(x, x2)
plt.xlabel('x1')
plt.ylabel('x2')
```

```
[[1.95764525 1.58342264]]
```

Out[ ]: Text(0, 0.5, 'x2')



Submission: Please submit the ipynb notebook file and a PDF copy to Gradescope. Do not create a zip file. Thanks.