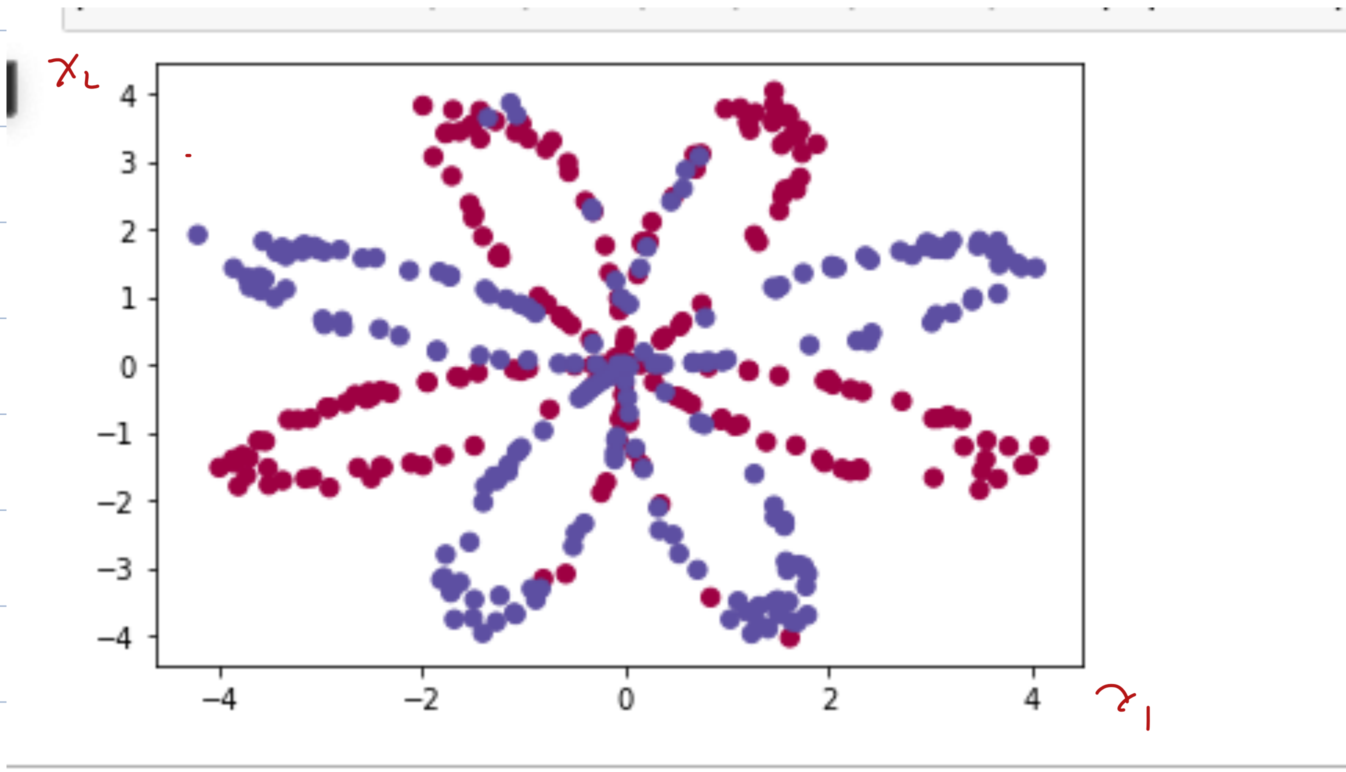
numpy Array $X$ that contains $(x_1, x_2)$ features

No not $64 \times 64 \times 3$,

only two, $x_1$ and $x_2$ axis




You have:

$y \rightarrow$ Contains labels $\boxed{\text{Red}: 0, \text{ blue}: 1.}$

$\therefore$ find how many examples, and the shape $X$.

$$X = \begin{bmatrix} | & | & & \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & \end{bmatrix}$$

$$X = \underbrace{\begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \cdots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & & x_2^{(m)} \end{bmatrix}}_{m.} \; n_x$$

$n = X.\text{shape}[1]$ ? is $m$

## 3 - Simple Logistic Regression

Before building a full neural network, let's check how logistic regression performs on this problem. You can use sklearn's built-in functions for this. Run the code below to train a logistic regression classifier on the dataset.

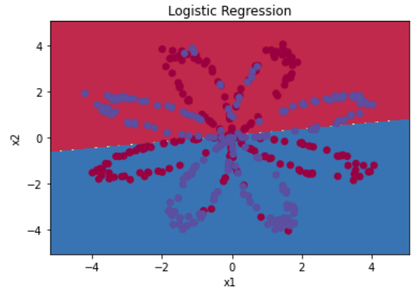
In [6]:
```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

You can now plot the decision boundary of these models! Run the code below.

In [7]:
```
 Plot the decision boundary for logistic regression
lot_decision_boundary(lambda x: clf.predict(x), X, Y)
lt.title("Logistic Regression")

 Print accuracy
R_predictions = clf.predict(X.T)
rint ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR_predictions) + np.dot(1-Y,1-LR_predictions))/float
      '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 47 % (percentage of correctly labelled datapoints)
```



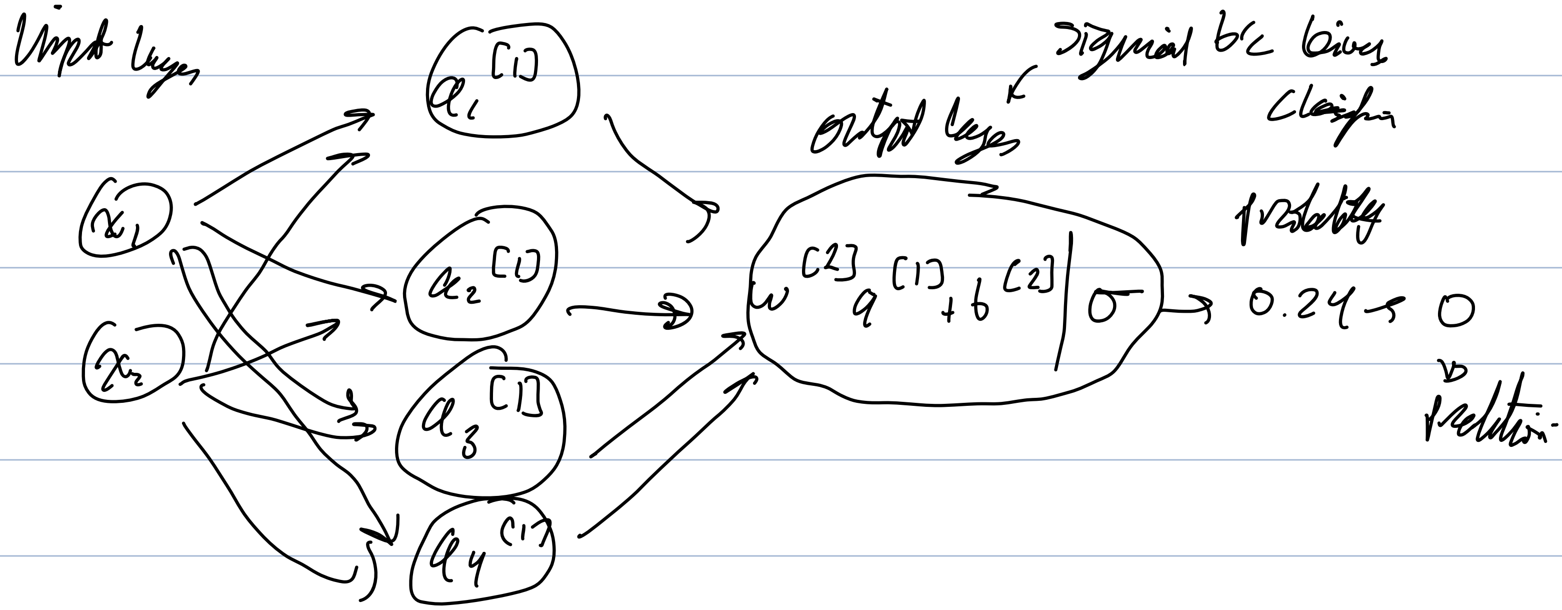

→ Since the dataset cannot be separated linearly it doesn't perform well.

**Expected Output:**

| | |
|---|---|
| Accuracy | 47% |

Try with Neural Networks Now:

w/ single hidden layer




hidden layer,
4 neurons,
$g = \tanh$.

Assume 1 example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \tanh(z^{[1](i)})$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)})$$

$$\hat{y}^{(i)} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

∴ given predictions on everything, Cost $J$:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log(\underbrace{\hat{y}^{(i)}}_{a^{[2](i)}}) + (1-y^{(i)})\log(\underbrace{\hat{y}^{(i)}}_{a^{[2](i)}})\right)$$

⇒ General methodology :

1) Define Neural Network structure ( # inputs, # of hidden units etc. )
2) Initialize parameters randomly.
3) Loop :
   - forward propagation
   - Compute loss
   - backward propagation to get gradient
   - update parameters ( gradient descent )

Helper functions for 1-3, then call in fcn nn_model().
Then predictions on new data.

Forward Prop

$$Z^{[1]} = W^{[1]} \overset{A^{[0]}}{X} + b^{[1]}$$
$$A^{[1]} = \tanh(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

Cross Entropy loss.

$$J = -\frac{1}{m}\sum \left( y^{(i)} \log(a^{[2](i)}) + (1-y^{(i)})\log(1-a^{[2](i)}) \right)$$

$A^{[2]} \nearrow$ is all $a^{[2](i)}$

$$A^{[2]} = \begin{bmatrix} \vert & & \\ a^{[2](1)} & \cdots & a^{[2](m)} \\ \vert & & \end{bmatrix}$$

## Backward Prop using the Eqns provided

### 4.5 - Implement Backpropagation

Using the cache computed during forward propagation, you can now implement backward propagation.

lab

### Exercise 6 - backward_propagation

Implement the function `backward_propagation()`.

**Instructions**: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.

**Summary of gradient descent**

| | |
|---|---|
| $dz^{[2]} = a^{[2]} - y$ | $dZ^{[2]} = A^{[2]} - Y$ |
| $dW^{[2]} = dz^{[2]}a^{[1]^T}$ | $dW^{[2]} = \frac{1}{m} dZ^{[2]}A^{[1]^T}$ |
| $db^{[2]} = dz^{[2]}$ | $db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$ |
| $dz^{[1]} = W^{[2]T}dz^{[2]} * g^{[1]\prime}(z^{[1]})$ | $dZ^{[1]} = W^{[2]T}dZ^{[2]} * g^{[1]\prime}(Z^{[1]})$ |
| $dW^{[1]} = dz^{[1]}x^T$ | $dW^{[1]} = \frac{1}{m} dZ^{[1]}X^T$ |
| $db^{[1]} = dz^{[1]}$ | $db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$ |

Andrew Ng

**Figure 1**: Backpropagation. Use the six equations on the right.

- Tips:
  - To compute dZ1 you'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So you can compute $g^{[1]'}(Z^{[1]})$ using `(1 - np.power(A1, 2))`.

## Next update Parameters

Gradient descent.

$$\theta = \theta - \alpha \frac{dJ}{d\theta}$$

for $\theta' = W^1, W_2, b_1, b_2$