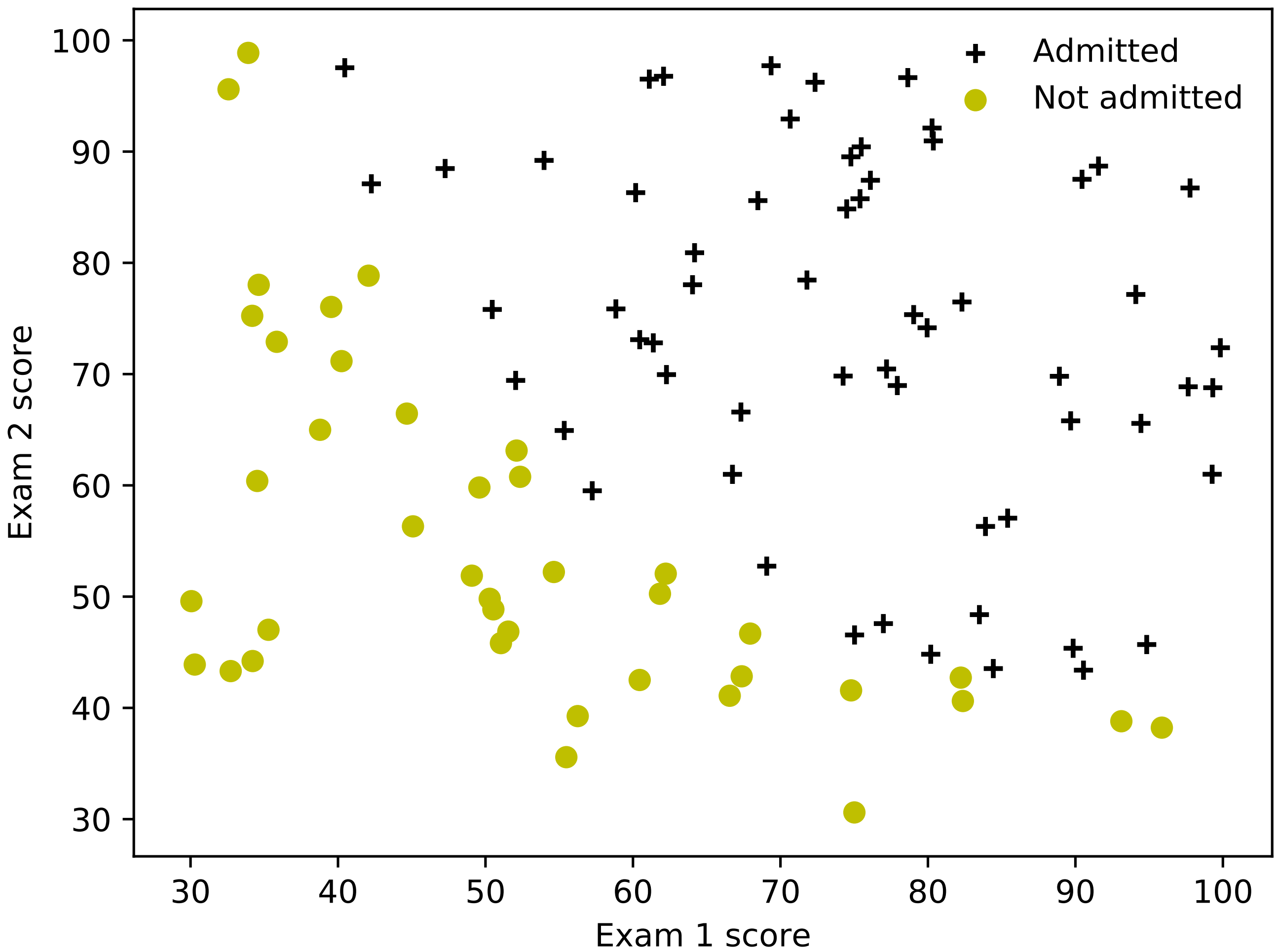
Práctica 2: regresión logística



Admitted
Not admitted
Exam 2 score
Exam 1 score
100
90
80
70
60
50
40
30
30
40
50
60
70
80
90
100

numpy.where

numpy.where(*condition*[, *x*, *y*])

Return elements, either from *x* or *y*, depending on *condition*.

If only *condition* is given, return `condition.nonzero()`.

Parameters: **condition** : ***array_like, bool***

When True, yield *x*, otherwise yield *y*.

x, y : ***array_like, optional***

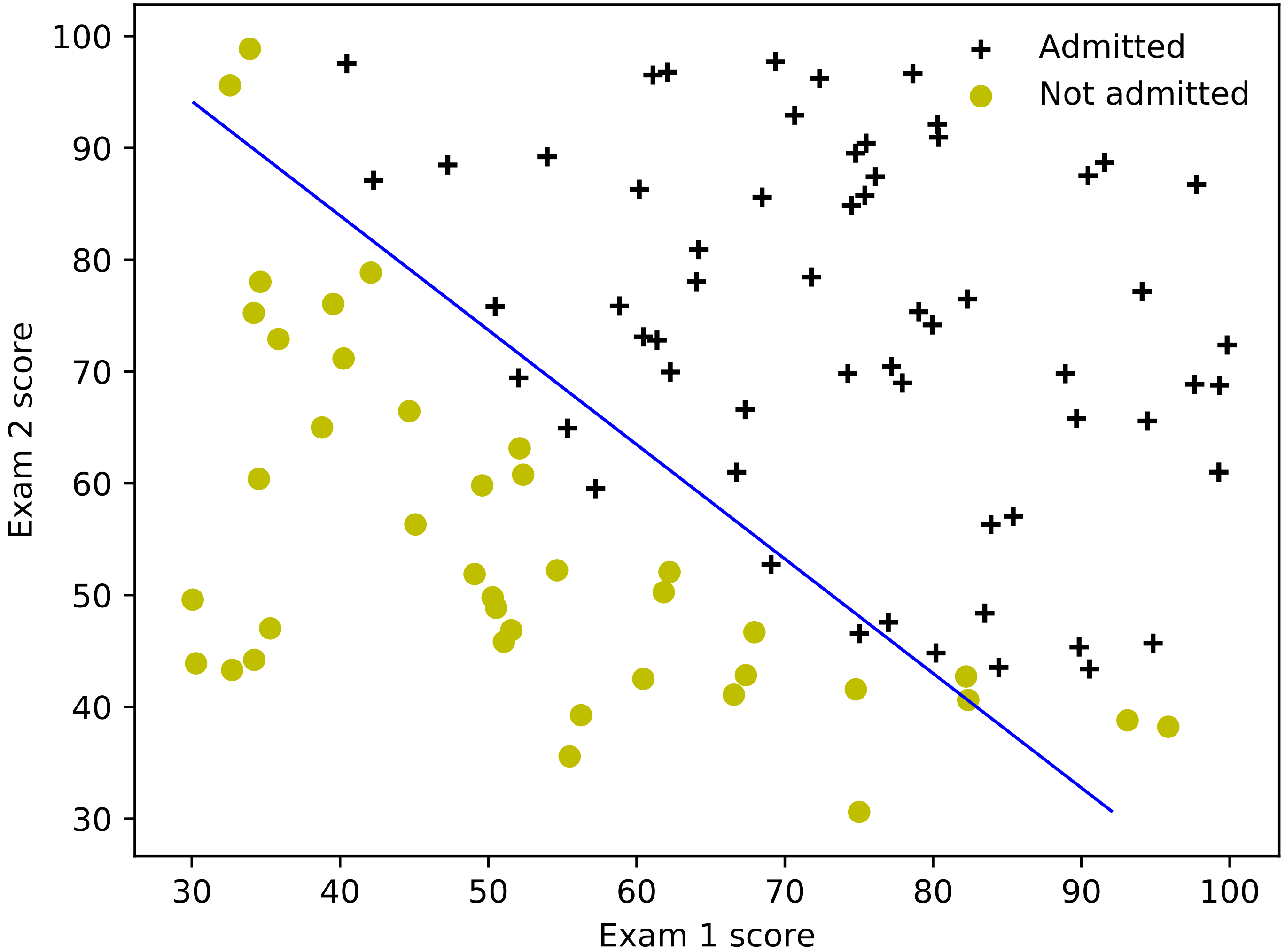
Values from which to choose. *x*, *y* and *condition* need to be broadcastable to some shape.

Returns: **out** : ***ndarray or tuple of ndarrays***

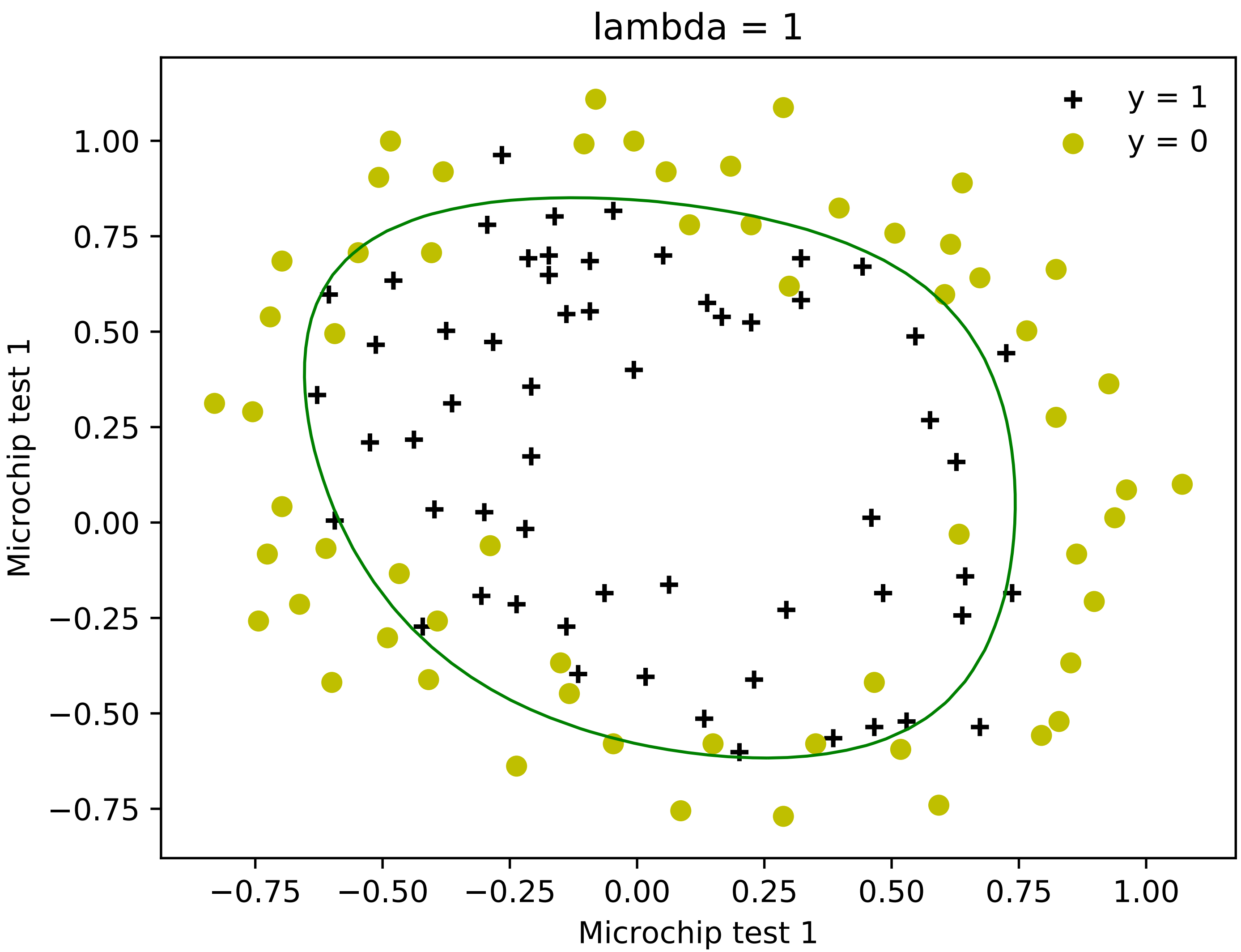
If both *x* and *y* are specified, the output array contains elements of *x* where *condition* is True, and elements from *y* elsewhere.
If only *condition* is given, return the tuple `condition.nonzero()`, the indices where *condition* is True.

```
# Obtiene un vector con los índices de los ejemplos positivos
pos = np.where(Y == 1)

# Dibuja los ejemplos positivos
plt.scatter(X[pos, 0], X[pos, 1], marker='+', c='k')
```



Admitted
Not admitted
Exam 2 score
Exam 1 score
100
90
80
70
60
50
40
30
30
40
50
60
70
80
90
100


lambda = 1
y = 1
y = 0
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
Microchip test 1
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
Microchip test 1

```python
def sigmoid(x):
    s = 1 / (1 + np.exp(-x))
    return s
```

```python
def pinta_frontera_recta(X, Y, theta):

    plt.figure()
    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
    x2_min, x2_max = X[:, 1].min(), X[:, 1].max()

    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max),
                           np.linspace(x2_min, x2_max))

    h = sigmoid(np.c_[np.ones((xx1.ravel().shape[0], 1)),
                      xx1.ravel(),
                      xx2.ravel()].dot(theta))
    h = h.reshape(xx1.shape)

    # el cuarto parámetro es el valor de z cuya frontera se
    # quiere pintar
    plt.contour(xx1, xx2, h, [0.5], linewidths=1, colors='b')
    plt.savefig("frontera.pdf")
    plt.close()
```

```
import numpy as np

In [3]: x = np.array([1,2,3])

In [4]: y = np.array([4,5,6])

In [5]: xx, yy = np.meshgrid(x,y)

In [6]: xx
Out[6]:
array([[1, 2, 3],
       [1, 2, 3],
       [1, 2, 3]])

In [7]: yy
Out[7]:
array([[4, 4, 4],
       [5, 5, 5],
       [6, 6, 6]])
```

numpy.c

`numpy.c_` = *<numpy.lib.index_tricks.CClass object>*¶

Translates slice objects to concatenation along the second axis.

This is short-hand for `np.r_['-1,2,0', index expression]`, which is useful because of its common occurrence. In particular, arrays will be stacked along their last axis after being upgraded to at least 2-D with 1's post-pended to the shape (column vectors made out of 1-D arrays).

See also:

`column_stack`	Stack 1-D arrays as columns into a 2-D array.
`r_`	For more detailed documentation.

Examples

```
>>> np.c_[np.array([1,2,3]), np.array([4,5,6])]
array([[1, 4],
       [2, 5],
       [3, 6]])
>>> np.c_[np.array([[1,2,3]]), 0, 0, np.array([[4,5,6]])]
array([[1, 2, 3, 0, 0, 4, 5, 6]])
```

numpy.ravel

numpy.**ravel**(*a*, *order='C'*) [source]

Return a contiguous flattened array.

A 1-D array, containing the elements of the input, is returned. A copy is made only if needed.

As of NumPy 1.10, the returned array will have the same type as the input array. (for example, a masked array will be returned for a masked array input)

Parameters:	**a** : *array_like*
	Input array. The elements in *a* are read in the order specified by *order*, and packed as a 1-D array.

```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(np.ravel(x))
[1 2 3 4 5 6]
```

sklearn.preprocessing.PolynomialFeatures

class `sklearn.preprocessing.` **`PolynomialFeatures`**(*degree=2*, *interaction_only=False*, *include_bias=True*) [source]

Generate polynomial and interaction features.

Generate a new feature matrix consisting of all polynomial combinations of the features with degree less than or equal to the specified degree. For example, if an input sample is two dimensional and of the form [a, b], the degree-2 polynomial features are [1, a, b, a^2, ab, b^2].

Parameters:

degree : *integer*

The degree of the polynomial features. Default = 2.

interaction_only : *boolean, default = False*

If true, only interaction features are produced: features that are products of at most `degree` *distinct* input features (so not `x[1] ** 2`, `x[0] * x[2] ** 3`, etc.).

include_bias : *boolean*

If True (default), then include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

```
>>> X = np.arange(6).reshape(3, 2)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5]])

>>> poly = PolynomialFeatures(2)
>>> poly.fit_transform(X)
array([[ 1.,  0.,  1.,  0.,  0.,  1.],
       [ 1.,  2.,  3.,  4.,  6.,  9.],
       [ 1.,  4.,  5., 16., 20., 25.]])
```

```
def plot_decisionboundary(X, Y, theta, poly):

    plt.figure()

    x1_min, x1_max = X[:, 0].min(), X[:, 0].max()
    x2_min, x2_max = X[:, 1].min(), X[:, 1].max()
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max),
                           np.linspace(x2_min, x2_max))

    h = sigmoid(poly.fit_transform(np.c_[xx1.ravel(),
                                         xx2.ravel()]).dot(theta))
    h = h.reshape(xx1.shape)

    plt.contour(xx1, xx2, h, [0.5], linewidths=1, colors='g')

    plt.savefig("boundary.pdf")
    plt.close()
```

Vectorización

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log(h_\theta(x^{(i)})) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix} \quad y \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

$$X\theta = \begin{bmatrix} (x^{(1)})^T\theta \\ (x^{(2)})^T\theta \\ \vdots \\ (x^{(m)})^T\theta \end{bmatrix} = \begin{bmatrix} \theta^T(x^{(1)}) \\ \theta^T(x^{(2)}) \\ \vdots \\ \theta^T(x^{(m)}) \end{bmatrix}$$

$$J(\theta) = -\frac{1}{m}((\,log\,(g(X\theta)))^T y + (\,log\,(1-g(X\theta)))^T(1-y))$$

$$J(\theta) = -\frac{1}{m}((\,log\,(g(X\theta)))^T y + (\,log\,(1 - g(X\theta)))^T (1 - y))$$

```python
def cost(theta, X, Y):

    # H = sigmoid(np.matmul(X, np.transpose(theta)))

    H = sigmoid(np.matmul(X, theta))

    # cost = (- 1 / (len(X))) * np.sum( Y * np.log(H) +
                                         (1 - Y) * np.log(1 - H) )

    cost = (- 1 / (len(X))) * (np.dot(Y, np.log(H)) +
                               np.dot((1 - Y), np.log(1 - H)))

    return cost
```

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

$$\begin{aligned}
\begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_0} \\ \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix} &= \frac{1}{m}\begin{bmatrix} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)} \\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)} \\ \vdots \\ \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_n^{(i)} \end{bmatrix} \\
&= \frac{1}{m}\sum_{i=1}^{m}\left((h_\theta(x^{(i)}) - y^{(i)})x^{(i)}\right) \\
&= \frac{1}{m}X^T(h_\theta(x) - y)
\end{aligned}$$

$$h_\theta(x) - y = \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ h_\theta(x^{(2)}) - y^{(2)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}$$

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m}X^T(g(X\theta) - y)$$

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m} X^T (g(X\theta) - y)$$

```python
def gradient(theta, XX, Y):

    H = sigmoid( np.matmul(XX, theta) )

    grad = (1 / len(Y)) * np.matmul(XX.T, H - Y)
    return grad
```