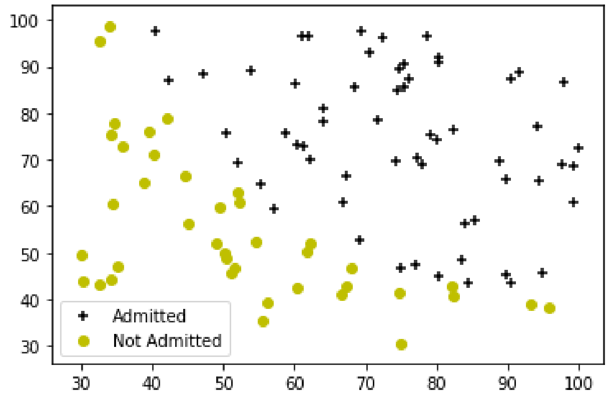
# Práctica 2: Regresión Logística

Lo primero es visualizar en una gráfica los casos positivos y negativos, leyendo los datos del archivo *ex2data1.csv*, que proporciona los ejemplos de entrenamiento:

In [6]:
```
data = carga_csv("ex2data1.csv")
X = data[:, :-1]
Y = data[:, -1]

pos = np.where(Y == 1);
neg = np.where(Y == 0);
pts_pos = plt.scatter(X[pos, 0], X[pos, 1], c='k', marker='+')
pts_neg = plt.scatter(X[neg, 0], X[neg, 1], c='y', marker='o')
plt.legend((pts_pos, pts_neg), ('Admitted', 'Not Admitted'))
plt.show()
```




En segundo lugar, vamos a implementar la función sigmoide, aplicable indistintamente a un número, vector o matriz. Es una implementación directa de su expresión matemática:

$$g(z) = \frac{1}{1 + e^{-z}}$$

In [7]:
```
def g(z):
    return 1 / (1 + np.exp(-z))
```

A continuación, implementaremos las funciones de gradiente y de coste. La función *coste* en regresión logística es una implementación vectorizada de su expresión matemática:

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

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

In [8]:
```
def coste(Theta, X, Y):
    m = np.shape(X)[0]
    Aux = (np.log(g(X @ Theta))).T @ Y
    Aux += (np.log(1 - g(X @ Theta))).T @ (1 - Y)
    return -Aux / m
```

La función *gradiente* en regresión logística también es una implementación vectorizada de su correspondiente expresión matemática:

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

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

In [9]:
```
def gradiente(Theta, X, Y):
    m = np.shape(X)[0]
    Aux = X.T @ (g(X @ Theta) - Y)
    return Aux / m
```

Si comprobamos ambas funciones con valores de cero para *Theta*, obtenemos los resultados esperados:

In [10]:
```
data = carga_csv("ex2data1.csv")
X = data[:, :-1]
Y = data[:, -1]

m = np.shape(X)[0]
X_ones = np.hstack([np.ones([m, 1]), X])
n = np.shape(X_ones)[1]

Theta = np.zeros(n)
print(coste(Theta, X_ones, Y))
print(gradiente(Theta, X_ones, Y))
```

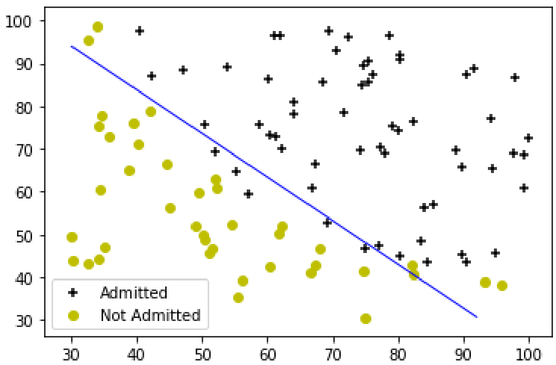
```
0.6931471805599452
[ -0.1         -12.00921659 -11.26284221]
```

A continuación, pondremos a prueba nuestras funciones de coste y gradiente calculando el valor óptimo de los parámetros *Theta*. Para ello utilizaremos la función *fmin_tnc* del submódulo *optimize* de *SciPy*. De nuevo, el valor óptimo es el esperado:

In [13]:
```
result = opt.fmin_tnc(func=coste, x0=Theta,
                      fprime=gradiente, args=(X_ones, Y))
theta_opt = result[0]
print(coste(theta_opt, X_ones, Y))
```

```
0.20349770158947497
```

Ahora podemos pintar la recta de decisión a partir de los valores optimizados de *Theta*:

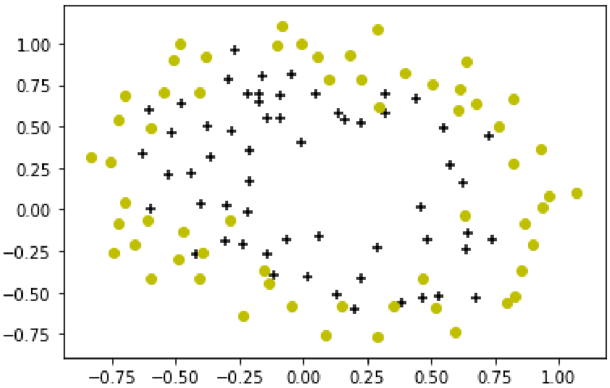


Si comparamos los resultados obtenidos de la regresión logística con los de los ejemplos de entrenamiento, comprobaremos que son bastante acertados. Para hacerlo, debemos aplicar la función sigmoide a los ejemplos de entrenamiento, con los valores óptimos de *Theta* obtenidos; después, comparamos con los datos y obtenemos el porcentaje de positivos y negativos reales:

```
In [35]: z = X_ones @ theta_opt
z = g(z)

pos = np.where(Y == 1);
neg = np.where(Y == 0);
z_pos = np.where(z >= 0.5)
z_neg = np.where(z < 0.5)

pos_perc = np.shape(z_pos)[1] * 100 / np.shape(pos)[1]
neg_perc = np.shape(z_neg)[1] * 100 / np.shape(neg)[1]

if pos_perc > 100:
    pos_perc = 200 - pos_perc

if neg_perc > 100:
    neg_perc = 200 - neg_perc

print("{}% of positives accuracy.".format(pos_perc))
print("{}% of negatives accuracy.".format(neg_perc))

98.33333333333333% of positives accuracy.
97.5% of negatives accuracy.
```

El segundo dataset presenta una dispersión de los datos que impide trazar una frontera de decisión recta. Primero, cargamos los datos del archivo *ex2data2.csv* y los mostramos en una gráfica:

```
In [17]: data = carga_csv("ex2data2.csv")
X = data[:, :-1]
Y = data[:, -1]

pos = np.where(Y == 1);
neg = np.where(Y == 0);
pts_pos = plt.scatter(X[pos, 0], X[pos, 1], c='k', marker='+')
pts_neg = plt.scatter(X[neg, 0], X[neg, 1], c='y', marker='o')
plt.show()
```




Para obtener un mejor ajuste a los ejemplos de entrenamiento añadiremos nuevos atributos a la descripción de los ejemplos, combinando los atributos originales. Gracias a la clase *PolynomialFeatures* del módulo *sklearn.preprocessing* podremos extender cada ejemplo de entrenamiento hasta la sexta potencia, completando un total de 28 atributos por cada ejemplo:

In [40]:
```python
poly = PolynomialFeatures(degree=6)
X_reg = poly.fit_transform(X)
X_reg.shape
```
Out[40]:
```
(118, 28)
```

La función *coste* para la regresión logística regularizada es una implementación vectorizada de su expresión matemática:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\left[-y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) - (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

$$J(\theta) = -\frac{1}{m}((\log(g(X\theta)))^T Y + (\log(1-g(X\theta)))^T(1-Y) + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

In [42]:
```python
def coste_reg(Theta, X, Y, Lambda):
    m = np.shape(X)[0]
    Aux = (np.log(g(X @ Theta))).T @ Y
    Aux += (np.log(1 - g(X @ Theta))).T @ (1 - Y)
    Cost = -Aux / m
    Regcost = (Lambda / (2 * m)) * sum(Theta ** 2)
    return Cost + Regcost
```

Si inicializamos la tasa de aprendizaje *Lambda* a 1.0 y todos los elementos de $\theta$ a cero, obtenemos los resultados esperados:

In [47]:
```python
Lambda = 1.0
Theta = np.zeros(n)
print(coste_reg(Theta, X_reg, Y, Lambda))
```
```
0.6931471805599453
```

La función *gradiente* para la regresión logística regularizada es una implementación vectorizada de su expresión matemática:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)} \qquad para\ j = 0$$

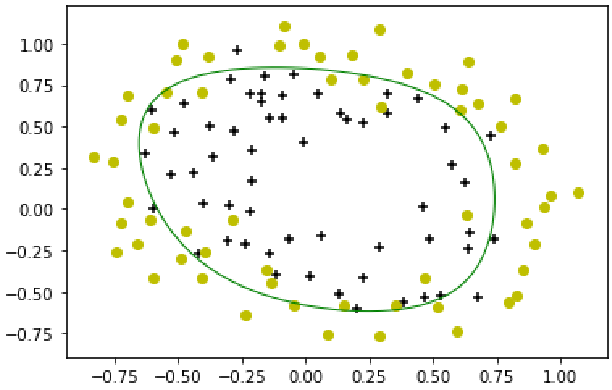
$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{m}\theta_j \qquad para\ j \geq 1$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m}X^T(g(X\theta) - Y) + \frac{\lambda}{m}\theta_j$$

In [63]:
```python
def gradiente_reg(Theta, X, Y, Lambda):
    m = np.shape(X)[0]
    Aux = X.T @ (g(X @ Theta) - Y)
    Grad = Aux / m
    Grad[1:] = Grad[1:] + (Lambda / m) * Theta[1:]
    return Grad
```

Ahora podemos volver a aplicar la función *fmin_tnc* para comprobar el resultado:

```
In [69]: data = carga_csv("ex2data2.csv")
         X = data[:, :-1]
         Y = data[:, -1]

         m = np.shape(X)[0]
         poly = PolynomialFeatures(degree=6)
         X_reg = poly.fit_transform(X)
         n = np.shape(X_reg)[1]

         Lambda = 1.0
         Theta = np.zeros(n)
         result = opt.fmin_tnc(func=coste_reg, x0=Theta,
                               fprime=gradiente_reg, args=(X_reg, Y, Lambda))
         theta_opt = result[0]
         plot_decisionboundary(theta_opt, X, Y, poly)
```



Hemos obtenido el valor óptimo de $\theta$ para la versión regularizada de la función de coste. Sin embargo, todavía podemos variar el valor que le damos a *Lambda* para ajustar aún más la frontera de decisión. A continuación, se exponen las gráficas obtenidas para varios valores de *Lambda*:

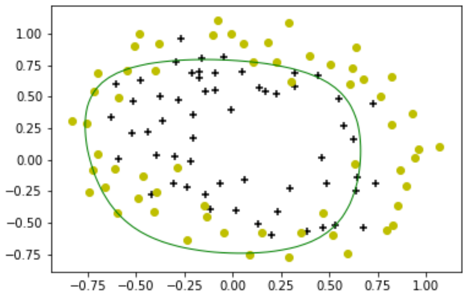


*Ilustración 1 - Lambda 10.0*

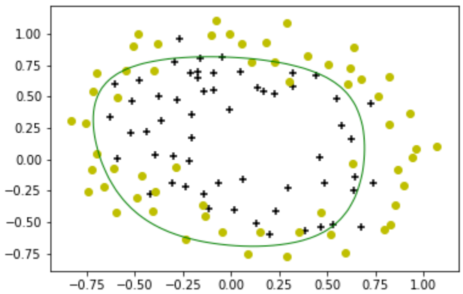


*Ilustración 2 - Lambda 5.0*

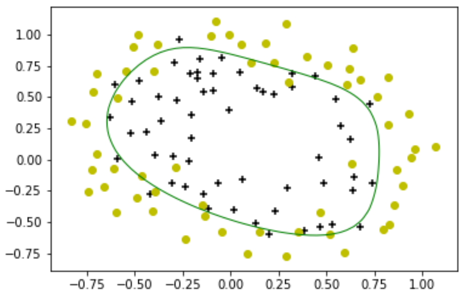


*Ilustración 3 - Lambda 0.01*

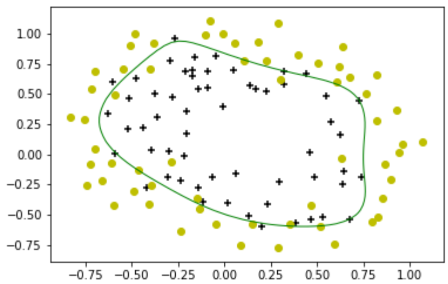


*Ilustración 4 - Lambda 0.001*