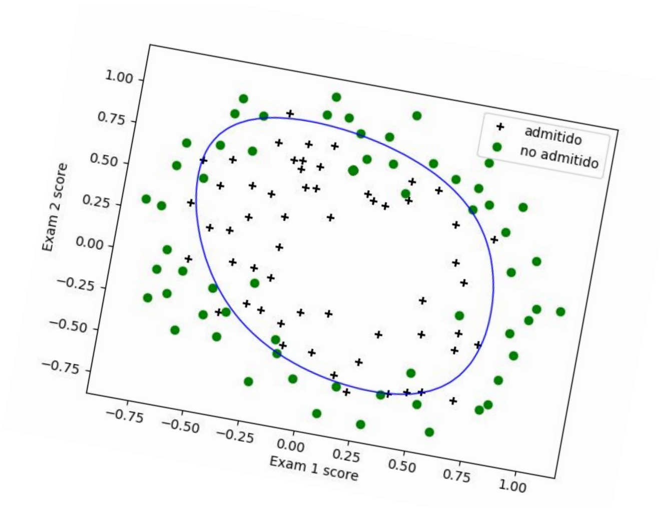
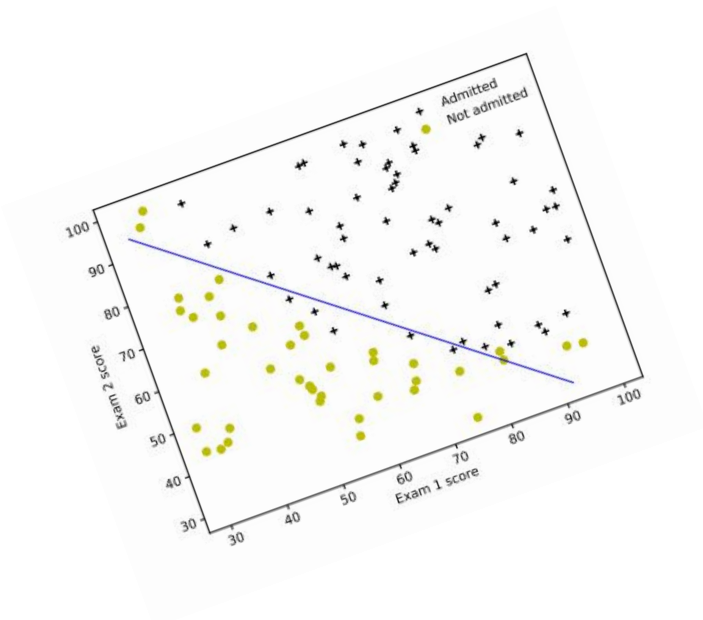
REGRESIÓN LOGÍSTICA:

A) SIN REGULARIZACIÓN

B) CON REGULARIZACIÓN





Realizado por:
Montserrat Sacie Alcázar y
Tomás Golomb Durán
Universidad Complutense de Madrid
Asignatura: Aprendizaje Automático y Big Data
Profesor: Pedro Antonio González Calero
Curso: 2018 - 2019

A) REGRESIÓN LOGÍSTICA SIN REGULARIZACIÓN

En la primera parte de esta práctica qué alumnos han sido admitidos a la universidad o no en función de las calificaciones de dos exámenes. Así tenemos dos variables para la nota del examen1 (x1) y del examen2(x2) y una variable binaria y (0 - no admitido y 1-admitido).

```
def sigmoide(z):
    return 1 / (1 + np.exp(-1*z))
```

Función sigmoide

Calcula la función sigmoide al valor de la variable z, que puede ser un número, una matriz o un vector. Devuelve el resultado que será del mismo tipo que z.

```
def coste(cThetas, mX, cY):
    cThetas = np.matrix(cThetas).transpose()
    gX = sigmoide(np.dot(mX, cThetas))
    s1 = np.dot((np.log(gX)).transpose(), cY)
    s2 = np.dot((np.log(1 - gX)).transpose(), 1 - cY)
    # Es una matriz de 1x1
    return np.ravel((-1/float(len(mX)))*(s1 + s2))[0]
```

Función coste

La función coste es una función vectorizada que recibe como parámetros una columna con los θ iniciales y los ejemplos de entrenamiento separados en una matrizX con la columna de 1 añadida(variable independiente) y una columna con las y. Devuelve el valor del coste.

Aplicamos la fórmula:

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

```
def gradienteCoste(thetas, mX, cY):
    thetas = np.matrix(thetas).transpose()
    return np.ravel((1/float(len(mX))) * np.dot(mX.transpose(),(sigmoide(np.dot(mX, thetas)) - cY)))
```

Función gradienteCoste

La función gradienteCoste es una función vectorizada que recibe los mismos parámetros que la función anteriormente definida. Aplicando la ecuación normal de descenso de gradiente logístico, devolvemos un vector con nuevos valores de θ que mejoran la función coste.

Aplicamos la fórmula:

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

```
def parametros(thetas, mX, mY):
    result = opt.fmin_tnc(func=coste , x0=thetas, fprime=gradienteCoste , args=(mX, mY))
    return result[0]
```

Función parametros

Se aplica el algoritmo de descenso de gradiente con la función generalizada de la biblioteca SciPy.optimize para devolver los θ que minimizan la función coste.

```
def evaluacion(cThetas, mX, mY):
    h = np.dot(mX, cThetas)
    z = sigmoide(h)
    z = (np.ravel(z) >= 0.5)
    z = (z == np.ravel(mY))
    return np.ravel(sum(z)/float(len(z))*100)[0]
```

Función evaluacion

Recibe como parámetros la columna de θ, los valores de las variables xi de los ejemplos de entrenamiento(mX) y una columna con los valores de y (mY). Con estos datos primero calculamos la función hipótesis de los datos (h).A continuación aplicamos la función sigmoide a cada uno de los valores del vector h , obteniendo las estimaciones de y.

Si dicha estimación es mayor o igual que 0.5, consideramos que fue admitido (aproximamos a z = 1) y en caso contrario , lo redondeamos a 0.

La función devuelve el porcentaje de acierto de los datos reales y frente los z estimados.

```
def graficaFronteraDecision(mX, mY, cThetas):
    pos = np.where(mY == 1)
    pos0 = np.where(mY == 0)
    pl.figure()
    pl.xlabel('Exam 1 score')
    pl.ylabel('Exam 2 score')
    pl.scatter(mX[pos,1], mX[pos,2], marker='+', c='k', label='admitido')
    pl.scatter(mX[pos0,1], mX[pos0,2], marker='o', c='g', label='no admitido')
    pl.legend()

    x1_min, x1_max = mX[:, 1].min(), mX[:, 1].max()
    x2_min, x2_max = mX[:, 2].min(), mX[:, 2].max()
    xx1, xx2 = np.meshgrid(np.linspace(x1_min, x1_max),
                           np.linspace(x2_min, x2_max))
    h = sigmoide(np.c_[np.ones((xx1.ravel().shape[0], 1)),
                       xx1.ravel(), xx2.ravel()].dot(cThetas))
    h = h.reshape(xx1.shape)
    pl.contour(xx1, xx2, h, [0.5], linewidths=1, colors='b')

    pl.savefig('grafica2d.png')
    pl.close()
```

Función graficaFronteraDecisión

Se dibujan los puntos correspondientes a los ejemplos de entrenamiento y la frontera con límite 0.5 de probabilidad.

```python
def comprobacion():
    datos = lee_csv("ex2data1.csv")
    # Quitamos columna Y
    datosX = np.delete(datos, len(datos[0])-1, 1)
    mX = np.c_[np.ones(len(datosX)), datosX]
    mY = (np.array([datos[:,len(datos[0]) - 1]])).transpose()
    thetas = (np.matrix(np.zeros(len(datos[0])))).transpose()
    thetas = parametros(thetas, mX, mY)
    graficaFronteraDecision(mX, mY, thetas)
    print evaluacion(np.matrix(thetas).transpose(), mX, mY)
```

Funcion comprobacion
Es la función principal, en la leemos los ejemplos de entrenamiento del fichero, separamos las variables x y la y, y llamamos a la función parametros, graficaFronteraDecision y evaluacion.

B) REGRESIÓN LOGÍSTICA CON REGULARIZACIÓN

En la segunda parte de la práctica se busca predecir si un microchip pasará o no el control de calidad mediante los resultados de dos tests. Además se crearán variables extras (un total de 28) que son combinaciones lineales de las dos variables que representan los tests. A éstas se les deberán aplicar el método de regularización ya que se trabajan con variables exponenciales de hasta grado 6.

Ediciones sobre el código de la parte anterior:

```python
def coste(cThetas, mX, cY, lamb):
    cThetas = np.matrix(cThetas).transpose()
    gX = sigmoide(np.dot(mX, cThetas))
    s1 = np.dot((np.log(gX)).transpose(), cY)
    s2 = np.dot((np.log(1 - gX)).transpose(), 1 - cY)
    # Es una matriz de 1x1
    vs = np.ravel((-1/float(len(mX)))*(s1 + s2))[0]
    vs = vs + (lamb/2*len(mX)) * np.ravel(np.dot(cThetas.transpose(), cThetas))[0]
    return vs
```

```python
def gradienteCoste(thetas, mX, cY, lamb):
    lThetas = thetas * (lamb/len(mX))
    lThetas[0] = 0
    thetas = np.matrix(thetas).transpose()
    return np.ravel((1/float(len(mX))) *
        np.dot(mX.transpose(),(sigmoide(np.dot(mX, thetas)) - cY))) + lThetas
```

Función coste y gradiente
Se añade al cálculo la operación vectorizada para la regularización de las variables. Esto introduce un nuevo parámetro lambda.

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

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

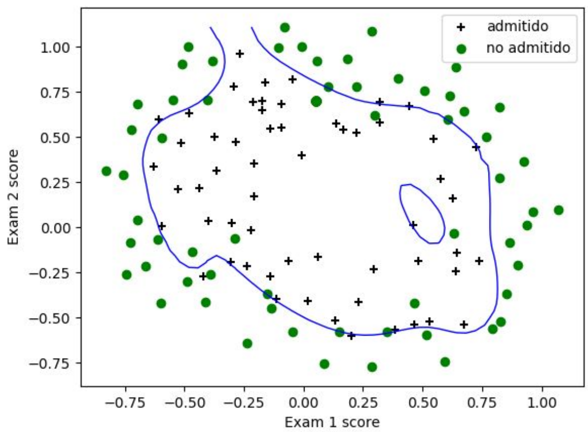
Función comprobacion

Se extienden el número de variables hasta 28 utilizando la clase PolynomialFeatures del paquete SciKit Learn

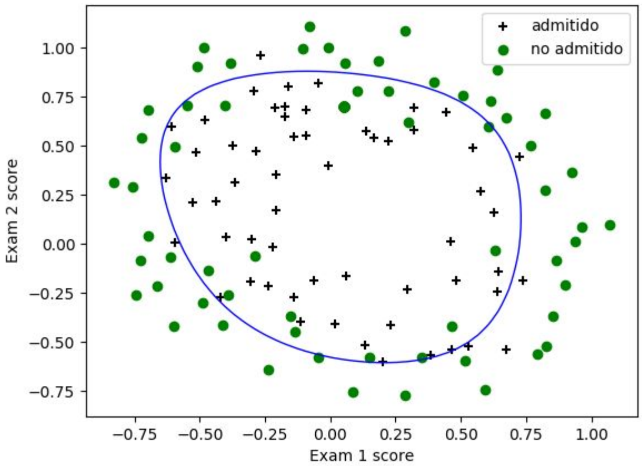
```
def comprobacion():
    datos = lee_csv("ex2data2.csv")
    # Quitamos columna Y
    datosX = np.delete(datos, len(datos[0])-1, 1)
    poly = pf(6)
    datosX = poly.fit_transform(datosX)
    mX = datosX
    mY = (np.array([datos[:,len(datos[0]) - 1]])).transpose()
    thetas = (np.matrix(np.zeros(len(datosX[0])))).transpose()
    thetas = parametros(thetas, mX, mY)
    plot_decisionboundary(mX, mY, thetas, poly)
```

Conclusiones

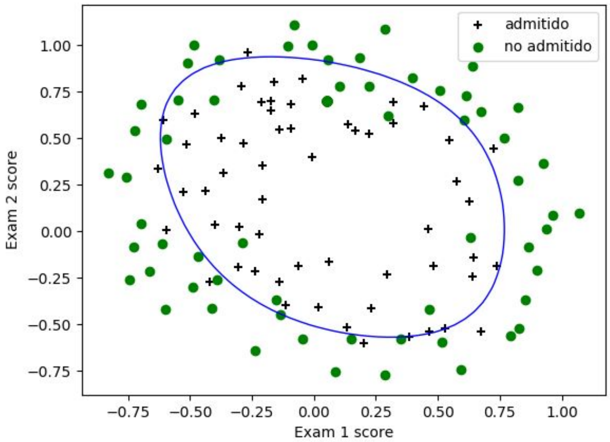
El parámetro λ del término de regularización penaliza la función coste; de forma que si λ es muy pequeño(por ejemplo: λ = 1), la penalización es mínima y la función de aproximación se ajusta demasiado a los ejemplos de entrenamiento. Para λ = 5 , la penalización es mayor y se ajusta menos.



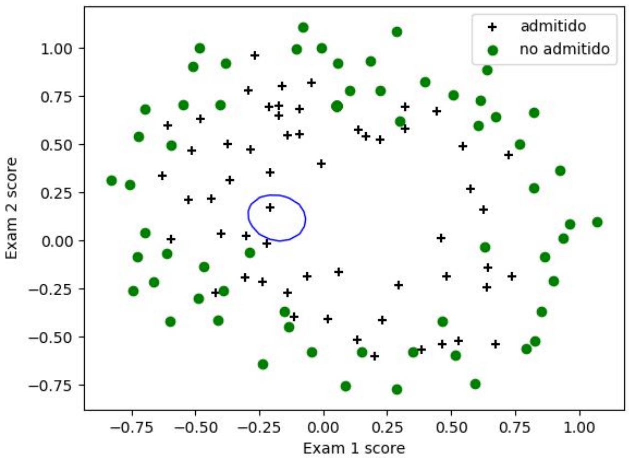
Para λ = 1



Para λ = 1.5



Para λ = 5



Para λ = 200 (En este caso, la penalización es demasiado grande y por tanto la función de estimación no se ajusta apenas a los ejemplos de entrenamiento).