PROYECTO DE APRENDIZAJE AUTOMÁTICO

Tratamiento de dataset mediante las técnicas de regresión logística, redes neuronales y SVMs

Gonzalo Sanz Lastra y Jorge Rodríguez García

Índice de contenidos:

1- Tratamiento del dataset

Para el proyecto hemos elegido un dataset con información sobre juegos presentes en la plataforma Steam. El dataset cuenta con un total de 27075 juegos (m), con 17 atributos (n) cada uno.
Estos atributos son: id de la aplicación, nombre del juego, fecha de lanzamiento, idioma, desarrolladora, publisher, plataformas, edad requerida, categoría, género, tags de steam, logros, votos positivos, votos negativos, tiempo de juego promedio, tiempo de juego medio, número de propietarios y precio.

Como Y a predecir, elegimos el atributo “número de propietarios”, sobre el que construimos una nueva columna que representa si el juego tiene éxito (1) o no (0). Consideramos que un juego tiene éxito si supera la media total de propietarios. Esto vendría a ser como la “fama” del juego, ya que un juego con muchos votos negativos se considerará exitoso según este baremo, puesto que para tener tantos votos, habrá sido comprado por mucha gente.

En una primera instancia, redujimos el número de atributos a los 11 que creíamos que mejor describían el dataset para poder inferir nuestras Ys. Sin embargo, obtuvimos un porcentaje relativamente bajo al aplicar la primera de las técnicas de aprendizaje automático, regresión logística, logrando sacar tan solo un 64% de aciertos.

Como consecuencia, planteamos varias soluciones, ya que intuimos que el problema estaba en el tratamiento del dataset. Una de las posibilidades fue aumentar el número de clases de nuestras Ys, dividiendo en intervalos de propietarios en vez de tener una solución binaria de “éxito / no éxito”.
La otra fue reducir aún más el número de atributos de nuestro dataset, que pasó de los anteriores 11 a los 6 atributos más descriptivos del dataset. Con estos 6 atributos, obtuvimos un porcentaje del 88% con la misma técnica, por lo que dimos por solucionado el problema, al haber quitado dimensiones sobrantes que metían ruido en el dataset y no aportaban verdadera información.
Los 6 atributos sobre los cuales trabajamos son: fecha de lanzamiento, plataforma, género, votos positivos, votos negativos y el precio, usando el número de propietarios como Ys.

A continuación, necesitábamos procesar varios de los atributos para obtener valores numéricos, ya que muchas de las columnas del dataset son cadenas. Para esto, codificamos nuestros datos de la siguiente manera:

-La fecha de lanzamiento solo tiene en cuenta el año, no el mes ni el día.
-Las plataformas en las que se encuentra el juego son: windows + mac + linux (1), windows (2), otras (3).
-Género: escogimos los 10 géneros más recurrentes del dataset para codificarlos, dejando como valor 0 por defecto para “otros géneros”. Estos géneros principales están codificados mediante números primos, de forma que siempre se pueda saber a qué géneros pertenece un juego descomponiendo en números primos el valor de su columna de géneros.

> Ejemplo: un juego de acción / RPG tendrá en su columna el valor 7 + 13 = 20, que ninguna otra combinación de géneros puede imitar.

DataReader.py:
Este archivo tiene métodos para cargar/guardar una matriz como csv, quitar las columnas de una matriz, pasar los strings de nuestro conjunto de datos a números y otro para crear la columna de Ys a partir del número de propietarios.

```
def load_csv(file_name):
    """carga el fichero csv especificado y lo devuelve en un array de
numpy"""
    return read_csv(file_name, header=None).values

def save_csv(file_name, array):
    """guarda el array especificado y lo vuelca como csv"""
    np.savetxt(file_name, array, delimiter=",")

def reduceData(data, columns):
    """quita de "data" las columnas establecidas por "columns" """
    for i in range(columns.shape[0]):
        data = np.delete(data, columns[i], 1)

    return data

def dataToNumbers(data):
    """ pasa todos los datos que no sean numeros a valores numericos"""

    for i in range(data.shape[0]):
        for j in range(data.shape[1]):

            # release date
            if j == 0:
                data[i, j] = data[i, j][:4]

            # platform
            elif j == 1:
                if data[i, j] == 'windows;mac;linux':
                    data[i, j] = 1
                elif data[i, j] == 'windows':
                    data[i, j] = 2
                else:
                    data[i, j] = 3

            # genre
```

```
            elif j == 2:
                aux = 0
                if data[i,j].find('Action') != -1:
                    aux += 7
                if data[i,j].find('Strategy') != -1:
                    aux += 11
                if data[i,j].find('RPG') != -1:
                    aux += 13
                if data[i,j].find('Casual') != -1:
                    aux += 17
                if data[i,j].find('Simulation') != -1:
                    aux += 19
                if data[i,j].find('Racing') != -1:
                    aux += 23
                if data[i,j].find('Adventure') != -1:
                    aux += 29
                if data[i,j].find('Sports') != -1:
                    aux += 31
                if data[i,j].find('Indie') != -1:
                    aux += 37

                data[i, j] = aux
                #Action Strategy RPG Casual Simulation Racing Adventure
Sports Indie

            # owners mean
            elif j == 5:
                index = data[i,j].find('-')
                minimum = float(data[i,j][:index])
                maximum = float(data[i,j][index+1:])
                data[i, j] = (minimum + maximum)/2

            data[i, j] = float(data[i, j])

    return data

def createY(owners):
    """crea las Ys sobre los datos de los propietarios: si el juego
tiene mas
```

```
    propietarios que la media de propietarios, se considera exitoso (y
= 1); si no,
    se considera no exitoso (y = 0)"""

    L = owners.shape[0]
    mean = owners.sum()/L
    aux = (owners > mean)
    return aux.astype(int)

def main():
    X = load_csv("steam.csv")
    #columnas que desechamos
    X = reduceData(X, np.array([15, 14, 11, 10, 8, 7, 5, 4, 3, 1, 0]))

    #print de todos los tags (lo que significa cada columna)
    tags = X[0, :]
    print(tags)

    X = np.delete(X, 0, 0) # quitamos tags
    X = dataToNumbers(X)
    X = np.random.permutation(X) # desordenamos aleatoriamente
    Y = createY(X[:, 5])[np.newaxis].T
    X = np.delete(X, 5, 1) # borramos columna de owners
    X = np.append(X, Y, 1) # añadimos la transformacion (Ys) al final

    save_csv("steamReduced.csv", X)

main()
```

ValsLoader.py:
Archivo donde se tratan los valores, con métodos para normalizar una matriz, para polinomizarla y otro para leer y separar los valores del dataset en Xs e Ys de entrenamiento, validación y testeo.

```
def loadValues(file_name):
    """lee valores y los procesa, dividiendolos en datos de
entrenamiento, validacion y test"""

    valores = load_csv(file_name)
```

```
    totalX = valores[:, :valores.shape[1] - 1]
    totalY = valores[:, valores.shape[1] - 1][np.newaxis].T

    Xlength = int(totalX.shape[0] * 0.6)
    XvalLength = int(totalX.shape[0] * 0.2)

    X = totalX[:Xlength, :]
    Y = totalY[:Xlength, :]

    Xval = totalX[Xlength:XvalLength + Xlength, :]
    Yval = totalY[Xlength:XvalLength + Xlength, :]

    Xtest = totalX[XvalLength + Xlength:, :]
    Ytest = totalY[XvalLength + Xlength:, :]

    return X, Y, Xval, Yval, Xtest, Ytest

def polynomize(X, p):
    """polinomiza X con grado p"""
    poly = preprocessing.PolynomialFeatures(p)
    return poly.fit_transform(X) # añade automaticamente la columna de
1s

def normalize(X):
    """normalizacion de escalas, para cuando haya mas de un atributo"""
    mu = X.mean(0)[np.newaxis]   # media de cada columna de X
    sigma = X.std(0)[np.newaxis] # desviacion estandar de cada columna
de X

    X_norm = (X - mu)/sigma

    return X_norm, mu, sigma

def normalizeValues(valoresPrueba, mu, sigma):
    """normaliza los valores de prueba con la mu y sigma de los
atributos X (al normalizarlos)"""
    return (valoresPrueba - mu)/sigma
```

2- Regresión logística

Aquí quisimos seguir el mismo tratamiento de los datos que se explicó en clase: teniendo diferentes Xs de entrenamiento (60% de los datos), validación (20%) y testeo (20%). Esta separación de los datos es siempre aleatoria, ya que el dataset es reordenado aleatoriamente en cada ejecución (así nos evitamos coger los primeros juegos siempre para el entrenamiento, etc.). Estos datos son polinomizados a grado 2 y posteriormente normalizados.
Los de entrenamiento son utilizados para aplicar la regresión logística, los de validación para conseguir la mejor lambda (factor de regularización) y los de testeo para comprobar la efectividad del entrenamiento.

Para determinar qué combinación de dos columnas es la más efectiva / precisa para describir el dataset e inferir nuestras Ys, y poder verlo gráficamente (con más de dos dimensiones es imposible), probamos todas las combinaciones entre éstas de 2 en 2 (polinomizando con grado 6), hallando el porcentaje de aciertos obtenidos y sacando por pantalla la gráfica correspondiente.
En un principio creímos que los dos atributos más descriptivos serían el género y el precio, sin embargo, y para nuestra sorpresa, el algoritmo concluye que las columnas más importantes son la fecha de salida del juego y el número de votos negativos que éste tiene.

Con estas dos columnas se obtiene un porcentaje de acierto del 88.01477%.
Si se hace uso de todas las columnas para entrenar, se obtiene una ligera mejora que lleva el porcentaje al 88.21791%.

LogisticRegresion.py:
Tiene funciones que pintan un gráfico donde se muestran los dos valores más decisivos del conjunto de datos, diferenciando si tenían éxito o no, para marcar el contorno que los separa. Tiene también otro método para dibujar la curva de aprendizaje de lambda, además del propio algoritmo de regresión logística, y funciones para decir qué columnas son las más decisivas y qué porcentaje de acierto tienen. También cuenta con métodos de la función sigmoide, el gradiente, el coste etc.

```
def graphics(X, Y, O, poly):
    plot_decisionboundary(X, Y, O, poly)

    # 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] , c = 'blue', s=2)

    # Obtiene un vector con los índices de los ejemplos negativos
    pos = np.where(Y == 0)
    # Dibuja los ejemplos negativos
```

```
    plt.scatter(X[pos, 0] ,X[pos, 1] , c = 'red', s=2)

    plt.show()

def plot_decisionboundary(X, Y, theta, poly):
    """pinta el polinomio que separa los datos entre los que cumplen el
requisito y los que no"""
    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)) # grid de cada columna de Xs

    h = sigmoid(poly.fit_transform(np.c_[xx1.ravel(),
xx2.ravel()]).dot(theta))# ravel las pone una tras otra

    h = h.reshape(xx1.shape)

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

# g(X*Ot) = h(x)
def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def lambdaGraphic(errorX, errorXVal, lambdas):
    """pinta la curva de aprendizaje de lambda"""
    plt.figure()
    plt.plot(lambdas, errorX)
    plt.plot(lambdas, errorXVal)
    plt.show()

def coste(X, Y, O, reg):
    """devuelve la funcion de coste, dadas X, Y, y thetas"""
    O = O[np.newaxis]
    AuxO = O[:, 1:]

    H = sigmoid(np.dot(X, O.T))
    log1 = np.log(H).T
```

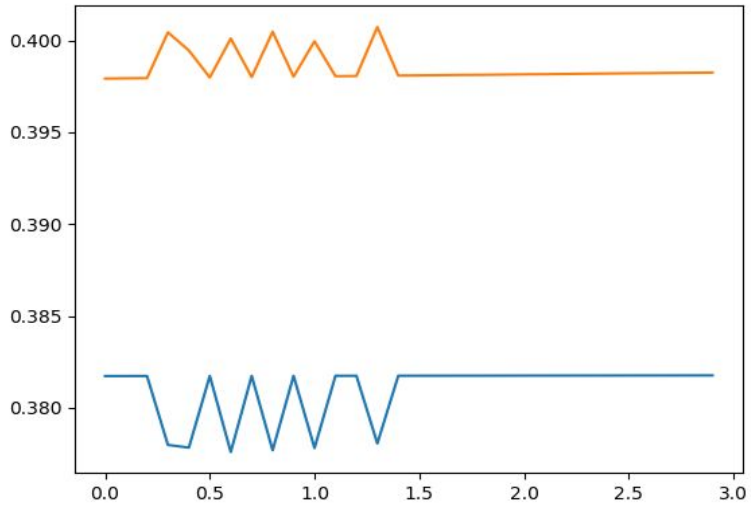
```
    Aux1 = np.dot(log1, Y)

    log2 = np.log(1 - H).T
    Aux2 = np.dot(log2, (1-Y))

    cost = (Aux1 + Aux2)/(len(X))

    return (-cost + (AuxO**2).sum()*reg/(2*X.shape[0]))[0, 0]

def gradiente(X, Y, O, reg):
    """la operacion que hace el gradiente por dentro -> devuelve un
vector de valores"""
    AuxO = np.hstack([np.zeros([1]), O[1:,]])
    O = O[np.newaxis]
    AuxO = AuxO[np.newaxis].T

    return ((X.T.dot(sigmoid(np.dot(X, O.T))-Y))/X.shape[0] +
(reg/X.shape[0])*AuxO)

def minimizeFunc(O, X, Y, reg):
    return (coste(X, Y, O, reg), gradiente(X, Y, O, reg))

def successPercentage(X, Y, O):
    """determina el porcentaje de aciertos comparando los resultados
estimados con los resultados reales"""
    results = (sigmoid(X.dot(O[np.newaxis].T)) >= 0.5)
    results = (results == Y)
    return results.sum()/results.shape[0]

def bestColumns(X, Y, Xval, Yval, Xtest, Ytest):
    """devuelve la mejor combinacion de columnas de X (las que obtienen
un mejor porcentaje de acierto), mostrando
    la grafica y el porcentaje de acierto obtenido de cada
combinacion"""
    mostSuccessfull = 0
    bestRow = 0
    bestColumn = 0

    for x in range(0, 6):
        for y in range(0, 6):
```

```
            print("Row: " + str(x))
            print("Column: " + str(y))
            Xaux = np.vstack([X[:, x][np.newaxis], X[:,
y][np.newaxis]]).T
            XvalAux = np.vstack([Xval[:, x][np.newaxis], Xval[:,
y][np.newaxis]]).T
            XtestAux = np.vstack([Xtest[:, x][np.newaxis], Xtest[:,
y][np.newaxis]]).T
            samples = np.random.choice(X.shape[0], 400)

            O, poly, success = logisticRegresion(Xaux, Y, XvalAux,
Yval, XtestAux, Ytest, 6)
            if(success > mostSuccessfull):
                mostSuccessfull = success
                bestRow = x
                bestColumn = y

            Xaux = Xaux[samples,:]
            Yaux = Y[samples]
            print(Xaux.shape)
            graphics(Xaux, Yaux, O, poly)

    return bestRow, bestColumn, mostSuccessfull

def logisticRegresion(X, Y, Xval, Yval, Xtest, Ytest, polyGrade):
    """aplica regresion logistica sobre un conjunto de datos,
entrenando con una seccion de entrenamiento,
    eligiendo la mejor lambda con una seccion de validacion, y probando
los resultados obtenidos (porcentaje
    de acierto) con una seccion de test"""

    poly = preprocessing.PolynomialFeatures(polyGrade)

    Xpoly = polynomize(X, polyGrade)                    # pone
automaticamente columna de 1s
    Xnorm, mu, sigma = normalize(Xpoly[:, 1:]) # se pasa sin la columna
de 1s (evitar division entre 0)
    Xnorm = np.hstack([np.ones([Xnorm.shape[0], 1]), Xnorm]) # volvemos
a poner columna de 1s
```

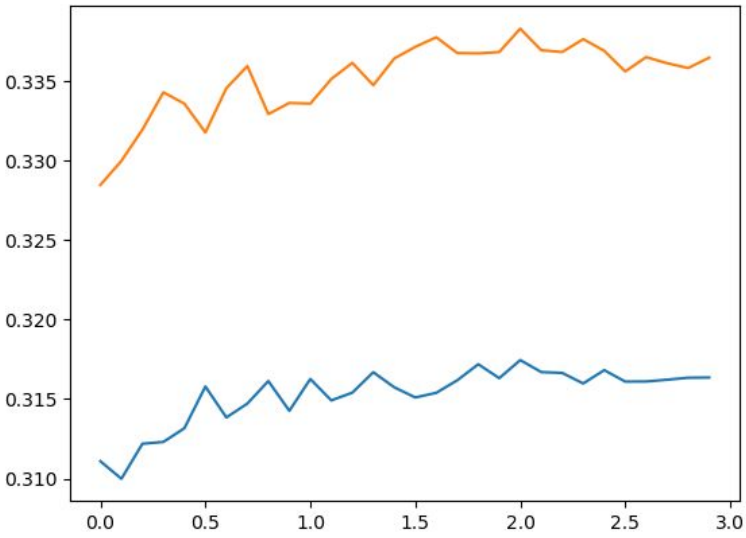
```
    XpolyVal = polynomize(Xval, polyGrade)
    XnormVal = normalizeValues(XpolyVal[:, 1:], mu, sigma)
    XnormVal = np.hstack([np.ones([XnormVal.shape[0], 1]), XnormVal])

    XpolyTest = polynomize(Xtest, polyGrade)
    XnormTest = normalizeValues(XpolyTest[:, 1:], mu, sigma)
    XnormTest = np.hstack([np.ones([XnormTest.shape[0], 1]),
XnormTest])

    m = Xnorm.shape[0]      # numero de muestras de entrenamiento
    n = Xnorm.shape[1]      # numero de variables x que influyen en el
resultado y, mas la columna de 1s

    thetaVec = np.zeros([n])

    l =  np.arange(0, 3, 0.1)

    errorX = np.zeros(l.shape[0])
    errorXVal = np.zeros(l.shape[0])

    # errores para cada valor de lambda
    for i in range(l.shape[0]):
        result = opt.minimize(fun = minimizeFunc, x0 = thetaVec,
        args = (Xnorm, Y, l[i]), method = 'TNC', jac = True, options =
{'maxiter':70})
        O = result.x

        errorX[i] = coste(Xnorm, Y, O, l[i])
        errorXVal[i] = coste(XnormVal, Yval, O, l[i])

    lambdaGraphic(errorX, errorXVal, l)

    # lambda que hace el error minimo en los ejemplos de validacion
    lambdaIndex = np.argmin(errorXVal)
    print("Best lambda: " + str(l[lambdaIndex]))

    # thetas usando la lambda que hace el error minimo (sobre ejemplos
de entrenamiento)
    result = opt.minimize(fun = minimizeFunc, x0 = thetaVec,
```

```
        args = (Xnorm, Y, l[lambdaIndex]), method = 'TNC', jac = True,
options = {'maxiter':70})

    O = result.x

    success = successPercentage(XnormTest, Ytest, O)
    print("Porcentaje de acierto: " + str(success*100) + "%")

    return O, poly, success

def main():
    # dataset
    X, Y, Xval, Yval, Xtest, Ytest = loadValues("steamReduced.csv")

    bestRow, bestColumn, mostSuccessfull = bestColumns(X, Y, Xval,
Yval, Xtest, Ytest)
    # mejor combinacion de columnas (mejor porcentaje de acierto)
    print("Best Row: " + str(bestRow))
    print("Best Column: " + str(bestColumn))
    print("Most Successfull: " + str(mostSuccessfull*100))

    # regresion logistica con todas las columnas
    logisticRegresion(X, Y, Xval, Yval, Xtest, Ytest, 2)

main()
```

Muestras del aprendizaje de lambda con diferentes combinaciones de columnas:

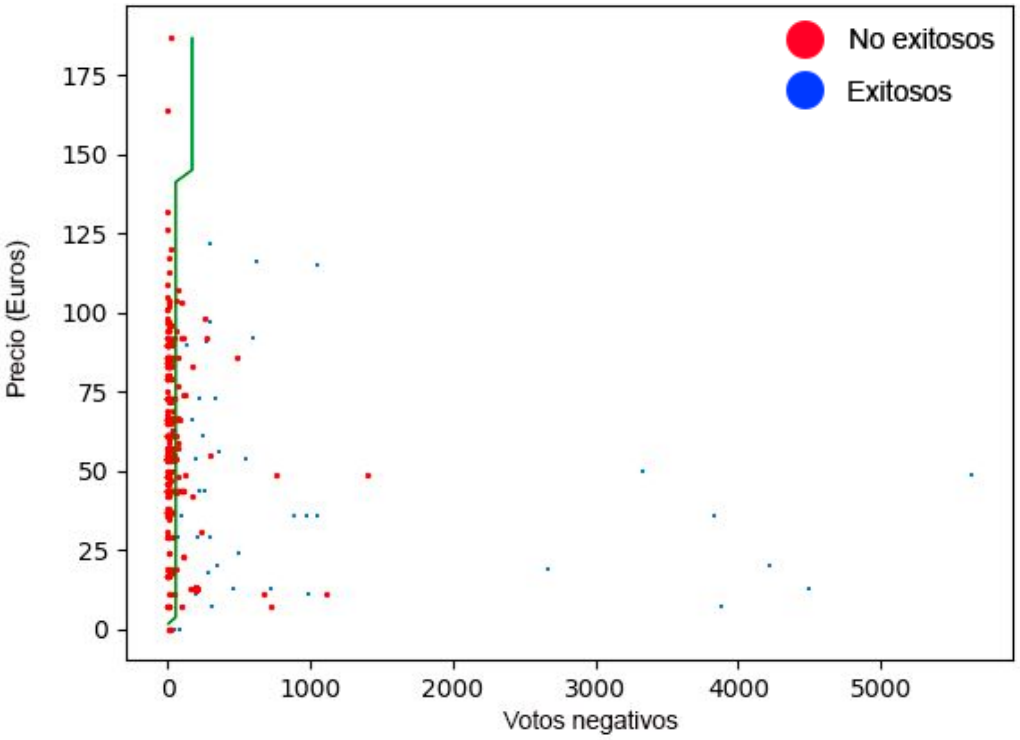


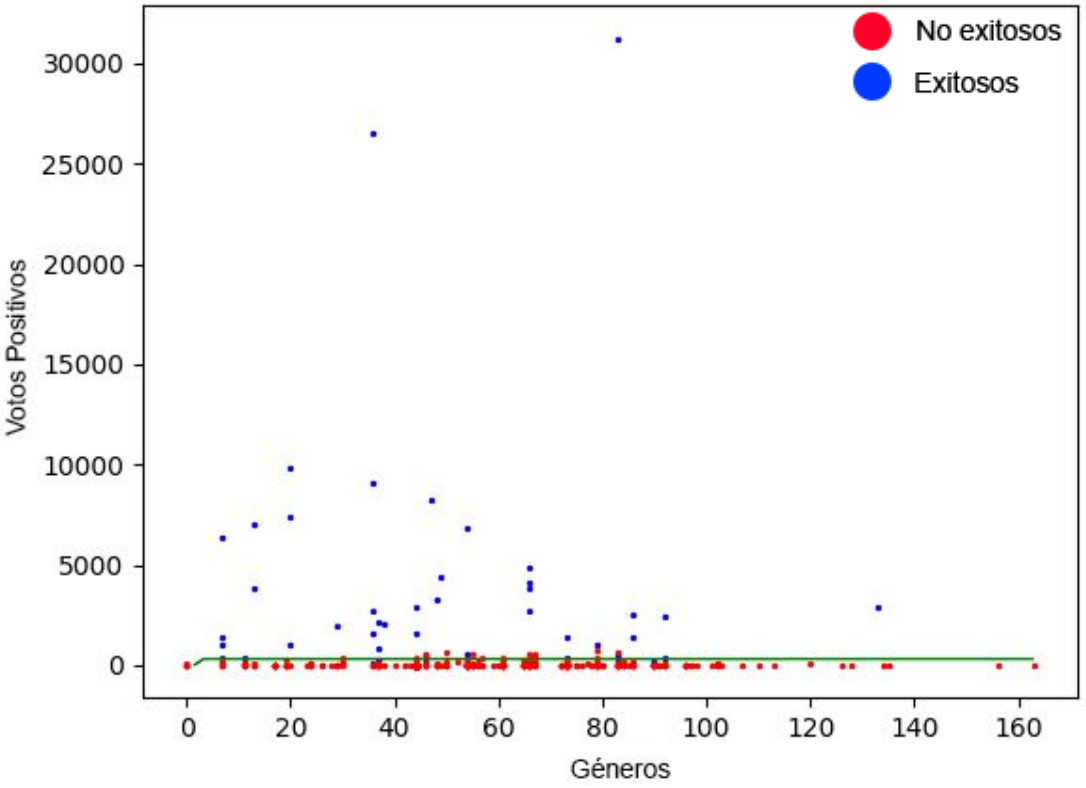
Curva con las columnas de precio y fecha de salida.

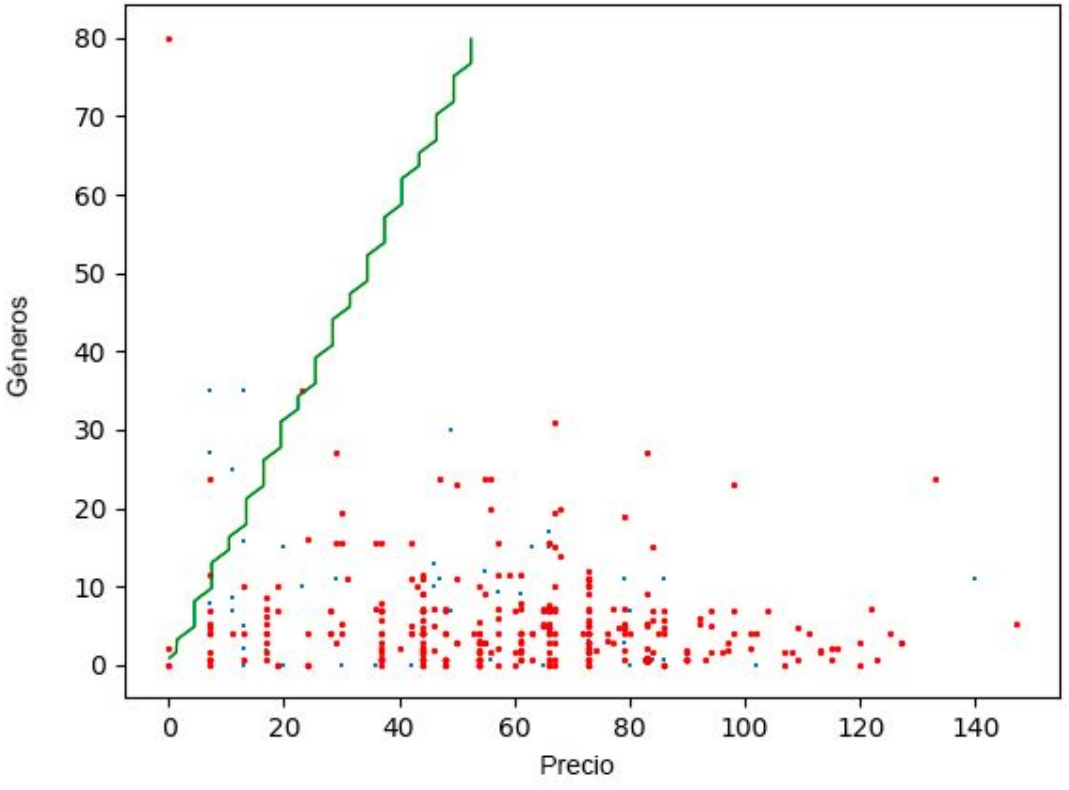


Curva con las columnas de precio y géneros.

Muestras del contorno con diferentes combinaciones de columnas:




No exitosos
Exitosos
Votos Positivos
Géneros
30000
25000
20000
15000
10000
5000
0
0
20
40
60
80
100
120
140
160


Géneros
Precio
80
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140

3- Redes neuronales:

Usando la misma separación de datos que antes, utilizamos un porcentaje de los mismos para entrenar la red neuronal, y el restante para probar su efectividad. Los datos fueron polinomizados a grado 3 y posteriormente normalizados.

De esta forma construimos una red neuronal de tres capas de (m x n) datos de entrada (Xs), (m x 40) en la capa intermedia u oculta y (m x numEtiquetas) en la capa de salida (h), siendo numEtiquetas = 2 (columna de Y para la clase 0, no éxito, y columna de Y para la clase 1, éxito). Para ello, precisamos de dos Thetas:

theta1 (n x 40), theta2 (40 x numEtiquetas)

Al igual que en el método anterior, calculamos qué combinación de dos columnas es la más representativa, obteniendo el mismo resultado: fecha de salida y número de votos negativos, obteniendo una precisión del 93.25946445060018%.
Utilizando todos los atributos, adquirimos una precisión ligeramente superior de 94.58910433979686%

NeuronalNetworks.py:
Tiene los métodos necesarios para entrenar una red neuronal: inicialización de pesos aleatorios, forward propagation, backward propagation, etc. vistos en clase, y un método que determina la combinación de atributos que genera un mayor porcentaje de acierto. Después de entrenar la red neuronal, guardamos las thetas entrenadas en archivos .csv para poder ser leídas en caso de querer aplicarlas en otros .py, y no tener que volver a entrenar la red neuronal (últimos apartados de la memoria).

```
import numpy as np
import scipy.optimize as opt              # para la funcion de gradiente
from matplotlib import pyplot as plt      # para dibujar las graficas
from valsLoader import *
from dataReader import save_csv

# g(X*Ot) = h(x)
def sigmoid(Z):
    return 1/(1+np.exp(-Z))

def dSigmoid(Z):
    return sigmoid(Z)*(1-sigmoid(Z))

def pesosAleatorios(L_in, L_out, rango):
    O = np.random.uniform(-rango, rango, (L_out, 1+L_in))
    return O
```

```
def cost(X, Y, O1, O2, reg):
    """devuelve un valor de coste"""

    a = -Y*(np.log(X))
    b = (1-Y)*(np.log(1-X))
    c = a - b
    d = (reg/(2*X.shape[0]))* ((O1[:,1:]**2).sum() +
(O2[:,1:]**2).sum())
    return ((c.sum())/X.shape[0]) + d

def neuronalSuccessPercentage(results, Y):
    """determina el porcentaje de aciertos de la red neuronal
comparando los resultados estimados con los resultados reales"""
    numAciertos = 0

    for i in range(results.shape[0]):
        result = np.argmax(results[i])
        if result == Y[i]: numAciertos += 1

    return (numAciertos/(results.shape[0]))*100

def forPropagation(X1, O1, O2):
    """propaga la red neuronal a traves de sus dos capas"""
    m = X1.shape[0]
    a1 = np.hstack([np.ones([m, 1]), X1])
    z2 = np.dot(a1, O1.T)
    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])
    z3 = np.dot(a2, O2.T)
    h = sigmoid(z3)

    return a1, z2, a2, z3, h

def backPropAlgorithm(X, Y, O1, O2, num_etiquetas, reg):
    G1 = np.zeros(O1.shape)
    G2 = np.zeros(O2.shape)

    m = X.shape[0]
    a1, z2, a2, z3, h = forPropagation(X, O1, O2)

    for t in range(X.shape[0]):
```

```
        a1t = a1[t, :] # (1, 401)
        a2t = a2[t, :] # (1, 26)
        ht = h[t, :] # (1, 10)
        yt = Y[t] # (1, 10)
        d3t = ht - yt # (1, 10)
        d2t = np.dot(O2.T, d3t) * (a2t * (1 - a2t)) # (1, 26)

        G1 = G1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])
        G2 = G2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    AuxO2 = O2
    AuxO2[:, 0] = 0

    G1 = G1/m
    G2 = G2/m + (reg/m)*AuxO2

    return np.concatenate((np.ravel(G1), np.ravel(G2)))


def backPropagation(params_rn, num_entradas, num_ocultas,
num_etiquetas, X, Y, reg):
    O1 = np.reshape(params_rn[:num_ocultas*(num_entradas + 1)],
(num_ocultas, (num_entradas+1)))
    O2 = np.reshape(params_rn[num_ocultas*(num_entradas+1):],
(num_etiquetas, (num_ocultas+1)))

    c = cost(forPropagation(X, O1, O2)[4], Y, O1, O2, reg)
    gradient = backPropAlgorithm(X,Y, O1, O2, num_etiquetas, reg)

    return c, gradient

def neuronalNetwork(X, Y, Xtest, Ytest, polyGrade):
    """aplica redes neuronales sobre un conjunto de datos, entrenando
con una seccion de entrenamiento,
    y probando los resultados obtenidos (porcentaje de acierto) con una
seccion de test"""
    poly = preprocessing.PolynomialFeatures(polyGrade)

    Xpoly = polynomize(X, polyGrade)                    # pone
automaticamente columna de 1s
```

```
    Xnorm, mu, sigma = normalize(Xpoly[:, 1:]) # se pasa sin la columna
de 1s (evitar division entre 0)

    XpolyTest = polynomize(Xtest, polyGrade)
    XnormTest = normalizeValues(XpolyTest[:, 1:], mu, sigma)

    m = Xnorm.shape[0]      # numero de muestras de entrenamiento
    n = Xnorm.shape[1]      # numero de variables x que influyen en el
resultado y, mas la columna de 1s

    num_etiquetas = 2
    l = 1

    AuxY = np.zeros((m, num_etiquetas))
    for i in range(m):
        AuxY[i][int(Y[i])] = 1

    capaInter = 40
    O1 = pesosAleatorios(Xnorm.shape[1], capaInter, 0.12)
    O2 = pesosAleatorios(capaInter, num_etiquetas, 0.12)
    thetaVec = np.append(O1, O2).reshape(-1)

    result = opt.minimize(fun = backPropagation, x0 = thetaVec,
     args = (n, capaInter, num_etiquetas, Xnorm, AuxY, l), method =
'TNC', jac = True, options = {'maxiter':70})

    O1 = np.reshape(result.x[:capaInter*(n + 1)], (capaInter, (n+1)))
    O2 = np.reshape(result.x[capaInter*(n+1):], (num_etiquetas,
(capaInter+1)))

    success = neuronalSuccessPercentage(forPropagation(XnormTest, O1,
O2)[4], Ytest)
    print("Neuronal network success: " + str(success) + " %")

    return O1, O2, poly, success

def bestColumns(X, Y, Xtest, Ytest):
    """devuelve la mejor combinacion de columnas de X (las que obtienen
un mejor porcentaje de acierto), mostrando
    el porcentaje de acierto obtenido de cada combinacion"""
```

```
    mostSuccessfull = 0
    bestRow = 0
    bestColumn = 0

    for x in range(0, 6):
        for y in range(0, 6):
            print("Row: " + str(x))
            print("Column: " + str(y))
            Xaux = np.vstack([X[:, x][np.newaxis], X[:,
y][np.newaxis]]).T
            XtestAux = np.vstack([Xtest[:, x][np.newaxis], Xtest[:,
y][np.newaxis]]).T

            O1, O2, poly, success = neuronalNetwork(Xaux, Y, XtestAux,
Ytest, 3)
            if(success > mostSuccessfull):
                mostSuccessfull = success
                bestRow = x
                bestColumn = y
    return bestRow, bestColumn, mostSuccessfull

def main():
    X, Y, Xval, Yval, Xtest, Ytest = loadValues("steamReduced.csv")

    bestRow, bestColumn, mostSuccessfull = bestColumns(X, Y, Xtest,
Ytest)
    # mejor combinacion de columnas (mejor porcentaje de acierto)
    print("Best Row: " + str(bestRow))
    print("Best Column: " + str(bestColumn))
    print("Most Successfull: " + str(mostSuccessfull))

    # redes neuronales con todas las columnas
    O1, O2, poly, success = neuronalNetwork(X, Y, Xtest, Ytest, 3)

    save_csv("O1.csv", O1)
    save_csv("O2.csv", O2)

main()
```

4- SVM (support vector machines)

Para esta técnica, dividimos los datos como antes para entrenar la SVM, elegir los valores óptimos de C y sigma (datos de validación) y para probar nuestros resultados (test), todos ellos normalizados.
El porcentaje de aciertos obtenidos mediante esta técnica es de un 95.01385%

SVM.py:
Con un método para entrenar una SVM recibiendo los datos separados por secciones y el grado al que se quieren polinomizar. Elige la mejor combinación de C y sigma del mismo modo visto en clase. Una vez entrenada la SVM, guardamos el objeto clf en un archivo .csv para poder ser accedido de manera más directa en el futuro desde otros archivos .py, y no tener que volver a entrenar la SVM (últimos apartados de la memoria).

```
def SVM(X, Y, Xval, Yval, Xtest, Ytest, polyGrade):
    """aplica SVM sobre un conjunto de datos, entrenando con una
seccion de entrenamiento,
    eligiendo las mejores C y sigma con una seccion de validacion, y
probando los resultados obtenidos (porcentaje
    de acierto) con una seccion de test"""

    Xpoly = polynomize(X, polyGrade)                    # pone
automaticamente columna de 1s
    Xnorm, mu, sigma = normalize(Xpoly[:, 1:]) # se pasa sin la columna
de 1s (evitar division entre 0)

    XpolyVal = polynomize(Xval, polyGrade)
    XnormVal = normalizeValues(XpolyVal[:, 1:], mu, sigma)

    XpolyTest = polynomize(Xtest, polyGrade)
    XnormTest = normalizeValues(XpolyTest[:, 1:], mu, sigma)

    Y = Y.ravel()

    C = 0.01
    sigma = 0.01

    # mejores valores (resultado de probar lo de debajo, de esta forma
no tenemos que ejecutarlo siempre)
    bestC = 66.0
    bestSigma = 2.5
```

```
    # probamos la mejor combinacion de C y sigma que de el menor error
sobre los ejemplos de validacion
    """maxCorrects = 0
    for i in range(8):
        C = C * 3
        sigma = 0.01
        for j in range(8):
            sigma = sigma * 3
            clf = SVC(kernel='rbf', C=C, gamma= 1/(2*sigma**2))
            clf.fit(Xnorm, Y)
            corrects = (Yval[:, 0] == clf.predict(XnormVal)).sum()

            if maxCorrects < corrects:
                maxCorrects = corrects
                bestC = C
                bestSigma = sigma

    print(bestC)
    print(bestSigma)"""

    clf = SVC(kernel='rbf', C=bestC, gamma= 1/(2*bestSigma**2))
    clf.fit(Xnorm, Y)

    corrects = (Ytest[:, 0] == clf.predict(XnormTest)).sum()
    print((corrects / Xtest.shape[0])*100)

    return clf

def main():
    X, Y, Xval, Yval, Xtest, Ytest = loadValues("steamReduced.csv")

    clf = SVM(X, Y, Xval, Yval, Xtest, Ytest, 1)
    dump(clf, 'clf.joblib')

main()
```

5- Would be successful?

Una vez tratado nuestro dataset con las distintas técnicas de aprendizaje automático vistas en la asignatura, decidimos hacer un pequeño programa al que introducirle datos sobre juegos, ya sean reales o completamente inventados, y que infiera sus probabilidades de éxito.

Para ello, decidimos utilizar las thetas entrenadas anteriormente mediante redes neuronales, aunque podrían utilizarse los resultados de cualquiera de las otras técnicas aplicadas. No hace falta entrenar en cada ejecución, sino que hemos entrenado cada técnica una vez y guardado sus resultados (thetas y objeto clf) en .csv.

WouldBeSuccessful.py:

```
def main():
    O1 = load_csv("O1.csv")
    O2 = load_csv("O2.csv")
    myX = np.array([])

    print("Introduce los parámetros de tu juego: ")

    print("Fecha de salida (año): ")
    myX = np.append(myX, input())
    print("Plataforma (windows;mac;linux, windows): ")
    myX = np.append(myX, input())
    print("Género (Action, Strategy, RPG, Casual, Simulation, Racing,
Adventure, Sports, Indie): ")
    myX = np.append(myX, input())
    print("Votos positivos: ")
    myX = np.append(myX, input())
    print("Votos negativos: ")
    myX = np.append(myX, input())
    myX = np.append(myX, '0.0')
    print("Precio (euros): ")
    myX = np.append(myX, input())

    myX = myX[np.newaxis]
    myX = dataToNumbers(myX)
    myX = np.delete(myX, 5, 1) # borramos columna de owners
    myX = myX.astype(np.float)
```

```
    print("Probabilidad de éxito: " + str(forPropagation(myX, O1,
O2)[4][0,1] * 100) + "%")


main()
```

Ejemplos de algunos videojuegos:

- Counter-Strike Global offensive, incluido en el dataset de Steam:

```
Introduce los parámetros de tu juego:
Fecha de salida (año):
2010
Plataforma (windows;mac;linux, windows):
windows;mac;linux
Género (Action, Strategy, RPG, Casual, Simulation, Racing, Adventure, Sports, Indie):
Action
Votos positivos:
124710
Votos negativos:
3339
Precio (euros):
7.19
Probabilidad de éxito: 99.99997154128343%
```

- Just Dance 2015, no incluido en el dataset de Steam:

```
Introduce los parámetros de tu juego:
Fecha de salida (año):
2015
Plataforma (windows;mac;linux, windows):
windows
Género (Action, Strategy, RPG, Casual, Simulation, Racing, Adventure, Sports, Indie):
Casual
Votos positivos:
60
Votos negativos:
600
Precio (euros):
60
Probabilidad de éxito: 91.02622431233799%
```

- Juego con datos inventados:

```
Introduce los parámetros de tu juego:
Fecha de salida (año):
1995
Plataforma (windows;mac;linux, windows):
other
Género (Action, Strategy, RPG, Casual, Simulation, Racing, Adventure, Sports, Indie):
ActionRPGCasual
Votos positivos:
100
Votos negativos:
150
Precio (euros):
10
Probabilidad de éxito: 8.56765716337066%
```

6- Conclusión

Tras la realización del trabajo, los problemas que se nos han presentado y cómo los hemos solucionado, podemos decir que una de las partes más importantes y a la vez más difíciles es la elección y el tratamiento de tu dataset.

En un principio, las técnicas de aprendizaje automático utilizadas durante las prácticas nos resultaron complejas, pero con el dataset y los parámetros adecuados para cada una de ellas, era relativamente más “fácil” dar con una solución fija que considerábamos como buena.
Ahora bien, cuando empezamos con el trabajo, nos vimos algo abrumados sin esa linealidad por objetivos que planteaban las prácticas, teniendo “demasiada” libertad a la hora de elegir el dataset, como íbamos a tratarlo, qué información queríamos elegir como resultado, qué dimensiones eran las más idóneas para inferirlo, etc.

Nuestro problema principal durante todo el trabajo ha sido precisamente este, y podríamos haberlo abordado de distintas maneras (eligiendo otras Ys, convirtiendo el problema en un resultado multiclase, reduciendo aún más los atributos, … incluso llegamos a plantearnos cambiar de dataset), aunque estamos contentos con el resultado final.

Otro de los problemas que se nos presentó fue a la hora de tener gráficas que concedieran un punto de vista más visual al trabajo, y es que no sabíamos muy bien cómo hacerlo al tener tantas dimensiones en nuestro dataset. Fue por esto que se nos ocurrió hacer las combinaciones de dos columnas y sacar sus porcentajes además del porcentaje con todas ellas, tratando así de sacar las más relevantes que en conjunto sacaran gráficas bastante informativas.

Nos planteamos también la posibilidad de incluir clustering sobre nuestro dataset, pero nos pareció algo complicado sacar tipos muy definidos de entre nuestros datos, y viendo las gráficas resultantes no tenía demasiado sentido incluirlo. Nuestra idea era que los juegos se agrupasen por géneros o por fecha de lanzamiento, pero estos grupos no parecían tener características comunes tan fuertes como para formar clusters diferenciados. Una vez más, problema del dataset.

Sin embargo, el conjunto resultante de aplicar los métodos de aprendizaje automático principales, sumado a la posibilidad de comprobar el éxito de nuevos juegos introducidos por el usuario, da lugar a un trabajo que recoge todo lo aprendido en la asignatura, aplicado a un dataset sin tratar, y aproximando los conocimientos académicos obtenidos a problemas de aprendizaje automático reales.

7- Referencias

-Link al dataset: https://www.kaggle.com/nikdavis/steam-store-games