

# BodyPerfomance Classification

---

Autores	email
Clara Daniela Sima	csima@ucm.es
Stiven Arias Giraldo	starias@ucm.es

Resultados del proyecto final de la asignatura de **Aprendizaje Automático y Minería de Datos** del grado de **Desarrollo de Videojuegos** de la **Universidad Complutense de Madrid**

## Descripción

---

La base de datos que hemos escogido está compuesta por múltiples atributos que describen características físicas de diversas personas. Por ser más concretos, contamos con **13393 ejemplos de entrenamiento**, es decir, 13393 personas, de las cuales tenemos **11 atributos** y un resultado para cada una.

Esto quiere decir que nuestro dataset está compuesto por una **matriz de (13393, 12) dimensiones**, donde la última columna nos indica la calificación de un **entrenamiento físico** de cada persona en función del resto de atributos, pudiendo ser **A, B, C, D** dicha calificación, siendo **A** el mejor resultado posible.

Description

### Context

This is data that confirmed the grade of performance with age and some exercise performance data.

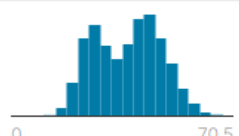

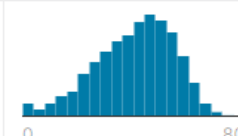
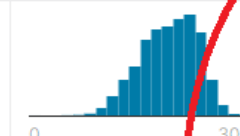
### Content

data shape : (13393, 12)

- age : 20 ~64
- gender : F,M
- height\_cm : (If you want to convert to feet, divide by 30.48)
- weight\_kg
- body fat\_%
- diastolic : diastolic blood pressure (min)
- systolic : systolic blood pressure (min)
- gripForce
- sit and bend forward\_cm
- sit-ups counts
- broad jump\_cm
- class : A,B,C,D ( A: best) / stratified

Detail Compact Column

12 of 12 columns ▾

#	# gripForce	# sit and bend forw...	# sit-ups counts	# broad jump_cm	▲ class	
					C	25%
					D	25%
					Other (6695)	50%
0	70.5	-25	213	0	303	
23.1		13.1	28.0	144.0	C	
52.5		19.2	55.0	232.0	C	
48.9		7.2	54.0	213.0	C	
34.1		19.0	30.0	155.0	A	
25.7		22.9	39.0	178.0	C	
59.6		17.8	61.0	239.0	A	

Así pues, la idea principal de este proyecto es aprovechar los diferentes algoritmos de aprendizaje automático realizados durante el curso, procesando los diferentes datos para poder determinar qué sistema de aprendizaje resulta más óptimo para clasificar correctamente el *dataset*. Cada uno de los sistemas tendrán que predecir cuál ha sido el grado de eficiencia del usuario (A, B, C o D) para finalmente comparar dicha predicción con los datos reales.

Los sistemas son los siguientes: *SVM*, *Regresión logística* y *Redes Neuronales*

## Inicialización y selección de los datos de entrenamiento

Dentro del **main.py** tenemos el lanzador del programa, donde importamos los diferentes métodos de los sistemas de clasificación.

Para agilizar el proceso de selección del sistema tenemos una variable **system** para seleccionar el sistema que se desea probar:

```
from svmPerformance import svmClassification as svm
from logisticRegression import bestPairClassification as pairLog,
logisticRegressionClassification as log
from redesNeuronales import neuralNetworkClassification as red_neu
from initData import *

def main(system=0):
    # Carga de los datos en un diccionario dataset
    allX, allY, dataset = loadData()
    # Fragmentación del dataset
    X, y, Xval, yval, Xtest, ytest = selectingData(allX, allY)
```

```

if system == 0:
    # Clasificación de los datos mediante SVM
    svm(X, y, Xval, yval, Xtest, ytest)
elif system == 1:
    # Clasificación de los datos mediante Regresión logística
    log(X, y, Xval, yval, Xtest, ytest)
    # Clasificación de los datos mediante Regresión logística
    # escogiendo el mejor par de atributos
    pairLog(X, y, Xval, yval, Xtest, ytest, dataset)
elif system == 2:
    # Clasificación de los datos mediante Redes Neuronales
    red_neu(X, y, Xval, yval, Xtest, ytest)
return 0

system = 1
main(system)

```

En primer lugar, se cargan los datos con **loadData**, donde se lee el **.csv** y se hacen las correspondientes conversiones de datos para obtener matrices de **floats**. Además, tenemos el método **plot\_corr**, el cual nos sirve para dibujar una gráfica que muestra la matriz de correlaciones entre cada par de datos (exceptuando **gender** y **Class**, ya que es lo que devuelve **df.corr()**, además que nuestro sistema va a clasificar los valores en función de **Class**). Esta matriz de correlaciones nos parece interesante porque con ella se pueden tener otro tipo de observaciones, sin embargo, no es del todo relevante para nuestro sistema de clasificación.

```

from tarfile import DIRTTYPE
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def plot_corr(df, size=10):
    """
    Dibuja una matriz de correlaciones por filas y columnas
    para cada par de datos del dataset
    """

    corr = df.corr()
    fig, ax = plt.subplots(figsize=(size, size))
    im = ax.matshow(corr, cmap="RdPu")
    plt.colorbar(im, label="Correlación entre los atributos")
    plt.xticks(range(len(corr.columns)), corr.columns, rotation=20)
    plt.yticks(range(len(corr.columns)), corr.columns)

    values = corr.values
    for (i, j), z in np.ndenumerate(values):
        ax.text(j, i, '{:0.1f}'.format(z), fontsize=14, ha='center', va='center')

    plt.show()

def loadData(dispatch_corr=False):
    """
    Lectura del dataset que contiene los datos de entrenamiento del sistema
    """

```

```

# Carga de los datos del csv y conversión a diccionario sencillo de atributos
dataset = pd.read_csv("./src/assets/bodyPerformance.csv")
dataset.drop_duplicates(inplace=True)
dataset.columns = dataset.columns.str.strip().str.replace(' ', '_')
dataset.rename(columns={'class': 'Class', 'body_fat_%': 'body_fat',
'sit-ups_counts': 'sit_ups_counts',
'sit_and_bend_forward_cm': 'sit_bend_forw_cm'}, inplace=True)

# El número de elementos del data set es de 14K * 12, por tanto,
# para reducir el tiempo de Debug del programa se va a elegir un grupo
reducido
rows = int(dataset.shape[0])
cols = int(dataset.shape[1] - 1)

# Muestra el gráfico de las correlaciones
plot_corr(dataset)

# Carga de atributos en matrices de numpy
features = np.array(dataset.values[:rows, :cols])
# Se consideran hombres = 1, mujeres = 0
features[:, 1] = (dataset['gender'][:rows] == 'M') * 1
features = features.astype(np.double)

# Carga de los resultados de cada ejemplo de entrenamiento
results = np.zeros(rows)
test = dataset['Class'].values
for i in range(rows):
    results[i] = ord(test[i]) - 64

return features, results, dataset

```

Hemos separado los datos en 3 grupos: **entrenamiento**, **validación** y **testing** donde la proporción de cada grupo es 60%, 20% y 20% del número total de datos respectivamente.

```

def selectingData(allX, allY):
    """
    Selección de los datos de entrenamiento,
    de validación y de pruebas
    """
    # Se cogerá un 60% de los datos para entrenar
    X = normalizeMatrix(allX[:int(0.6 * np.shape(allX)[0])])
    y = allY[:int(0.6 * np.shape(allY)[0])]

    # Después se coge un 20% para evaluar
    Xval = normalizeMatrix(allX[int(0.6 * np.shape(allX)[0]) : int(0.8 *
np.shape(allX)[0])])
    yval = allY[int(0.6 * np.shape(allY)[0]) : int(0.8 * np.shape(allX)[0])]

    # Por último, el 20% restante para testing
    Xtest = normalizeMatrix(allX[int(0.8 * np.shape(allX)[0]):])
    ytest = allY[int(0.8 * np.shape(allX)[0]):]

```

```

        return X, y, Xval, yval, Xtest, ytest

def normalizeMatrix(X):
    """
    Normaliza el array X
    xi = (xi - ui) / si
    """
    matriz_normal = np.empty_like(X)

    # La media y la varianza de cada columna
    u = np.mean(X, axis=0)
    s = np.std(X, axis=0)

    matriz_normal = (X - u) / s
    return matriz_normal

```

En los valores **X** guardamos los atributos de cada ejemplo de entrenamiento y en los valores **Y**, los resultados.

La idea de esta distinción de valores es utilizar los **datos de entrenamiento** para entrenar el sistema utilizado. A partir de los valores generados por el sistema, se utilizarán los **datos de validación** para verificar que los valores obtenidos son realmente óptimos y, finalmente, con los **datos de testing** se pone a prueba el sistema al completo.

Para observar los resultados de nuestros sistemas de clasificación, hemos realizado diversas pruebas, por lo que vamos a mostrar los resultados finales y más relevantes. Además, hemos cogido todos los ejemplos de entrenamiento del dataset porque era lo que mejor resultado nos daba en este caso, es decir, casi 14000 ejemplos de entrenamiento, a pesar de que fueran demasiados.

## Sistema SVM

En primer lugar, realizamos la clasificación de los datos mediante **SVM** (Support Vector Machine).

Para realizar el entrenamiento con SVM hay que tener en cuenta 2 parámetros: **initialValue**, **iters**.

- **initialValue**: sirve para inicializar tanto el parámetro de regularización **C** y el parámetro **sigma**. Ambos valores serán usados en la función **SVC**, la cual está basada en un kernel gaussiano **rbf** y se irán modificando a través del bucle de entrenamiento.
- **iters**: es el número de iteraciones que habrá para realizar el entrenamiento de los valores **C** y **sigma**, es decir, en total habrá **iters \* iters** iteraciones para el entrenamiento.

Durante este proceso, vamos guardando los mejores valores del entrenamiento en función de la puntuación obtenida con los datos de validación **Xval**, **yval**.

```

def svmClassification(X, y, Xval, yval, Xtest, ytest):
    """
    Clasificación de los datos mediante SVM
    """
    print("Entrenando sistema de clasificacion de bodyPerfomance")
    initialValue = 0.0001

```

```

    iters = 20
    XX = np.insert(X, 0, 1, axis = 1)
    XXval = np.insert(Xval, 0, 1, axis = 1)
    XXtest = np.insert(Xtest, 0, 1, axis = 1)

    svm, params, bestIndex, bestScore, eTraining, eValidation =
selectParameters(XX, y, XXval, yval, initialValue, iters)

    reg = params[0, bestIndex[0]]
    sigma = params[1, bestIndex[1]]

    # Precision del svm
    print(f"Error: {1 - bestScore}")
    print(f"Mejor reg: {reg}")
    print(f"Mejor sigma: {sigma}")
    testScore = np.zeros(3)
    testScore[0] = svm.score(XX, y)
    print(f"Precisión sobres los datos de entrenamiento: {testScore[0] * 100}%")
    testScore[1] = svm.score(XXval, yval)
    print(f"Precisión sobres los datos de evaluación: {testScore[1] * 100}%")
    testScore[2] = svm.score(XXtest, ytest)
    print(f"Precisión sobres los datos de testing: {testScore[2] * 100}%")
    print(f"Precisión media: {testScore.mean() * 100}%")
    print("Success")

    drawGraphics(XX, y, XXval, yval, XXtest, ytest, svm, params, eTraining,
eValidation, bestIndex)

```

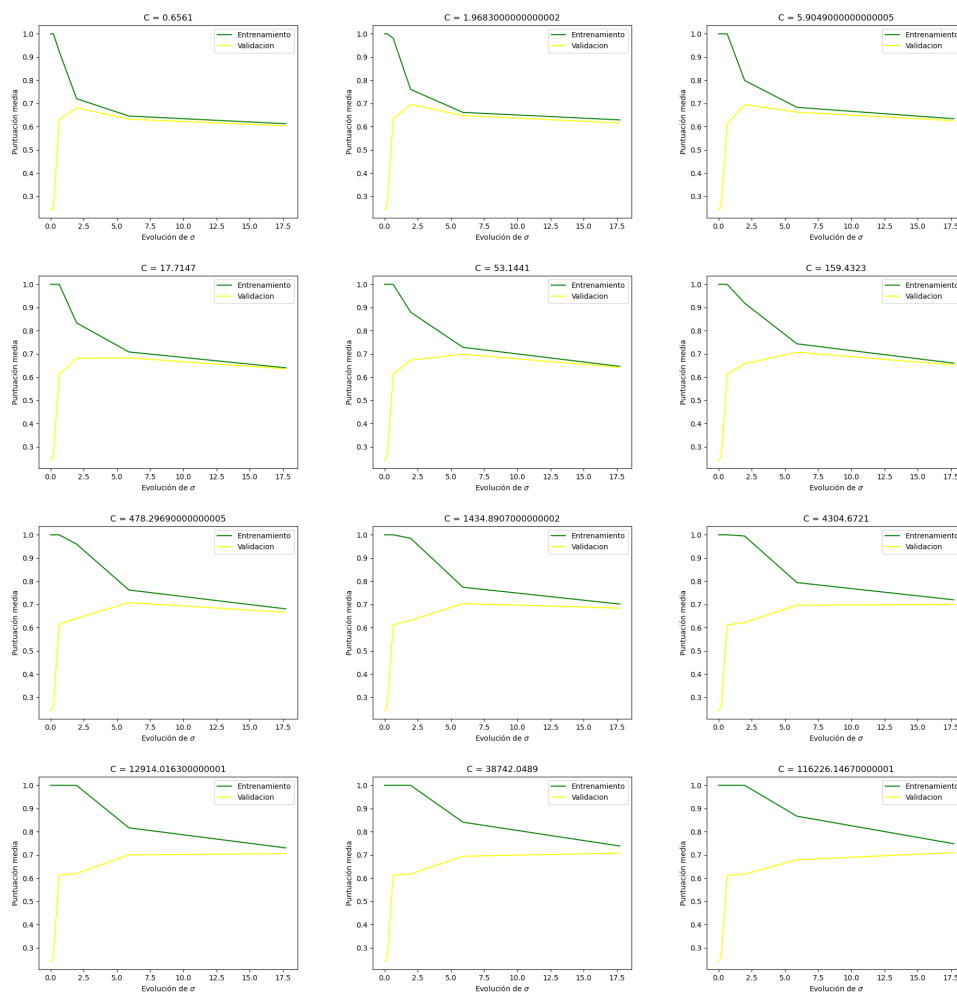
Finalmente, no solo utilizamos `Xtest`, `ytest` para realizar la comprobación de resultados, aunque sus datos serían los más relevantes, sino que comprobamos la puntuación de cada grupo de datos, imprimiendo los cálculos en consola y mostrándolos mediante gráficas.

# SVM Resultados

Valores iniciales.  $\sigma = 0.01$ ,  $C = 0.01$ , iteraciones = 20

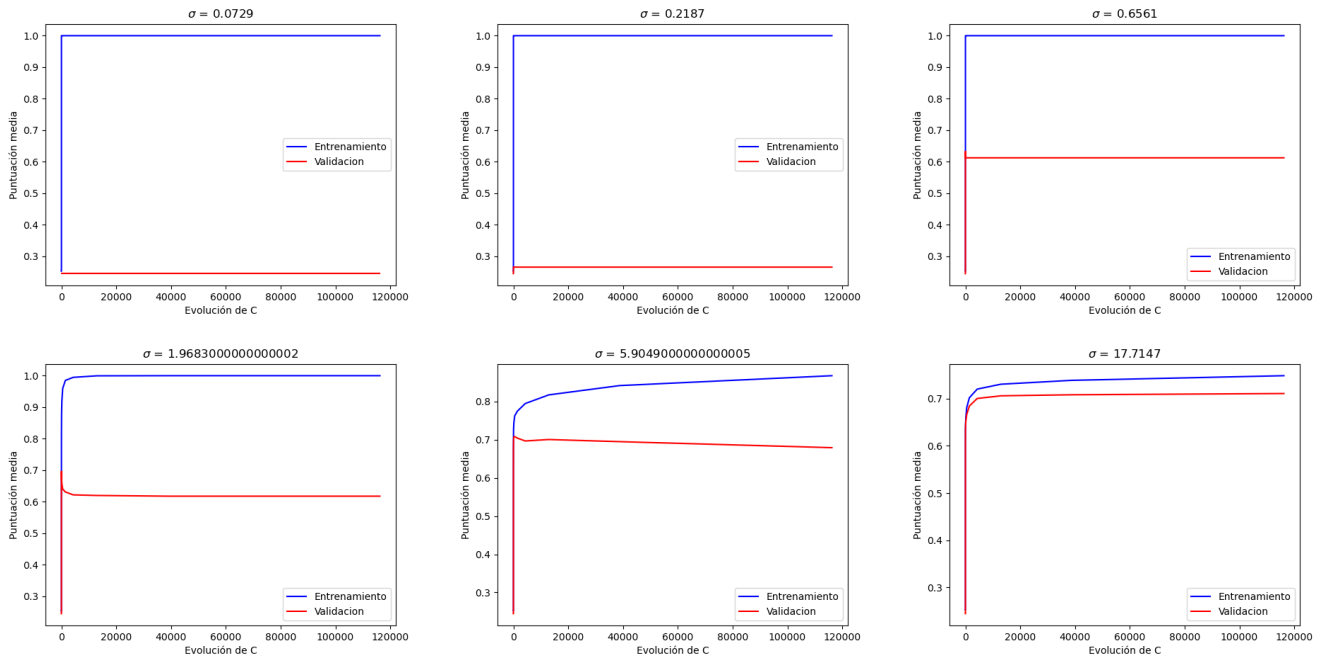
## Evolución de Sigma

En esta imagen podemos observar la evolución del parámetro  $\sigma$  en función de  $C$ . A pesar de haber estado entrenando con 20 valores distintos para  $C$  y  $\sigma$ , es decir, con 20 iteraciones para cada uno, resultaban gráficas que no mostraban ningún tipo de evolución pues el valor de los parámetros en ese instante no tenía ningún impacto. Así pues, mediante estas gráficas podemos observar como crece la precisión media de las predicciones, tanto para los datos de entrenamiento como los de validación, a medida que aumenta  $C$ , es decir, tal vez con mayores iteraciones y mayores valores de  $C$  podríamos obtener valores más precisos.



## Evolución de C

Ahora bien, con estas gráficas observamos el efecto contrario, la evolución de **C** en función de **sigma**. En este caso, obtenemos menos gráficas relevantes puesto que los valores altos de sigma no otorgaban ningún tipo de impacto ni de beneficio al sistema de clasificación.



## Resultados

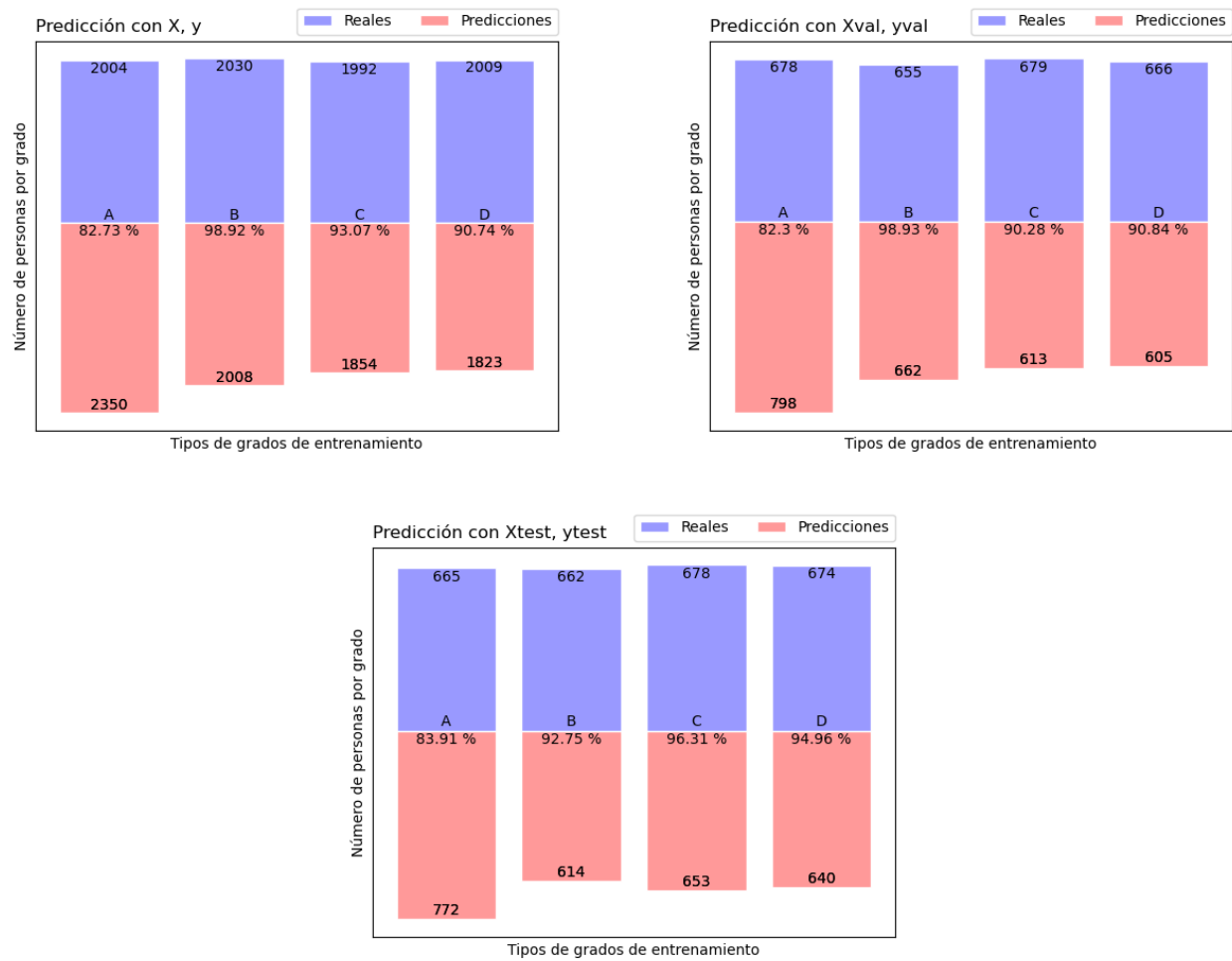
Como podemos observar, los mejores valores son: **C (reg) = 116226**. y **sigma = 17.7** otorgando las precisiones que se pueden observar en la imagen, calculadas mediante **svm.score**.

```
Entrenando sistema de clasificacion de bodyPerfomance
Error: 0.28939507094846906
Mejor reg: 116226.14670000001
Mejor sigma: 17.7147
Precisión sobres los datos de entrenamiento: 74.83509645301805%
Precisión sobres los datos de evaluación: 71.0604929051531%
Precisión sobres los datos de testing: 70.51138484509146%
Precisión media: 72.1356580677542%
Success
```



# Predicciones

Finalmente, pusimos a prueba el sistema utilizando los mejores valores, obteniendo una gráfica de barras como la que se puede observar. En ella estamos comparando los valores reales con las predicciones, indicando el número de personas que hay por grado, a qué grado pertenecen y el porcentaje de precisión de esa columna.



# Sistema de Regresión Logística

---

Para entrenar los datos mediante **Regresión Logística** hacemos uso de **oneVsAll**, donde hay que tener en cuenta dos parámetros: **initReg** e **iters**.

- **initReg**: sirve para inicializar el parámetro de regularización **reg**.
- **iters**: es el número de iteraciones que habrá para entrenar el término de regularización, de manera que pueda ofrecer los mejores resultados para **theta**.

Para el entrenamiento, al igual que antes, tenemos en cuenta la mejor evaluación en función de los datos de validación **Xval**, **yval** y, finalmente, imprimimos los mejores resultados por consola.

```
def oneVsAll(X, y, Xval, yval, initReg=0.01, iters=8, num_labels=4):
    """
    Entrenamiento de varios clasificadores por regresión logística
    """
    numFeatures = X.shape[1]
    # Matriz de parámetros theta
    theta = np.zeros((num_labels, numFeatures))
    perfmY = getLabelMatrixY(y, num_labels)
    # Matriz de etiquetas yval
    ylv = getLabelMatrixY(yval, num_labels)

    # Entrenamiento
    validation = np.zeros(num_labels)
    bestScore = np.zeros(num_labels)
    bestReg = np.zeros(num_labels)
    bestTheta = np.zeros((num_labels, numFeatures))

    for i in range(num_labels):
        for j in range(iters):
            reg = initReg * 3**j
            # Se entrena con las X
            result = opt.fmin_tnc(func = coste, x0 = theta[i, :], fprime =
gradiente,
                                args=(X, perfmY[:, i], reg), disp=0)
            theta[i, :] = result[0]

            # Se evalua con las Xval
            # Matriz de etiquetas yval
            validation[i] = evalua(i, theta[i, :], Xval, ylv[:, i])
            if(validation[i] > bestScore[i]):
                bestScore[i] = validation[i]
                bestReg[i] = reg
                bestTheta[i, :] = theta[i, :]

    return bestScore, bestReg, bestTheta

def logisticRegresionClassification(X, y, Xval, yval, Xtest, ytest):
    """
    Clasificación de los datos mediante Regresión Logística
```

```

"""
print("Entrenando sistema de clasificación de bodyPerfomance")
bestScore, bestReg, bestTheta = oneVsAll(X, y, Xval, yval)

#-----PRINT-DATA-----#
num_labels = 4

print("\n-----")
print("\nMejores resultados del entrenamiento")
for i in range(num_labels):
    str1 = f"Mejor reg {chr(i + 65)}: {bestReg[i]}"
    str2 = f"Evaluación {chr(i + 65)}: {bestScore[i] * 100}%"
    print(str1 + " - " + str2)

print(f"Error: {1 - bestScore.mean()}")
print("Evaluación media: ", bestScore.mean() * 100)
print("\n-----")

print("\nComprobación de parámetros con ytest, Xtest")
testResults = np.zeros(num_labels)
# Matriz ytest de etiquetas
y1t = getLabelMatrixY(ytest, num_labels)

for i in range(num_labels):
    testResults[i] = evalua(i, bestTheta[i, :], Xtest, y1t[:, i])
    print(f"Evaluación {chr(i + 65)}: {testResults[i] * 100}%")
print("Evaluación media test: ", testResults.mean() * 100)

print("\n-----")
print("Success")

#-----GRAPHICS-----#

return 0

```

Por otro lado, hemos realizado el mismo sistema de entrenamiento, pero con el objetivo de averiguar cuál es el mejor par de atributos para predecir el resultado. Para ello, tenemos las siguientes funciones:

```

def calculatePair(X, Xval, i, j):
    """
    Devuelve el par correspondiente de datos
    """
    numFeatures = 2
    pair = np.zeros((X.shape[0], numFeatures))
    pair[:, 0] = X[:, i]
    pair[:, 1] = X[:, j]

    pair_val = np.zeros((Xval.shape[0], numFeatures))
    pair_val[:, 0] = Xval[:, i]
    pair_val[:, 1] = Xval[:, j]

```

```
return pair, pair_val
```

```
def bestPairClassification(X, y, Xval, yval, Xtest, ytest, dataset):
    """
    Clasificación de los datos mediante Regresión Logística
    para cada par de datos, escogiendo los 2 mejores atributos
    que clasifiquen el resultado
    """
    print("Entrenando sistema de clasificación de bodyPerfomance")
    bestScore = np.zeros(1)
    bestPair = 0
    bestReg = 0
    bestTheta = 0
    f1, f2 = 0, 0

    initReg = 0.0003
    n = 1
    for i in range(X.shape[1]):
        for j in range(i + 1, X.shape[1]):
            pair, pair_val = calculatePair(X, Xval, i, j)
            score, reg, theta = oneVsAll(pair, y, pair_val, yval, initReg, 10)
            if(score.mean() > bestScore.mean()):
                bestScore = score
                bestPair = [pair, pair_val]
                bestReg = reg
                bestTheta = theta
                f1, f2 = i, j

    #-----PRINT-DATA-----#
    num_labels = 4
    features = dataset.columns[f1] + ", " + dataset.columns[f2]

    print("\n-----")

    print("\nMejores resultados del entrenamiento")
    print(f"Mejor par de atributos: {features}")
    print(f"Mejor reg: {bestReg}")
    for i in range(num_labels):
        str1 = f"Mejor reg {chr(i + 65)}: {bestReg[i]}"
        str2 = f"Evaluación {chr(i + 65)}: {bestScore[i] * 100}%"
        print(str1 + " - " + str2)

    print("Evaluación media: ", bestScore.mean() * 100)
    print(f"Error: {1 - bestScore.mean()}")

    print("\n-----")

    print("\nComprobación de parámetros con ytest, Xtest")
    testResults = np.zeros(num_labels)
    # Matriz ytest de etiquetas
    numFeatures = 2
    ylt = getLabelMatrixY(ytest, num_labels)
    pair_test = np.zeros((Xtest.shape[0], numFeatures))
    pair_test[:, 0] = Xtest[:, f1]
    pair_test[:, 1] = Xtest[:, f2]
```

```

for i in range(num_labels):
    testResults[i] = evalua(i, bestTheta[i, :], pair_test, ylt[:, i])
    print(f"Evaluación {chr(i + 65)}: {testResults[i] * 100}%")
print("Evaluación media test: ", testResults.mean() * 100)

print("\n-----")

return 0

```

## Regresión Logística resultados

Resultados con todos los atributos

Valores iniciales. `initReg=0.01`, `iters=8`

Aquí podemos observar que el mejor término de regularización no es el mismo para predecir los resultados, es decir, para los grados **A** y **B** obtenemos un valor de **7.29** y para **C** y **D** de **0.01**.

Entrenando sistema de clasificación de bodyPerfomance

-----

Mejores resultados del entrenamiento

Mejor reg A: 7.29 - Evaluación A: 90.06721433905899%

Mejor reg B: 7.29 - Evaluación B: 82.30022404779686%

Mejor reg C: 0.01 - Evaluación C: 75.54144884241971%

Mejor reg D: 0.01 - Evaluación D: 74.64525765496639%

Error: 0.19361463778939503

Evaluación media: 80.63853622106049

-----

Comprobación de parámetros con ytest, Xtest

Evaluación A: 87.94326241134752%

Evaluación B: 82.45614035087719%

Evaluación C: 75.28928704740575%

Evaluación D: 74.65472191116088%

Evaluación media test: 80.08585293019785

-----

## Resultados del mejor par de atributos

---

Valores iniciales. `initReg = 0.0003, iters=10`

Finalmente, obtenemos que el mejor par de atributos para predecir los resultados sería el que está compuesto por ***sit\_bend\_forw\_cm*** y ***sit\_ups\_counts***, que son básicamente dos tipos de ejercicios físicos que se realizan durante el *BodyPerformance*. Además, lo más curioso a destacar es que la precisión es prácticamente la misma que la obtenida con todos los atributos, por lo que de cara a una mejora en el sistema, sería interesante pensar en cómo utilizar el menor número de atributos para realizar la predicción.

```
Entrenando sistema de clasificación de bodyPerformance

-----

Mejores resultados del entrenamiento
Mejor par de atributos: sit_bend_forw_cm, sit_ups_counts
Mejor reg: [5.9049e+00 3.0000e-04 3.0000e-04 3.0000e-04]
Mejor reg A: 5.9049 - Evaluación A: 86.78117998506347%
Mejor reg B: 0.0003 - Evaluación B: 78.90216579536967%
Mejor reg C: 0.0003 - Evaluación C: 75.54144884241971%
Mejor reg D: 0.0003 - Evaluación D: 74.64525765496639%
Evaluación media: 78.96751306945482
Error: 0.2103248693054518

-----

Comprobación de parámetros con ytest, Xtest
Evaluación A: 85.81560283687944%
Evaluación B: 78.46211272863009%
Evaluación C: 75.28928704740575%
Evaluación D: 74.69204927211646%
Evaluación media test: 78.56476297125793

-----
```

# Sistema de Redes Neuronales

---

En primer lugar se inicializan los pesos de forma aleatoria (**Theta1** y **Theta2**). Luego se implementa **forward propagation** para obtener la **hipótesis**, seguido de la implementación de la **función de coste**. Posteriormente, se implementa **backpropagation** para computar las derivadas parciales. Finalmente, se utiliza el **descenso del gradiente** para encontrar los mejores parámetros.

La red neuronal está constituida por 3 capas, de manera que en la primera se encuentra la **capa de entrada** que tiene tantos nodos como atributos tenga el dataset. Luego nos encontramos con las **capas ocultas** donde hemos escogido el número de capas ocultas mediante prueba y error hasta encontrar un número que nos satisfaga. Por último, se encuentra la **capa de salida** que tiene tantos nodos como tipo de resultados tiene nuestro dataset.

Ahora bien, para entrenar la red neuronal hay que tener en cuenta los siguientes parámetros: **número de capas ocultas**, **número de nodos en la capa oculta**, **epsilon**, **número de iteraciones para el backpropagation** y **lambda**.

Además, se han realizado pruebas con diferentes tamaños en el dataset, escogiendo el **10%** del total de los datos y el **100%**

También hemos utilizado la función de *checking* otorgada en la **práctica 4** para comprobar que el *backpropagation* funciona correctamente.

Por otro lado, para realizar un correcto entrenamiento hacemos uso de **cross-validation** combinando el mínimo error y el mejor resultado de precisión en cada iteración, de manera que hemos realizado la siguiente fórmula:

```
special_cost = Jval[i]/10 + diff/4
```

La razón de hacer esto es porque no queríamos que la varianza entre los datos de entrenamiento y los datos de validación fuese demasiado grande, pero tampoco queríamos resultados poco precisos, por lo que, de nuevo, mediante prueba y error, conseguimos combinar ambos términos para tener en cuenta la distancia mínima entre cada curva de aprendizaje del error y el mejor resultado de precisión de los datos de validación.

```
def make_neural_network(input_layer, hidden_layer, output_layer, X, y_labels,
                        lamb, iteration, epsilon):
    # Initialize the thetas random values between -eps and eps
    weights_size = hidden_layer * (input_layer + 1) + output_layer *
(hidden_layer + 1)
    weights = np.random.uniform(-epsilon, epsilon, weights_size)

    # Calculate the best thetas
    result = opt.minimize(fun = backprop, x0 = weights,
                        args = (input_layer, hidden_layer, output_layer, X, y_labels,
lamb),
                        method='TNC', jac=True, options={'maxiter': iteration})
    # As the result is an array we remake the thetas matrixes with the
```

correspondent sizes

```
    optT1 = np.reshape(result.x[:hidden_layer * (input_layer + 1)],
(hidden_layer, (input_layer + 1)))
    optT2 = np.reshape(result.x[hidden_layer * (input_layer + 1):],
(output_layer, (hidden_layer + 1)))
    return optT1, optT2
```

```
def neuralNetworkClassification(X, y, Xval, yval, Xtest, ytest):
```

```
    """
```

```
    Clasificación de los datos mediante Redes Neuronales
```

```
    """
```

```
    # We remember the path for the graphics
```

```
    script_dir = os.path.dirname(__file__)
```

```
    result_dir = script_dir[:-4] + '\\memoria\\assets\\neu_net\\'
```

```
    # Initialize variables
```

```
    num_features = X.shape[1]
```

```
    num_labels = 4
```

```
    y_labels = getLabelMatrixY(y, num_labels)
```

```
    yval_labels = getLabelMatrixY(yval, num_labels)
```

```
    # Epsilon
```

```
    init_epsilon = 0.1
```

```
    epsilons = 6
```

```
    best_eps = 0.1
```

```
    # NeuralNet
```

```
    neural_net_iters = 200 # 100
```

```
    hid = 75 # 25
```

```
    input_layer = num_features
```

```
    output_layer = num_labels
```

```
    # Lambdas
```

```
    initLambda = 0.01
```

```
    lambdas = np.zeros(7)
```

```
    best_lambda = 0.01
```

```
    # Misc
```

```
    min_cost = np.inf
```

```
    best_percent = -1
```

```
    best_optT1 = []
```

```
    best_optT2 = []
```

```
    min_diff = np.inf
```

```
    min_special_cost = np.inf
```

```
    # Notas: No hace falta repetir tanto el código, solo con esta cadena de for funciona
```

```
    for j in range(epsilons):
```

```
        epsilon = init_epsilon + 0.02 * j
```

```
        Jval = np.ones(len(lambdas))
```

```
        Jtraining = np.ones(len(lambdas))
```

```
        for i in range(len(lambdas)):
```

```
            # Training with X
```

```
            lambdas[i] = initLambda * 3**i
```

```
            optT1, optT2 = make_neural_network(input_layer, hid, output_layer, X,
                                                y_labels, lambdas[i], neural_net_iters,
```

```
epsilon)
```



```

print(f"Lambda {lambdas[i]}   Epsilon: {epsilon} Iterations:"
      + f"{neural_net_iters} Hidden layers: {hid}")

# We calculate the percentages of succes and using them we calculate
the error for the training data set and cross validation data set
Jtraining[i] = 10 - get_total_percent('training', X, y, optT1,
optT2)/10
current_percent = get_total_percent('validation', Xval, yval, optT1,
optT2)
Jval[i] = 10 - current_percent/10
# Jtraining[i] = J(optT1, optT2, X, y_labels)
# Jval[i] = J(optT1, optT2, Xval, yval_labels)

diff = np.abs(Jtraining[i] - Jval[i])
# We remember the thetas that have the smallest error(biggest
percent) on the cross validation data set but also the minimum distance
special_cost = Jval[i]/10 + diff/4
if Jval[i] < min_cost:
    min_cost = Jval[i]
    min_cost_lamb = lambdas[i]
    min_cost_eps = epsilon
if diff < min_diff:
    min_diff = diff
    diff_lamb = lambdas[i]
    diff_eps = epsilon
if special_cost < min_special_cost: #Combined distance and cost
    min_special_cost = special_cost
    best_percent = current_percent
    best_lambda = lambdas[i]
    best_eps = epsilon
    best_optT1 = optT1
    best_optT2 = optT2
print("\n")

# We prepare the parameters for the graphics
name = 'Lambda' + 'LearningCurve' + '_it_' + str(neural_net_iters) +
'_hidd_' + str(hid) + '_eps_' + str(round(epsilon,2)) + '_size_' +
str(X.shape[0])+'.png'
path = result_dir + name
parameters = 'Hidden layer = ' + str(hid) + ' ' + 'Iterations = ' +
str(neural_net_iters) + ' ' + '$\epsilon$ = ' + str(round(epsilon,2))
create_learning_curve_graphic(path, parameters, lambdas, Jtraining, Jval,
label = "$\lambda$")
print()
print(fr"Min Diff: {round(min_diff,2)} lamb: {diff_lamb} epsilon:
{round(diff_eps,2)}")
print(fr"Min Cost {round(min_cost,2)} Max Percent: {10 - round(min_cost,2)}
lamb: {min_cost_lamb} epsilon: {min_cost_eps}")
print(fr"Mejor lambda: {best_lambda}")
print(fr"Mejor epsilon: {round(best_eps,2)}")
print(f"Precisión mejor validación: {round(best_percent,2)} best cost: {100 -
round(best_percent,2)}")
get_total_percent('Current best test', Xtest, ytest, best_optT1, best_optT2)

```

```
return 0
```

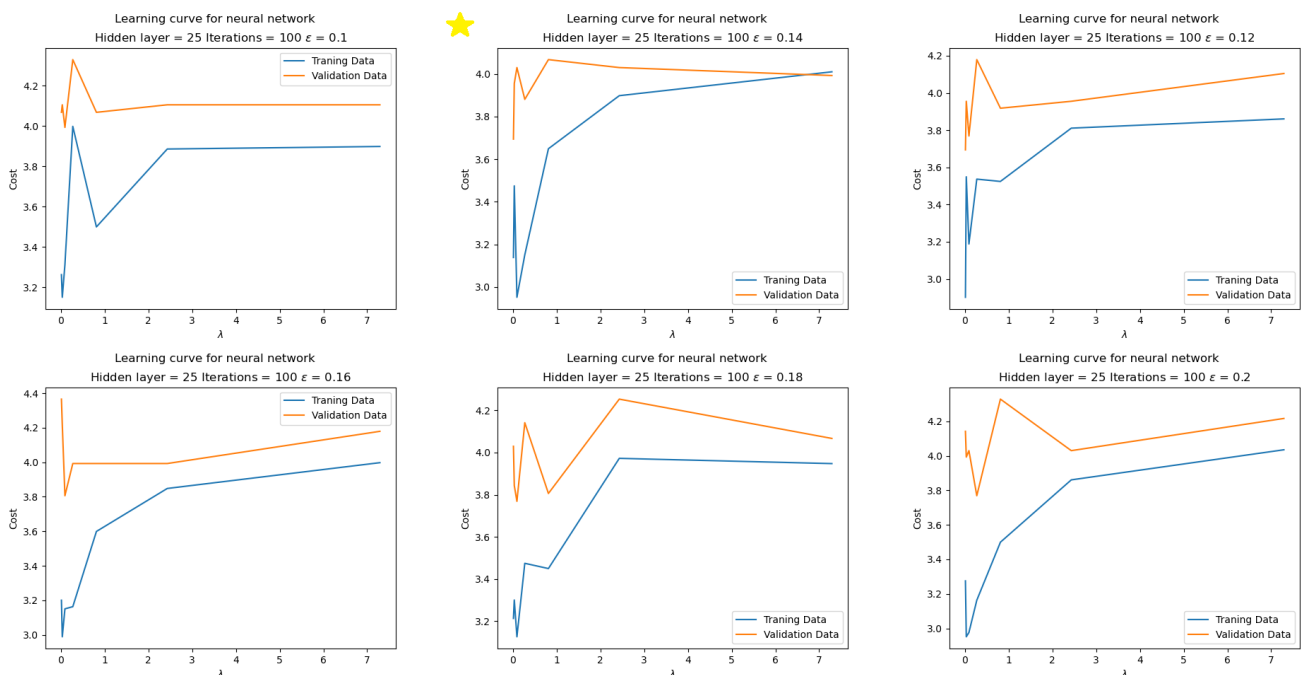
## Redes Neuronales resultados

### 10% de los datos

Valores iniciales. `hidden_layers = 25`, `iters = 100`, `size_training_set = 803` (10% of the data)

Con esta cantidad de datos podemos ver claramente la influencia de  $\lambda$  en el dataset, por lo que podemos ver cuando hay sesgo y varianza. Cuando el error es alto, la función está desajustada (cuanto mayor sea  $\lambda$ , mayor será el sesgo, por lo que las líneas comenzarán a juntarse). Por otro lado, cuando los datos de entrenamiento tienen poco error y los de validación tienen mucho, el sistema comienza a sobreajustarse.

Por ello, la mejor gráfica será la que porta una estrella, pues es en la que se muestra como la solución tiene poco error y también los datos están bastante cercanos los unos a los otros.

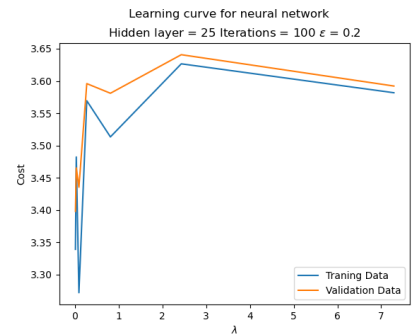
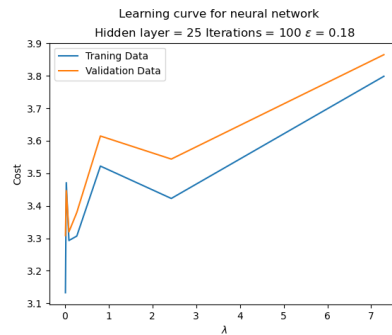
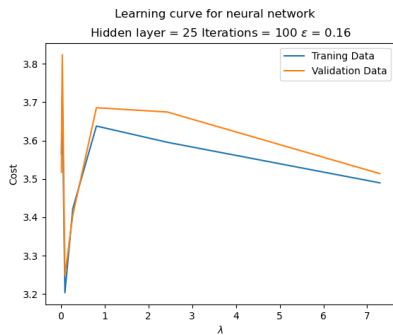
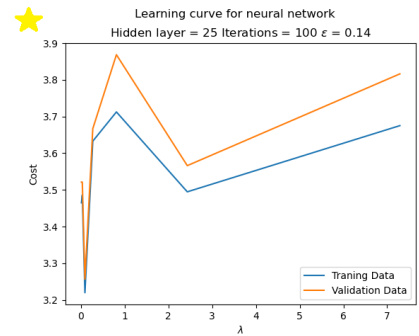
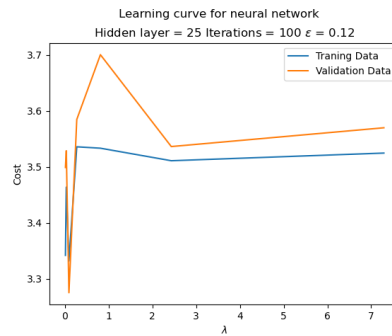
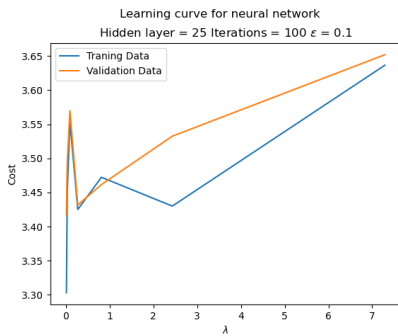


```
Mejor lambda: 7.29
Mejor epsilon: 0.14
Precisión mejor validación: 60.07 best cost: 39.93
Current best test   Porcentaje de acierto: 55.59701492537314%
```

### 100% de los datos

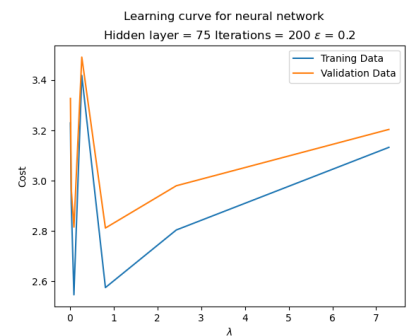
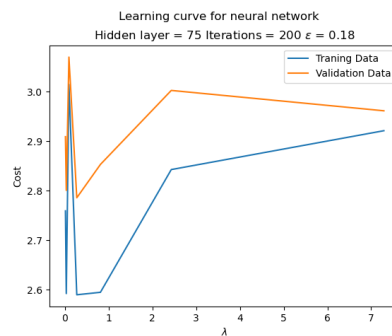
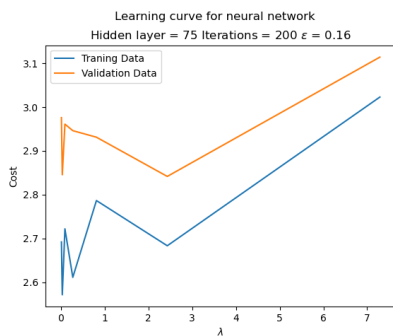
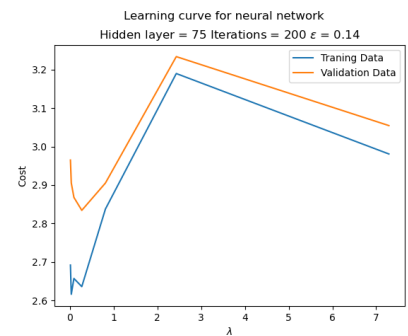
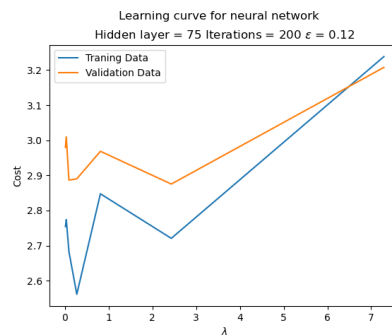
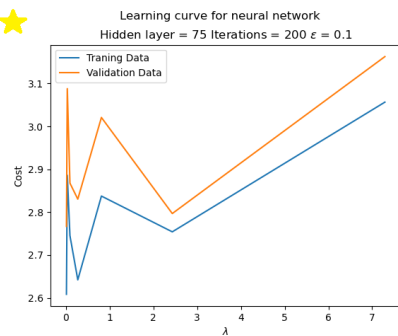
A continuación podemos observar diferentes gráficas con diferentes datos, aplicando lo que acabamos de explicar

Valores iniciales. hidden\_layers = 25, iters = 100



```
Mejor lambda: 0.09
Mejor epsilon: 0.14
Precisión mejor validación: 67.44
Current best test   Porcentaje de acierto: 68.04777902202315%
```

Valores iniciales. hidden\_layers = 75, iters = 200



```
Mejor lambda: 2.43  
Mejor epsilon: 0.1  
Precisión mejor validación: 72.03 best cost: 27.97  
Current best test Porcentaje de acierto: 70.88465845464725%
```

Como podemos observar al ajustar diversos parámetros, en este caso aumentando **iterations** y **hidden\_layers**, hemos ido encontrando poco a poco los mejores resultados para clasificar nuestros datos.