Centro Universitario de Ciencias Exactas e Ingenierías

Licenciatura en Ingeniería en Computación

Materia: Seminario de Solución de Problemas de Inteligencia Artificial II. Clave: I7041.

Profesor: Valdés López Julio Esteban

Estudiante: Silva Moya José Alejandro. Código: 213546894.

Práctica V: ADALINE Batch Gradient Descent

Problema a resolver: Programar un ADALINE BGD para poder resolver un problema de machine learning de una manera más eficiente al proceso realizado con un único perceptrón, o con perceptrones en unicapa.

Desarrollo

En el proceso de desarrollo y aprendizaje de una neurona en el modelo del ADALINE, comparado con el perceptrón, los principales aspectos que cambian son la función de predicción y la manera en que se entrena a la neurona. Observemos los puntos importantes.

```
13  class adaline2:
14      def __init__(self, dimensions, learning_rate):
15          self.dimensions = dimensions
16          self.learning_rate = learning_rate
17          self.w_vector = -1 + 2 * np.random.rand(dimensions, 1)
18          self.b_value  = -1 + 2 * np.random.rand()
19
20      def predict(self, x_vector):
21          return np.dot(self.w_vector.transpose(), x_vector) + self.b_value
```

Como podemos observar, se confirma que los aspectos de inicialización de la neurona se mantienen iguales que en el perceptrón; sin embargo, el cambio importante viene en la función de predicción, en donde podemos ver que se ha eliminado la función de activación, y en cada ocasión retornaremos directamente el producto punto de los vectores, junto con la suma del bias.

```
23      def train(self, X_matrix, y_vector, epochs):
24          n, m = X_matrix.shape
25          error = 0.0
26          for i in range(epochs):
27              w_sumatory_vector = np.zeros((self.dimensions, 1))
28              b_sumatory = 0
29
30              for j in range(m):
31                  y_estimated = self.predict(X_matrix[:,j])
32                  error += y_vector[j] - y_estimated
33                  w_sumatory_vector += (y_vector[j] - y_estimated) * X_matrix[:,j].reshape(-1, 1)
34                  b_sumatory        += (y_vector[j] - y_estimated)
35              self.w_vector += (self.learning_rate / m) * w_sumatory_vector
36              self.b_value  += (self.learning_rate / m) * b_sumatory
37              return error
```

En la imagen anterior podemos ver la función de entrenamiento. El primer cambio importante se encuentra dentro del ciclo de las épocas, en donde generamos un vector de pesos sinápticos inicializado en ceros, que tendrá las mismas dimensiones que el vector de pesos original de la neurona. Además de esto, también tendremos un bias temporal o auxiliar.

A continuación, obtenemos la estimación de la neurona y el error para graficar, e iremos acumulando en nuestras variables temporales:

- El error multiplicado por el vector de entradas correspondiente a la iteración, en W temporal.
- El error, en el bias temporal.

Al terminar la iteración de los patrones de entrenamiento – justo antes de comenzar una nueva época – actualizaremos los valores oficiales de nuestro vector de pesos sinápticos y de nuestro bias.

Esto representa los aspectos y cambios más importantes del programa principal de la presente entrega. Ahora podemos proceder a observar la función main y entender cómo se están procesando los datos.

```
26    neurons_array = []
27
28    for i in range(number_of_neurons):
29        net = Adaline_BGD.adaline2(columns, 0.1)
30        neurons_array.append(net)
31    #Perceptron initialization.
```

Comenzamos por generar un arreglo de neuronas de acuerdo a la cantidad de mismas que necesitaremos para resolver el problema en cuestión.

La carga de datos de patrones de aprendizaje y salidas deseadas se mantiene funcionando de la misma manera que lo hacía con anterioridad.

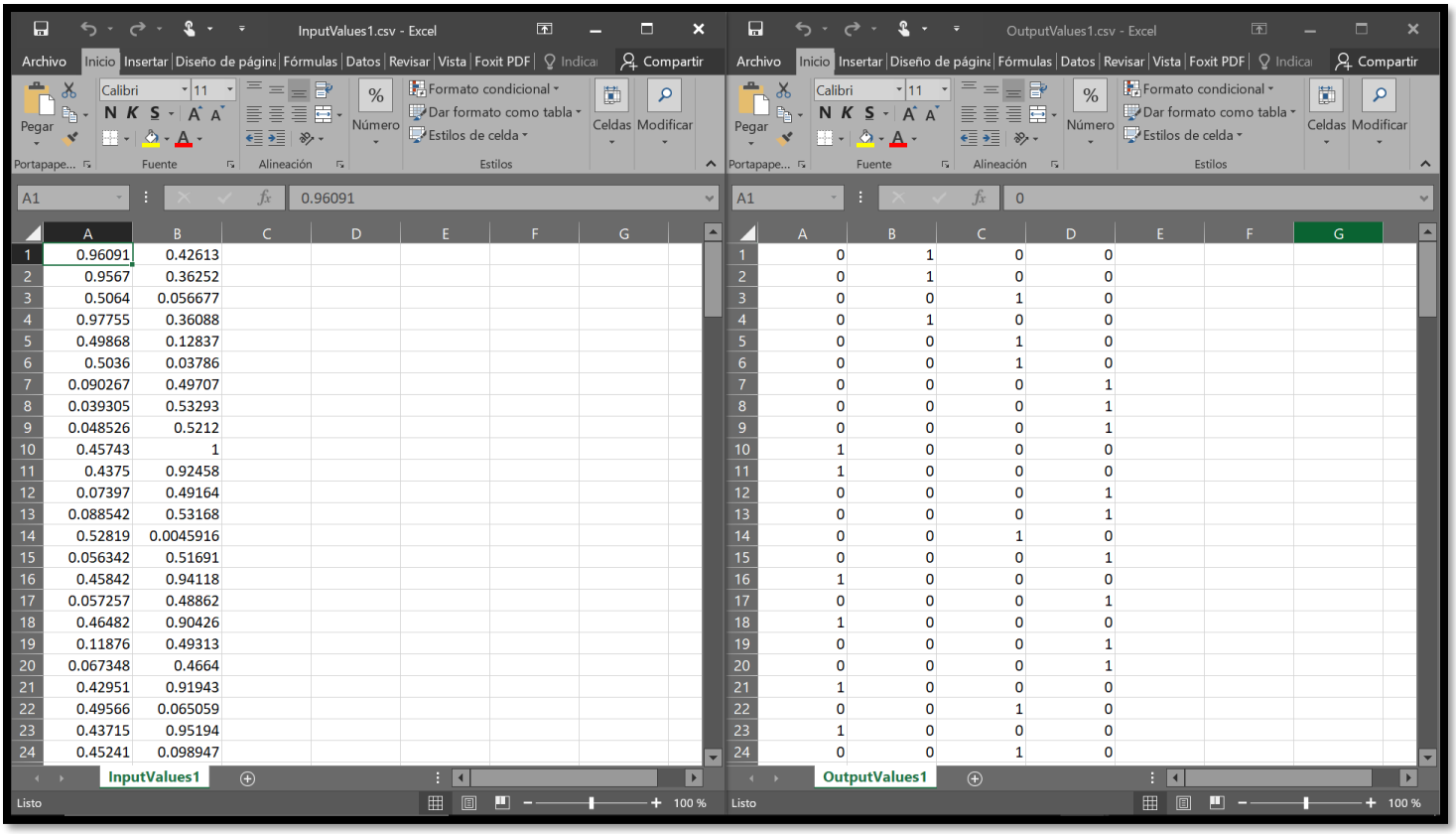
```
47    global_errors = []
48    individual_error = 0.0
49    results = []
50
51    for i in range(epochs):
52        for j in range(number_of_neurons):
53            net = neurons_array[j]
54            individual_error += net.train(X, y[j], 1)
55        individual_error /= number_of_neurons
56        global_errors.append(individual_error)
57        individual_error = 0.0
58        Adaline_BGD.graphError(i, global_errors[i][0])
59
60
61    for i in range(number_of_neurons):
62        net = neurons_array[i]
63        # individual_result = np.concatenate(net.predict(X).tolist())
64        individual_result = np.concatenate(net.predict(X))
65        results.append(individual_result)
66
67    results = np.array(results).T
68    np.savetxt("Results.csv", results, delimiter=",", fmt='%.5f')
```

El proceso de entrenamiento sí cambia de manera sustancial. Observemos en qué consiste.

Comenzamos entrenando las neuronas de acuerdo a la cantidad de épocas que hayamos delimitado al inicio del código; de esta manera podemos simular paralelismo en el entrenamiento de las neuronas del ADALINE sin tener que recurrir a la implementación de hilos. Una vez que entrenemos todas las neuronas por una época completa obtendremos el error de entrenamiento, su promedio, y lo agregaremos a una lista de datos para poder graficar correctamente una línea además de los puntos.

Finalmente, para cada neurona en el modelo obtendremos la predicción de los valores de entrenamiento y los agregaremos a un arreglo para poder mandarlos al csv de resultados finales, que podremos comparar con el documento de salidas esperadas. Evidentemente no serán iguales, debido a que las salidas esperadas son binarias, mientras que las salidas predichas por el modelo son discretas; sin embargo, resulta sencillo observar y deducir si el modelo ha sido entrenado de manera correcta, además de tener la gráfica del error como un dato visual extra.

Resultados



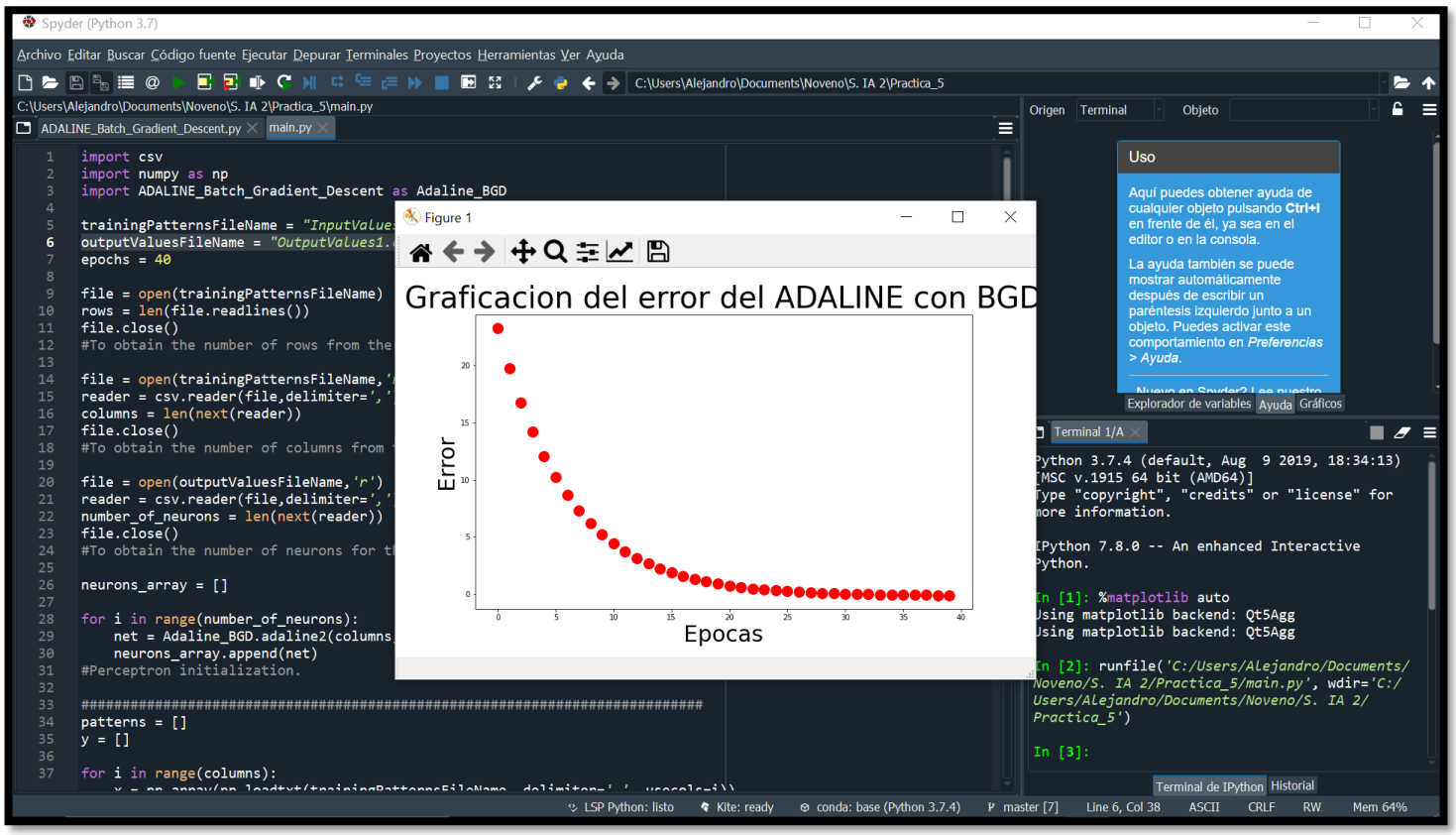
InputValues1.csv - Excel

	A	B	C	D	E	F	G
1	0.96091	0.42613					
2	0.9567	0.36252					
3	0.5064	0.056677					
4	0.97755	0.36088					
5	0.49868	0.12837					
6	0.5036	0.03786					
7	0.090267	0.49707					
8	0.039305	0.53293					
9	0.048526	0.5212					
10	0.45743	1					
11	0.4375	0.92458					
12	0.07397	0.49164					
13	0.088542	0.53168					
14	0.52819	0.0045916					
15	0.056342	0.51691					
16	0.45842	0.94118					
17	0.057257	0.48862					
18	0.46482	0.90426					
19	0.11876	0.49313					
20	0.067348	0.4664					
21	0.42951	0.91943					
22	0.49566	0.065059					
23	0.43715	0.95194					
24	0.45241	0.098947					

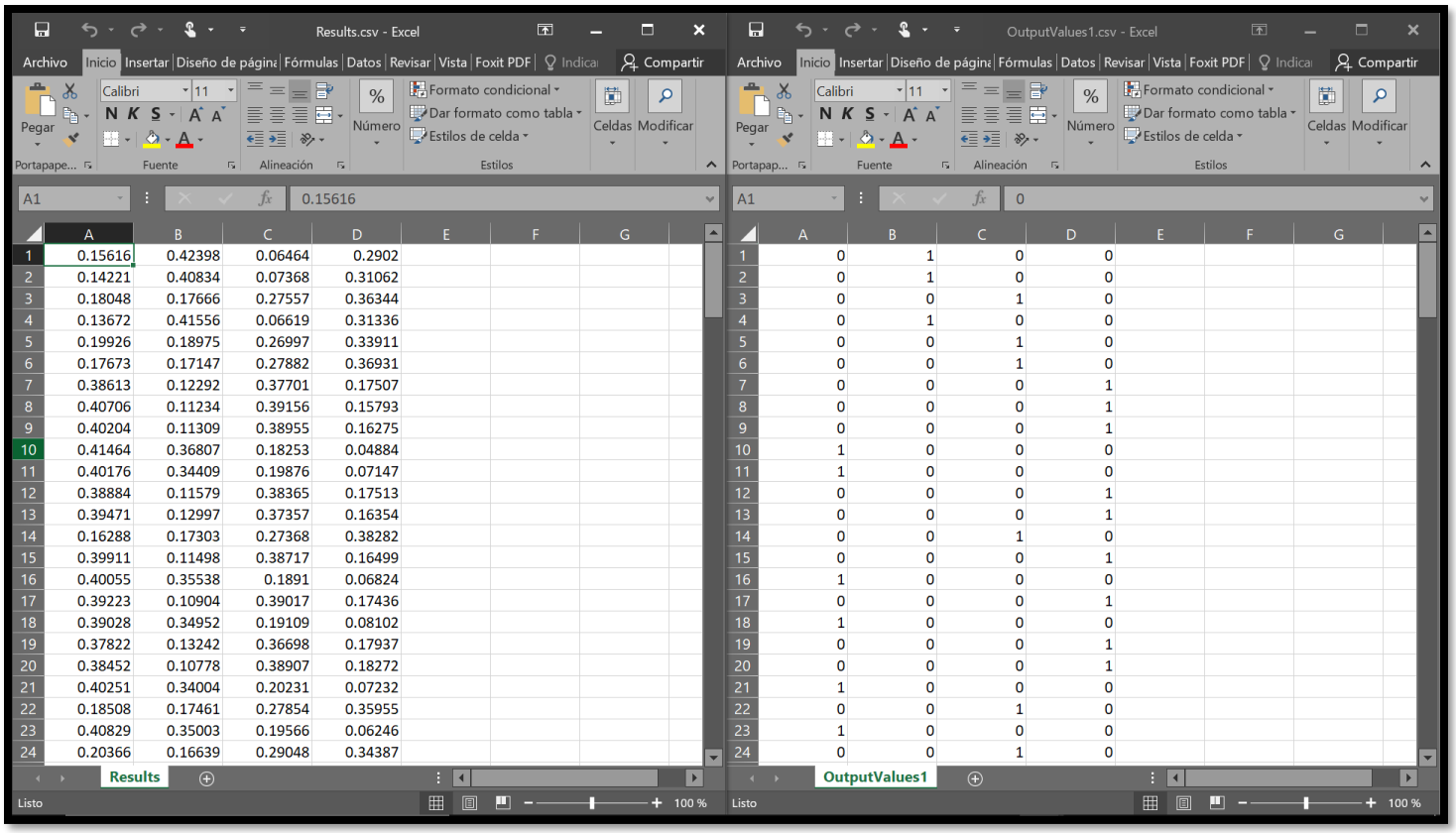
OutputValues1.csv - Excel

	A	B	C	D	E	F	G
1	0	1	0	0			
2	0	1	0	0			
3	0	0	1	0			
4	0	1	0	0			
5	0	0	1	0			
6	0	0	1	0			
7	0	0	0	1			
8	0	0	0	1			
9	0	0	0	1			
10	1	0	0	0			
11	1	0	0	0			
12	0	0	0	1			
13	0	0	0	1			
14	0	0	1	0			
15	0	0	0	1			
16	1	0	0	0			
17	0	0	0	1			
18	1	0	0	0			
19	0	0	0	1			
20	0	0	0	1			
21	1	0	0	0			
22	0	0	1	0			
23	1	0	0	0			
24	0	0	1	0			

Aquí podemos observar el primero de los dos datasets que hemos delimitado para entrenar a modelo. Es un problema en dos dimensiones que se resuelve con 4 neuronas. Observemos el desempeño y sus resultados.



Como podemos observar, el error se encuentra considerablemente cercano a cero después de 40 épocas de entrenamiento. Observemos los resultados numéricos.



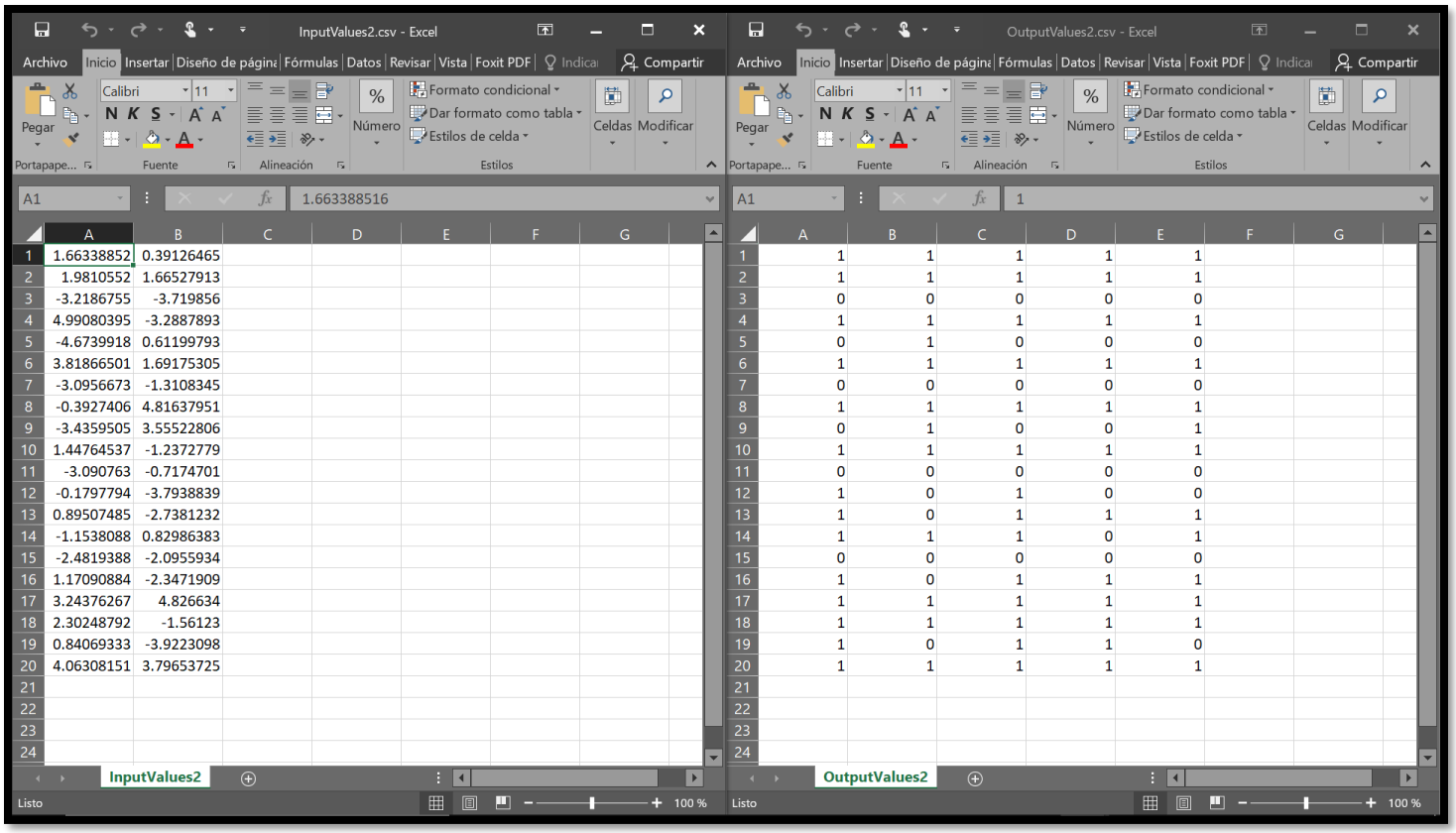
Results.csv - Excel

	A	B	C	D
1	0.15616	0.42398	0.06464	0.2902
2	0.14221	0.40834	0.07368	0.31062
3	0.18048	0.17666	0.27557	0.36344
4	0.13672	0.41556	0.06619	0.31336
5	0.19926	0.18975	0.26997	0.33911
6	0.17673	0.17147	0.27882	0.36931
7	0.38613	0.12292	0.37701	0.17507
8	0.40706	0.11234	0.39156	0.15793
9	0.40204	0.11309	0.38955	0.16275
10	0.41464	0.36807	0.18253	0.04884
11	0.40176	0.34409	0.19876	0.07147
12	0.38884	0.11579	0.38365	0.17513
13	0.39471	0.12997	0.37357	0.16354
14	0.16288	0.17303	0.27368	0.38282
15	0.39911	0.11498	0.38717	0.16499
16	0.40055	0.35538	0.1891	0.06824
17	0.39223	0.10904	0.39017	0.17436
18	0.39028	0.34952	0.19109	0.08102
19	0.37822	0.13242	0.36698	0.17937
20	0.38452	0.10778	0.38907	0.18272
21	0.40251	0.34004	0.20231	0.07232
22	0.18508	0.17461	0.27854	0.35955
23	0.40829	0.35003	0.19566	0.06246
24	0.20366	0.16639	0.29048	0.34387

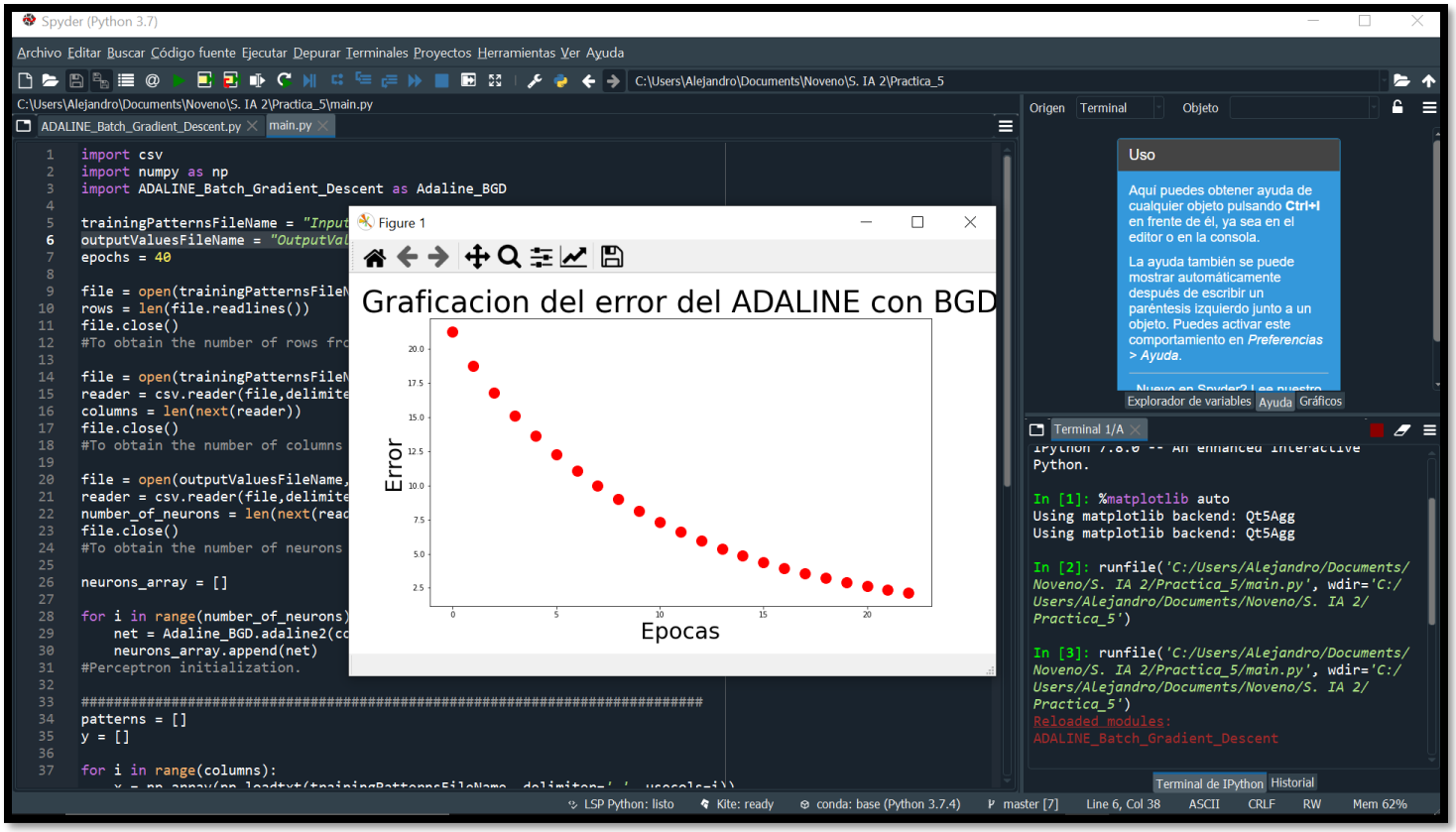
OutputValues1.csv - Excel

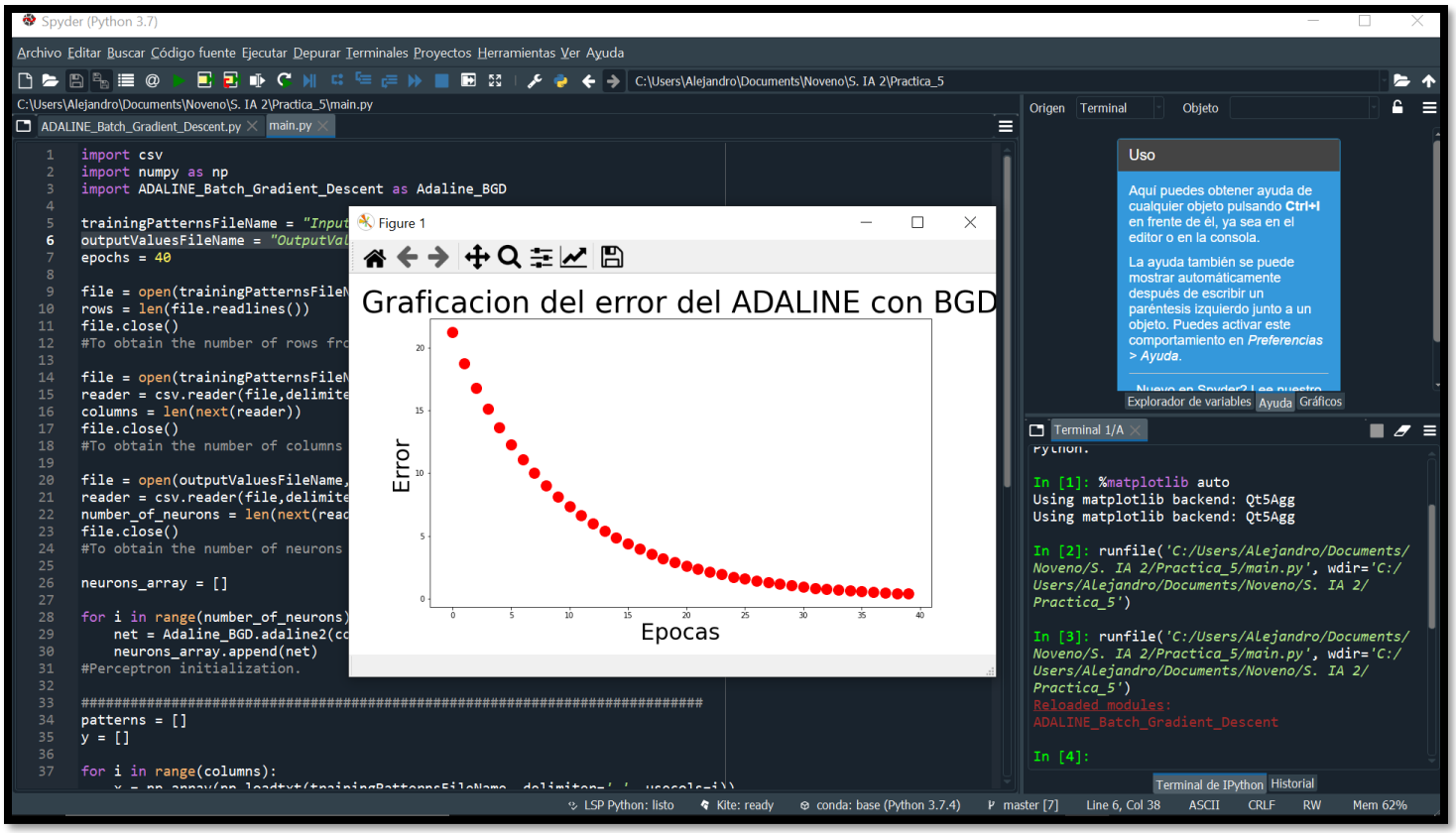
	A	B	C	D
1	0	1	0	0
2	0	1	0	0
3	0	0	1	0
4	0	1	0	0
5	0	0	1	0
6	0	0	1	0
7	0	0	0	1
8	0	0	0	1
9	0	0	0	1
10	1	0	0	0
11	1	0	0	0
12	0	0	0	1
13	0	0	0	1
14	0	0	1	0
15	0	0	0	1
16	1	0	0	0
17	0	0	0	1
18	1	0	0	0
19	0	0	0	1
20	0	0	0	1
21	1	0	0	0
22	0	0	1	0
23	1	0	0	0
24	0	0	1	0

Como podemos observar, exitosamente los valores que debería ser 1 en el archivo de salidas deseadas son cercanos a 1 en el archivo de salidas predichas, mientras que aquellos que deberían ser ceros están bastante cercanos a dicho valor, por lo cual podemos concluir que el entrenamiento ha sido bastante favorable. Ahora observemos los resultados con el segundo dataset.

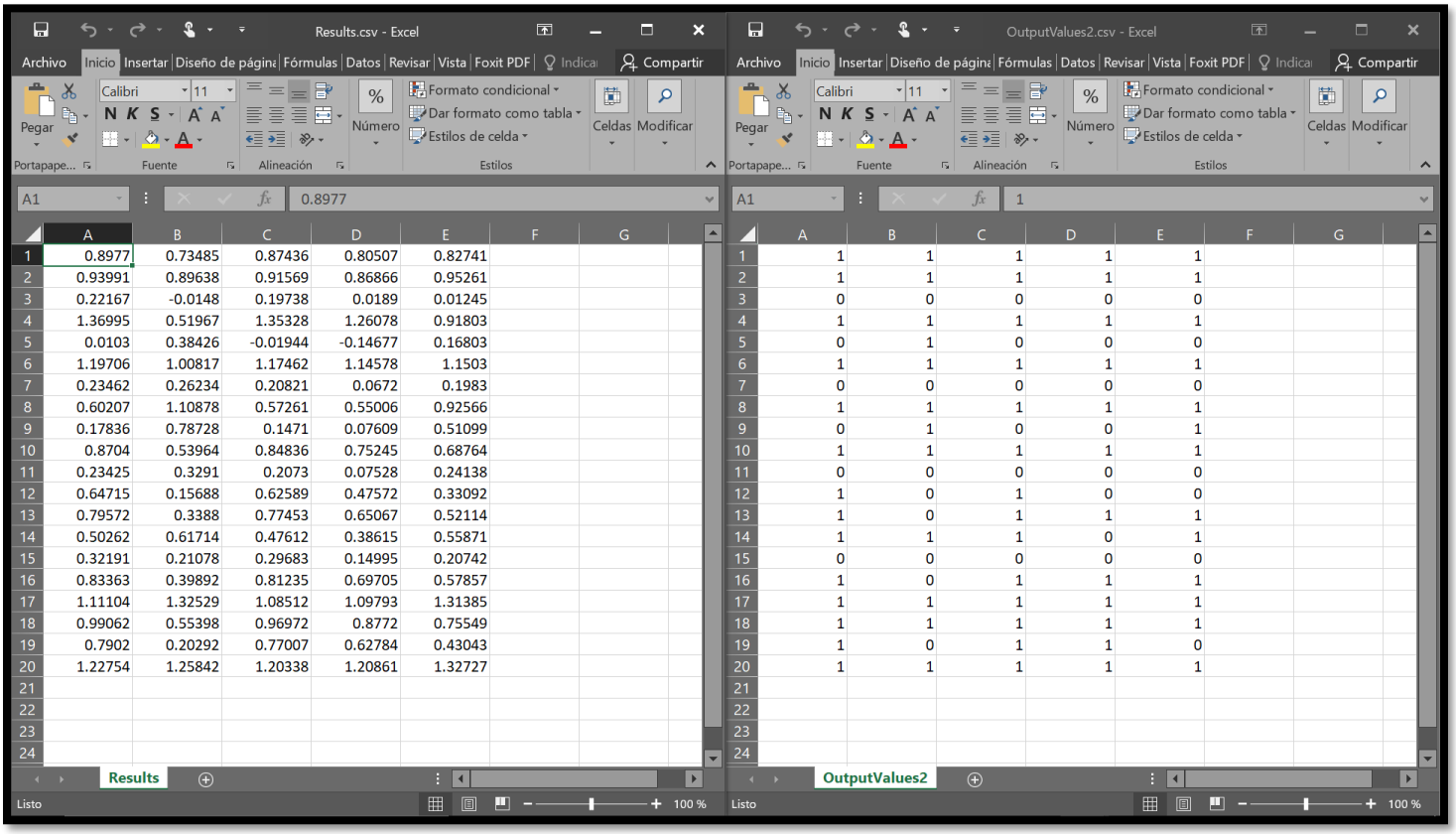


Este modelo igualmente es de dos dimensiones y tiene una cantidad mucho menor de datos de entrada; sin embargo, requiere de cinco neuronas para poder resolverse, por lo que se vuelve un poco más complejo. Observemos el entrenamiento y los resultados.





Una vez más podemos observar cómo en entrenamiento de las neuronas de este modelo ha sido exitoso, obteniendo resultados con error muy cercanos a cero. Veamos los CSV.



	A	B	C	D	E
1	0.8977	0.73485	0.87436	0.80507	0.82741
2	0.93991	0.89638	0.91569	0.86866	0.95261
3	0.22167	-0.0148	0.19738	0.0189	0.01245
4	1.36995	0.51967	1.35328	1.26078	0.91803
5	0.0103	0.38426	-0.01944	-0.14677	0.16803
6	1.19706	1.00817	1.17462	1.14578	1.1503
7	0.23462	0.26234	0.20821	0.0672	0.1983
8	0.60207	1.10878	0.57261	0.55006	0.92566
9	0.17836	0.78728	0.1471	0.07609	0.51099
10	0.8704	0.53964	0.84836	0.75245	0.68764
11	0.23425	0.3291	0.2073	0.07528	0.24138
12	0.64715	0.15688	0.62589	0.47572	0.33092
13	0.79572	0.3388	0.77453	0.65067	0.52114
14	0.50262	0.61714	0.47612	0.38615	0.55871
15	0.32191	0.21078	0.29683	0.14995	0.20742
16	0.83363	0.39892	0.81235	0.69705	0.57857
17	1.11104	1.32529	1.08512	1.09793	1.31385
18	0.99062	0.55398	0.96972	0.8772	0.75549
19	0.7902	0.20292	0.77007	0.62784	0.43043
20	1.22754	1.25842	1.20338	1.20861	1.32727

	A	B	C	D	E
1	1	1	1	1	1
2	1	1	1	1	1
3	0	0	0	0	0
4	1	1	1	1	1
5	0	1	0	0	0
6	1	1	1	1	1
7	0	0	0	0	0
8	1	1	1	1	1
9	0	1	0	0	1
10	1	1	1	1	1
11	0	0	0	0	0
12	1	0	1	0	0
13	1	0	1	1	1
14	1	1	1	0	1
15	0	0	0	0	0
16	1	0	1	1	1
17	1	1	1	1	1
18	1	1	1	1	1
19	1	0	1	1	0
20	1	1	1	1	1

Resulta nuevamente apreciable que los valores, si bien no son binarios, son bastante cercanos a los correspondientes que deberían de ser. De esta manera podemos entender que le modelo está entrenando de una manera satisfactoria.

Conclusión

El modelo aplicado en esta entrega difiere ligeramente con los anteriores estudiados; sin embargo, demuestra tener una ejecución bastante más eficiente, entregando resultados iguales o incluso mejores, por lo que se convierte en una opción más viable en general gracias a que trabaja con una función de optimización, y de esa manera nos evitamos estar fluctuando entre puntos altos y bajos como lo llegaba a hacer en ciertos momentos el perceptrón, llegando en ocasiones al escenario en que era virtualmente incapaz de aprender. En este caso corregimos este error y logramos obtener resultados completamente satisfactorios.

Código

ADALINE_Batch_Gradient_Descent.py

```
import numpy as np
import matplotlib.pyplot as plt

plt.ylabel('Error', fontsize=30)
plt.xlabel('Epocas', fontsize=30)
plt.title('Graficacion del error del ADALINE con BGD', fontsize=40)

def graphError(x_coordinate, y_coordinate):
    plt.scatter(x_coordinate, y_coordinate)
    plt.plot(x_coordinate, y_coordinate, 'ro', markersize=14)
    plt.pause(0.3)

class adaline2:
    def __init__(self, dimensions, learning_rate):
        self.dimensions = dimensions
        self.learning_rate = learning_rate
        self.w_vector = -1 + 2 * np.random.rand(dimensions, 1)
        self.b_value  = -1 + 2 * np.random.rand()

    def predict(self, x_vector):
        return np.dot(self.w_vector.transpose(), x_vector) + self.b_value

    def train(self, X_matrix, y_vector, epochs):
        n, m = X_matrix.shape
        error = 0.0
        for i in range(epochs):
            w_sumatory_vector = np.zeros((self.dimensions, 1))
            b_sumatory = 0

            for j in range(m):
```

```
            y_estimated = self.predict(X_matrix[:,j])
            error += y_vector[j] - y_estimated
            w_sumatory_vector += (y_vector[j] - y_estimated) * X_matrix[:,j].reshape(-1, 1)
            b_sumatory        += (y_vector[j] - y_estimated)
        self.w_vector += (self.learning_rate / m) * w_sumatory_vector
        self.b_value  += (self.learning_rate / m) * b_sumatory
        return error
```

main.py

```
import csv
import numpy as np
import ADALINE_Batch_Gradient_Descent as Adaline_BGD

trainingPatternsFileName = "InputValues2.csv"
outputValuesFileName = "OutputValues2.csv"
epochs = 40

file = open(trainingPatternsFileName)
rows = len(file.readlines())
file.close()
#To obtain the number of rows from the CSV file

file = open(trainingPatternsFileName,'r')
reader = csv.reader(file,delimiter=',')
columns = len(next(reader))
file.close()
#To obtain the number of columns from the CSV file

file = open(outputValuesFileName,'r')
reader = csv.reader(file,delimiter=',')
number_of_neurons = len(next(reader))
file.close()
#To obtain the number of neurons for the program. The number of output columns tells us the number of neurons.

neurons_array = []

for i in range(number_of_neurons):
    net = Adaline_BGD.adaline2(columns, 0.1)
    neurons_array.append(net)
#Perceptron initialization.

##############################################################################
patterns = []
y = []

for i in range(columns):
    x = np.array(np.loadtxt(trainingPatternsFileName, delimiter=',', usecols=i))
    patterns.append(x)
X = np.array(patterns)
```

```
for i in range(number_of_neurons):
    y.append(np.array(np.loadtxt(outputValuesFileName, delimiter=',', usecols=i)))
#Obtaining training patterns in X and output values in y.
#############################################################################

global_errors = []
individual_error = 0.0
results = []

for i in range(epochs):
    for j in range(number_of_neurons):
        net = neurons_array[j]
        individual_error += net.train(X, y[j], 1)
    individual_error /= number_of_neurons
    global_errors.append(individual_error)
    individual_error = 0.0
    Adaline_BGD.graphError(i, global_errors[i][0])


for i in range(number_of_neurons):
    net = neurons_array[i]
    # individual_result = np.concatenate(net.predict(X).tolist())
    individual_result = np.concatenate(net.predict(X))
    results.append(individual_result)

results = np.array(results).T
np.savetxt("Results.csv", results, delimiter=",", fmt='%.5f')
```

Link al repositorio

- https://github.com/TheGenesisX/S_IA_2/tree/master/Practica_5