Universidad Autónoma De Yucatán

Facultad de Matemáticas

Licenciatura en Ciencias de la Computación

Inteligencia Artificial

Proyecto Final

Aprendizaje Supervisado

Integrantes:

- Jesús Alberto Méndez Vela
- Pedro Manuel Ríos Méndez
- Ricardo Antonio Cervera Chacón

Profesora: Anabel Martín González

Contenido

Introducción.

Nuestro proyecto se basa en datos recopilados de habitaciones de hotel obtenidos por diferentes medidores (sensores de temperatura, de humedad, de luz, CO2). El objetivo de nuestro proyecto es poder clasificar exitosamente muestras de datos de dichos sensores en clases que representan cuando una habitación está ocupada o no.

Consideramos este tema para el desarrollo de nuestro proyecto pues nos pareció interesante el como dichas medidas se relacionan con el entorno y como este puede cambiar en gran o pequeña medida cuando una persona se encuentra en él.

Por otra parte, también nos pareció un tema bastante original (y sencillo de ejecutar) utilizando estrategias de inteligencia artificial que habíamos aprendido anteriormente en clase. Si bien, no consideramos que esta sea una problemática que afecte a la sociedad, sí consideramos que podría ser beneficioso o al menos útil para entornos de desarrollo tales como los de IoT (Internet de las cosas) o como en este caso, para poner en práctica la ejecución de lo aprendido a lo largo del curso.

Metodología.

Para este proyecto, principalmente habíamos planeado utilizar regresión logística (aunque en retrospectiva y tras la ejecución, no tenía mucho sentido nuestra opción) y por recomendación de la profesora decidimos elaborar una red neuronal simple.

Nuestra red neuronal podría considerarse un perceptrón, puesto que consiste en una única neurona artificial que realiza clasificaciones en 2 clases distintas.

Las clases que decidimos implementar para clasificar los datos (valga la redundancia) son la clase 0 para cuando una habitación no esté ocupada o la clase 1 para cuando lo esté.

Las entradas de datos de la neurona como se comentó anteriormente son 5, que constan en mediciones dadas por sensores de la habitación (temperatura, humedad, luz (en lúmenes), nivel de CO2 en el aire y el rango de humedad en el ambiente). En base a ellos decidimos implementar una única función (la función sigmoidal) que permitiera poder ejecutar los cálculos adecuados que nos permitieran lograr la tarea planteada.

La base de datos utilizada es consultable en el siguiente enlace:
https://archive.ics.uci.edu/dataset/357/occupancy+detection

Sin embargo, considero que es muy importante resaltar la utilización de una normalización para los datos de entrada. Esta decisión se tomó puesto que los datos eran muy dispersos, teniendo columnas de datos con promedio aproximado de 500 y otras columnas con datos menores a 1. La tarea de normalización se realizó utilizando técnicas de preprocesamiento de datos que describiremos más detalladamente en este documento.

Para esta tarea decidimos utilizar el lenguaje de programación Python con librerías adecuadas para el manejo de archivos separados por comas (csv, Pandas) y para la realización de operaciones matemáticas (NumPy).

Diseño de la red neuronal.

Primero decidimos utilizar una aproximación orientada a objetos para nuestra neurona, el resultado de dicho enfoque es el siguiente:

neuralNetwork.py > ...

```python
import numpy, random

class Perceptron:
    # Constructor.
    def __init__(self, inputsNumber):
        self.inputsNumber = inputsNumber
        self.bias = 1
        self.weights = [0] * inputsNumber

```

Como se puede observar decidimos llamar a la clase Perceptron por motivos antes mencionados en la introducción e implementamos métodos que resultaran útiles durante la implementación.

Nuestro modelo de Perceptron tiene como atributos los pesos de las entradas y el umbral representadas por la variable *weights,* y por la variable *bias* respectivamente. Así como un atributo llamado *inputsNumber* que sirve como variable de control para la cantidad de entradas que recibe nuestra neurona (esto en caso de requerir modificaciones posteriormente en el número de entradas o para su posterior reutilización en proyectos personales).

En cuanto a los métodos, podemos mencionar los siguientes:

```python
    # Función que inicializa los pesos para la red neuronal.
    def initializeWeights(self, weights=[]):
        if not weights:
            for i in range(len(self.weights)):
                self.weights[i] = random.uniform(-1, 1, )
        else:
            for i in range(len(self.weights)):
                self.weights[i] = weights[i]
```

El método *initializeWeights()* se utiliza para inicializar el valor de los pesos asignados a cada una de las unidades de entrada correspondientes. En caso de ser llamado sin un parámetro, se generan pesos aleatorios (esto para cuando sea el entrenamiento de nuestra red neuronal); por otra parte, cuando se le pasa una lista de pesos ya establecidos, estos se toman como los pesos para las unidades de entrada de la red neuronal.

```python
    # Función de activación (sigmoidal).
    def sigmoidFunction(self, inputs):
        result = numpy.dot(inputs, self.weights) + self.bias
        return 1 / (1 + numpy.exp(-result))
```

El método *sigmoidFunction()* implementa la función sigmoidal para una entrada dada, esto mediante el calculo del producto punto de las entradas con sus respectivos pesos y la posterior suma del bias.

```
30      # Función que cálcula el error.
31      def calculateError(self, output, expected):
32          error = output * (1 - output) * (expected - output)
33          return error
```

La función *calculateError()* es la función que calcula el error entre una salida dada y la salida real esperada para ese conjunto de datos. Esta función se llama cuando se genera un resultado de clasificación mediante el uso de la función sigmoidal y retorna su resultado para ser almacenado en un vector de errores que sirve para calcular el error general del algoritmo hasta el momento.

Entrenamiento.

Para el entrenamiento, primero es necesario cargar los datos desde el archivo descargado al lenguaje Python, para ello nos servimos de la librería Pandas que nos permite manejar archivos .csv de manera óptima, además de poder utilizarse en conjunto con la librería *sklearn* que nos otorga funciones para la normalización de los datos columna a columna.

```
1   # Lectura de datos desde el archivo .csv
2   from neuralNetwork import Perceptron
3   import pandas
4   from sklearn import preprocessing
5
6   def leerCsvNorm(nombreArchivo):
7       labels = []
8       data = []
9       results = []
10
11      datosArchivo = pandas.read_csv(nombreArchivo)
12      dataframe = pandas.DataFrame(datosArchivo)
13
14      scaler = preprocessing.StandardScaler()
15      dataframe[['Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio']] = scaler.fit_transform(dataframe[[
16
17      data = dataframe[['Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio']].values.tolist()
18      results = dataframe['Occupancy'].values.tolist()
19
20      return data, results
```

Como se puede observar, cargamos los datos (omitiendo las etiquetas correspondientes a las columnas) en los arreglos *data* y *results*, el primero contiene todas las filas de entradas de muestras de entrenamiento para la neurona y el segundo el valor esperado para la correspondiente muestra de entrenamiento.

Como se mencionó anteriormente, decidimos implementar la normalización en nuestros datos debido a que intentamos cargarlos directamente y ejecutar el algoritmo pero la dispersión entre los datos de cada columna ocasionaba que los valores de los pesos se dispararan y nunca hubiera una verdadera tasa de aprendizaje, manteniendo el error constante sin importar la tasa de entrenamiento o el cambio en las épocas que mantenía el entrenamiento.

Para la normalización de los datos se utiliza la función fit_transform de la librería sklearn que utiliza la fórmula

$$z = \frac{x - \mu}{\sigma} \qquad \mu = \text{Mean} \qquad \sigma = \text{Standard Deviation}$$

Donde al valor real se le resta la media y el resultado de dicha operación se divide entre la desviación estándar de los datos de la columna para obtener un valor *z* que sirve como valor normalizado con la finalidad de mantener una “armonía” o “cercanía” entre los datos que favorezca el desarrollo del entrenamiento de la red neuronal.

```
22  # ---------- EJECUCIÓN PRINCIPAL ----------
23  # Resultados obtenidos de la red neuronal.
24  results = []
25  # Errores obtenidos.
26  errors = []
27
28  # trainingData, realResults = leerCsv("datatraining.csv")
29  # Lectura del archivo .csv con datos de entrenamiento normalizados.
30  trainingData, realResults = leerCsvNorm("datatraining.csv")
31
32  # Se crea la neurona con 5 unidades de entrada.
33  perceptron = Perceptron(5)
34
35  # Inicializamos los pesos al azar para las entradas.
36  perceptron.initializeWeights()
```

La ejecución principal se basa en leer los datos del archivo de entrenamiento en formato .csv y cargar los datos en los arreglos correspondientes. A su vez, se inicia el perceptrón con las 5 unidades de entrada y se llama a la función que inicializa los pesos de manera aleatoria.

```
38  # Entrenamiento de la red.
39  for i in range(251):
40      print(perceptron.weights, "bias =", perceptron.bias)
41      errors.clear()
42      if i == 250:
43          for j in range(len(trainingData)):
44              print(perceptron.sigmoidFunction(trainingData[j]), realResults[j])
45          print("")
46      for j in range(len(trainingData)):
47          result = perceptron.sigmoidFunction(trainingData[j])
48          error = perceptron.calculateError(result, realResults[j])
49          errors.append(error)
50          perceptron.weights[0] += 0.01 * trainingData[j][0] * error
51          perceptron.weights[1] += 0.01 * trainingData[j][1] * error
52          perceptron.weights[2] += 0.01 * trainingData[j][2] * error
53          perceptron.weights[3] += 0.01 * trainingData[j][3] * error
54          perceptron.weights[4] += 0.01 * trainingData[j][4] * error
55          perceptron.bias += 0.01 * error
56
57      print("El error en la epoca", i, "fue de", 100 / len(trainingData) * sum(errors), "\n")
```

Para el entrenamiento, decidimos utilizar un total de 250 épocas (espero no sean demasiadas) y como líneas de control decidimos imprimir los pesos correspondientes, el bias, así como el error total generado respecto a los valores esperados al final de la iteración.

Cuando se llega a la última iteración, se muestran los valores obtenidos por el algoritmo junto con los valores esperados para cada una de las muestras de entrenamiento.

Para el desarrollo de cada muestra de entrenamiento, primero se calcula el resultado de pasarlo por la función sigmoidal, posteriormente se calcula la diferencia del valor obtenido con el valor esperado y se agrega al arreglo de errores.

Para este entrenamiento, se utilizó el algoritmo de backpropagation y decidimos actualizar los pesos y el bias sumándole el ajuste del algoritmo con una tasa fija de aprendizaje de 0.01 con la esperanza de poder evitar un overfitting de los hiper parámetros.

```
59  # Guardar los pesos conseguidos en un archivo de texto.
60  weightsFile = open("weights.csv", "w")
61  for i in range(len(perceptron.weights)):
62      weightsFile.write("weight{}, ".format(i))
63  weightsFile.write("bias\n")
64  for weight in perceptron.weights:
65      weightsFile.write(str(weight) + ", ")
66  weightsFile.write(str(perceptron.bias))
67  weightsFile.close()
68  print("Los pesos y el bias generados se han guardado correctamente en el archivo 'weights.csv'.")
```

Finalmente, cuando se termina el entrenamiento, decidimos guardar los pesos y el umbral obtenido en un archivo .csv que nos permitiera poder mantener estos datos al finalizar el entrenamiento para su posterior consulta y uso durante el programa *test.py* que ejecuta el algoritmo y calcula su error respecto al archivo que contiene las muestras de prueba y validación.

Prueba y validación.

Nuestra base de datos contenía dos archivos adicionales para la prueba y validación de nuestra implementación. Para este apartado decidimos realizar un script aparte con el nombre *test*.py que permita poder ejecutar la prueba con los valores otorgados para los pesos y el umbral por el archivo de entrenamiento.

```
1   # Lectura de datos desde el archivo .csv
2   import csv
3   import pandas
4   from neuralNetwork import Perceptron
5   from sklearn import preprocessing
6
7   # Función que normaliza los datos del archivo .csv leído.
8   def leerCsvNorm(nombreArchivo):
9       labels = []
10      data = []
11      results = []
12
13      datosArchivo = pandas.read_csv(nombreArchivo)
14      dataframe = pandas.DataFrame(datosArchivo)
15
16      scaler = preprocessing.StandardScaler()
17      dataframe[['Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio']] = scaler.fit_transform(
18
19      data = dataframe[['Temperature', 'Humidity', 'Light', 'CO2', 'HumidityRatio']].values.tolist()
20      results = dataframe['Occupancy'].values.tolist()
21
22      return data, results
```

Para ello, utilizamos nuevamente la función *leerCsvNorm()* que nos permite leer el archivo de entrenamiento y obtener la normalización de cada una de las columnas de datos (El uso de esta función es debido a que se intentó cargar los datos de los archivos de prueba sin alteraciones y ejecutar las correspondientes pruebas, pero debido a que se obtenían valores de error muy grandes, se decidió probar normalizando nuevamente el conjunto de prueba y esto resultó en una mejora considerable).

También implementamos una segunda función que nos permita leer archivos .csv para leer los datos correspondientes a los pesos de la red neuronal así como al bias obtenido por la ejecución del entrenamiento.

```python
def leerCsv(nombreArchivo, modo):
    labels = []
    data = []
    results = []

    with open(nombreArchivo, "r") as csvFile:
        fileReader = csv.reader(csvFile)

        # Omitimos la fila de las etiquetas.
        labels = next(fileReader)

        for row in fileReader:
            # Control de la variables.
            if modo == 2:
                *rowData, bias = row
                rowData = list(map(float, rowData))
                bias = float(bias)
                return rowData, bias
            # Sintaxis de desestructuración.
            number, date, *rowData, result = row

            # Transformar los datos del CSV a float.
            rowData = list(map(float, rowData))
            # Agregando datos y resultados a los arrays.
            data.append(rowData)
            results.append(float(result))

    # Mostar el número de filas de datos leídas.
    print("Se han leido {} filas de datos del archivo {}".format(fileReader.line_num - 1, nombreArchivo))

# Se retornan los datos de las columnas y los resultados en arreglos separados.
return data, results
```

Esta función no se describe a detalle puesto que es prácticamente idéntica a la anterior, sin embargo, esta no normaliza los datos de ninguna manera al momento de cargarlos.

Al tener dos archivos para prueba y validación, diseñamos el programa de tal manera que pueda recibir una entrada del usuario para seleccionar el conjunto de datos a utilizar en la prueba:

```
58  # Array declarado para acumular los errores.
59  errors = []
60
61  # Selección del archivo csv de prueba.
62  print("Ingrese el número del conjunto de datos de prueba con el que desea trabajar (1 ó 2): ")
63  seleccion = int(input())
64  nombreArchivo = "datatest.csv" if (seleccion == 1) else "datatest2.csv"
65
```

Tras la selección, se cargan los datos de prueba, el bias y los pesos mediante la llamada a la función que lee el archivo .csv generado en la sección de entrenamiento:

```
67  # Se cargan los datos de prueba.
68  testData, realResults = leerCsvNorm(nombreArchivo)
69
70  # Cargar los pesos y el bias generados por el script de entrenamiento.
71  weights, bias = leerCsv("weights.csv", 2)
72
73  # Se inicializa la neurona con los valores ya calculados.
74  perceptron = Perceptron(5)
75  perceptron.initializeWeights(weights)
76  perceptron.bias = bias
77
```

Se inicializa la neurona con los hiper parámetros antes mencionados y con un total de cinco unidades de entrada.

```
79  print(perceptron.weights, "bias =", perceptron.bias, "\n")
80  for j in range(len(testData)):
81      result = perceptron.sigmoidFunction(testData[j])
82      print("j =", j, "; y* =", result, "; y =", realResults[j])
83      error = pow(result - realResults[j], 2)
84      errors.append(error)
85
86  print("\nEl error del algoritmo fue de", 100 / len(testData) * sum(errors), "\n")
87
```

Para cada una de las muestras del conjunto de prueba se calcula el resultado utilizando el perceptrón y se calcula el error utilizando la técnica del error cuadrado medio que nos permite observar de mejor manera el error que se obtiene de nuestro modelo de clasificación.

Finalmente, se imprime en consola cual fue el error obtenido para la ejecución del algoritmo en porcentaje.

Resultados obtenidos

Entrenamiento.

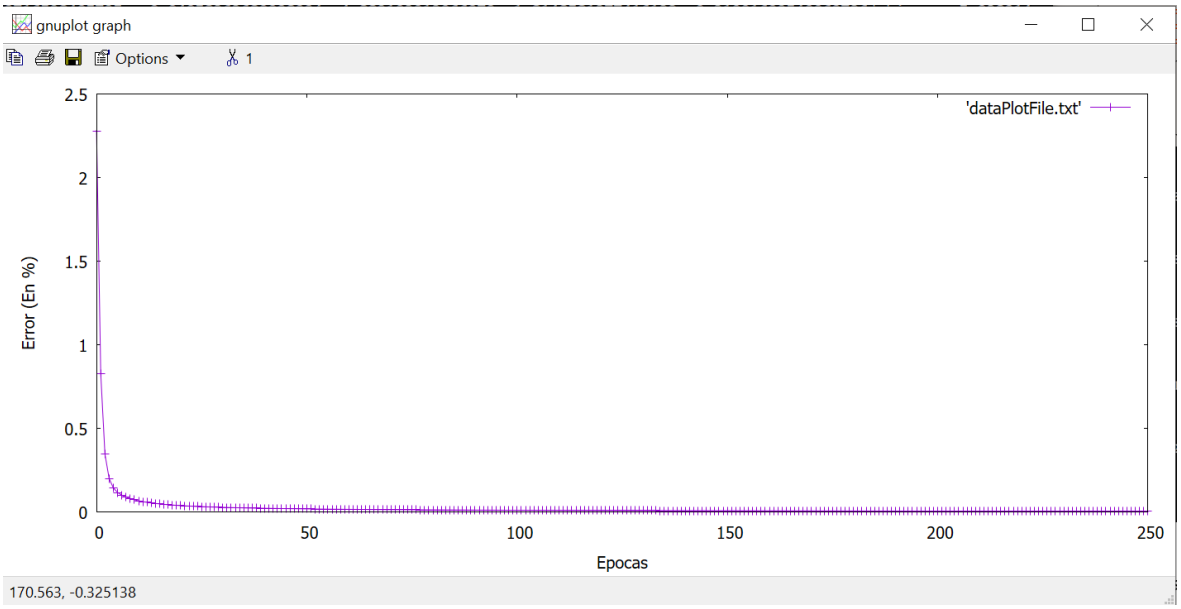
Los resultados al ejecutar el entrenamiento de la red neuronal se presentan de la siguiente manera, pudiendo observar el valor del error para todas las muestras así como el cambio que se genera en los parámetros de la red (los pesos y el bias).

Los 5 valores dentro de la lista corresponden a los pesos y el bias se encuentra señalado en la misma fila.

```
train.py
[0.6629289785363177, 0.5033661294627807, -0.3235297730565754, 0.44729624868559736, 0.7491466390151269] bias = 1
El error en la epoca 0 fue de 2.279472280359676 %

[0.6274875357428847, -0.03367067840419911, 1.287054290555094, 1.1554618386194975, 0.23748641945225543] bias = -0.8561742
778968955
El error en la epoca 1 fue de 0.8263146188575224 %

[0.044004632633746904, -0.044663984227403176, 1.5680986704577395, 0.9880339732880002, 0.07020429780744371] bias = -1.529
0422720325771
El error en la epoca 2 fue de 0.34657217387788036 %

[-0.2104764752901379, -0.04019796273245661, 1.7756723038129738, 0.866511567051452, 0.0076181193798741074] bias = -1.8112
55993221332
El error en la epoca 3 fue de 0.20089866096820042 %

[-0.33556120076321766, -0.012331821433770325, 1.9588399085943589, 0.841805247689479, 0.006016826794450838] bias = -1.974
847772847744
El error en la epoca 4 fue de 0.144385530947399 %

[-0.4119729514845803, 0.010577621042025753, 2.1119650265097216, 0.8506018689676748, 0.013677605188929041] bias = -2.0924
209106982055
El error en la epoca 5 fue de 0.11783204732933313 %

[-0.46802886463187, 0.02694491925141444, 2.237689607086817, 0.8699697573358309, 0.02048288209637957] bias = -2.188371546
838482
El error en la epoca 6 fue de 0.10149677867451104 %
```

```
El error en la epoca 244 fue de 0.005343455914734731 %

[-0.9249716528405725, -0.04058363680098994, 4.882103013066979, 1.8148581377416133, 0.31001692540069736] bias = -5.038914
421528731
El error en la epoca 245 fue de 0.00532455131807508 %

[-0.9250813793176393, -0.04100430947132372, 4.885760136487373, 1.8161007135430385, 0.31080326237341227] bias = -5.043250
203667078
El error en la epoca 246 fue de 0.0053057843725702394 %

[-0.9251910083304472, -0.04142443802898402, 4.889404443914518, 1.8173396311570469, 0.3115872249826708] bias = -5.0475707
03881656
El error en la epoca 247 fue de 0.005287153538869351 %

[-0.9253005382245805, -0.04184402196438981, 4.893036026278423, 1.818574913732342, 0.31236882851350817] bias = -5.0518760
33008357
El error en la epoca 248 fue de 0.005268657300750343 %

[-0.9254099673692596, -0.04226306079943444, 4.8966549735079985, 1.8198065841628404, 0.3131480880979808] bias = -5.056166
300648363
El error en la epoca 249 fue de 0.0052502941647074265 %

[-0.9255192941572145, -0.04268155408685764, 4.900261374546107, 1.8210346650918865, 0.3139250187171679] bias = -5.0604416
15186689
El error en la epoca 250 fue de 0.005232062659524634 %

Los pesos y el bias generados se han guardado correctamente en el archivo 'weights.csv'.
```

Cuando se terminan de recorrer las 250 épocas, se muestra el error final, así como un mensaje que indica que se han guardado los pesos y el umbral en el archivo *weights.csv*. Así como también se ejecuta una línea de código que llama al software graficador de Gnuplot que nos muestra como cambia el error respecto a las épocas.

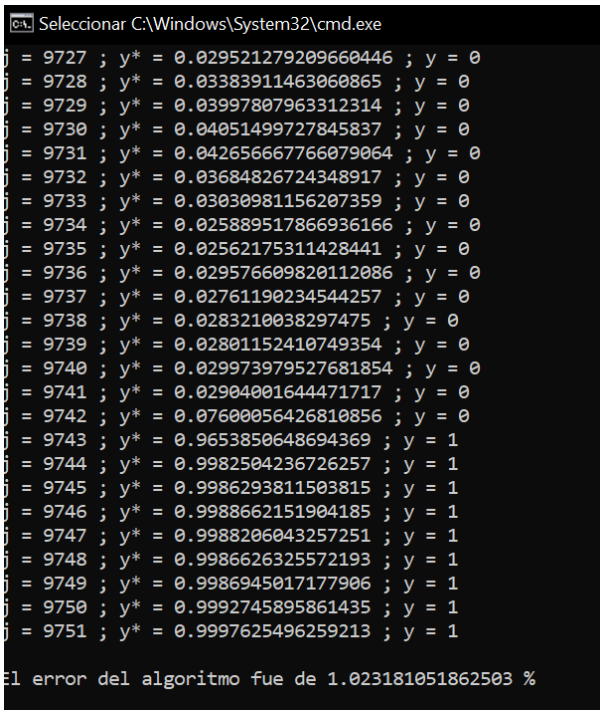


Pruebas.

Por otra parte, para las pruebas se decidió que se mostraran los valores obtenidos por la implementación junto con el valor esperado para poder comparar de manera directa los resultados obtenidos.

```
aArtificial\ProyectoFinal>python test.py
Ingrese el número del conjunto de datos de prueba con el que desea trabajar (1 ó 2):
2

[-0.925628517004569, -0.0430995014096393, 4.9038553173645205, 1.8222591789164666, 0.314
69963520312455] bias = -5.064702083810333

j = 0 ; y* = 0.9730816547364223 ; y = 1
j = 1 ; y* = 0.9668361117789096 ; y = 1
j = 2 ; y* = 0.9659021854370408 ; y = 1
j = 3 ; y* = 0.9705983233055148 ; y = 1
j = 4 ; y* = 0.9682852220213936 ; y = 1
j = 5 ; y* = 0.9708136113879101 ; y = 1
j = 6 ; y* = 0.9684599456252377 ; y = 1
j = 7 ; y* = 0.971272463575564 ; y = 1
j = 8 ; y* = 0.9694153373248008 ; y = 1
j = 9 ; y* = 0.9728778320626243 ; y = 1
j = 10 ; y* = 0.9763448711453602 ; y = 1
j = 11 ; y* = 0.978676578438474 ; y = 1
j = 12 ; y* = 0.9800336722690312 ; y = 1
j = 13 ; y* = 0.9783479256142545 ; y = 1
j = 14 ; y* = 0.9782882196047612 ; y = 1
j = 15 ; y* = 0.9762273374067895 ; y = 1
j = 16 ; y* = 0.9819415797829018 ; y = 1
j = 17 ; y* = 0.9821827473280365 ; y = 1
j = 18 ; y* = 0.9832290023018934 ; y = 1
j = 19 ; y* = 0.9836351455470483 ; y = 1
```

Primero se muestran los pesos y el bias cargados, posteriormente se muestran los valores obtenidos para cada una de las muestras representadas con el símbolo *y** para las aproximaciones dadas por la red neuronal y *y* para las salidas reales esperadas dadas por el conjunto de prueba.



Al finalizar, se muestra el valor obtenido para el error del algoritmo en porcentaje.

Conclusiones

Observando los resultados obtenidos por nuestra red neuronal para el entrenamiento, notamos un patrón donde siempre el tercer peso resulta ser el más alto de todos y el cuarto el más bajo, suponemos que esto se debe a que son los que tienen mayor y menor impacto sobre el resultado final respectivamente.

Por otra parte, es notable como incluso al inicializar los pesos al azar, el error resulta bastante bajo, lo que suponemos que puede ser incorrecto y es quizá una consecuencia de una "sobrenormalización" de los datos de entrada. También es destacable señalar que siempre que iterábamos con los datos sin alteraciones obteníamos un resultado totalmente nulo respecto al aprendizaje de nuestra red, como si simplemente no se generara ningún error incluso cuando la clase generada era totalmente opuesta a la correcta, esto debido al algoritmo de calculo de error implementado en la función que utilizamos.

En cuanto a los resultados aportados por el conjunto de prueba es notable como estos no son directamente salidas esperadas para la clasificación (no nos indican clase 0 o 1), sino que representan más una probabilidad de que la clase evaluada sea perteneciente a la clase 1, pues para la clase 1 se obtienen valores muy cercanos a 1, mientras que para las muestras de clase 0 se obtienen resultados muy cercanos a 0, con solamente unas décimas de diferencia.

Consideramos que nuestro proyecto es un éxito parcial pues no creímos poder implementarlo, pero es funcional, pese a las pegas que se han mencionado con anterioridad.

Quizá con un poco más de tiempo, estudio y trabajo, este proyecto pueda ser un éxito.