ITESO, Universidad Jesuita de Guadalajara

InstitutoTecnológico y de Estudios Superiores de Occidente

Maestría Ciencia de Datos

Investigación, Desarrollo e Innovación 2

Tarea 7: Aplicación perceptrón multicapa

Estudiante: Daniel Nuño
Profesor: Fernando Becerra
Fecha entrega: Octubre 23, 2021

En la tabla siguiente se muestran los datos de 1000 clientes que solicitaron créditos a un banco dado. La última columna muestra la información de los clientes que cayeron en mora en algún momento del período del crédito. El monto máximo de crédito que puede asignarse son $300,000 y la antigüedad laboral máxima que se toma en cuenta para asignar el crédito es de 15 años (es decir, antigüedades mayores ya no generan más posibilidad de ser aprobado).

Se busca una relación entre la información presentada (que se obtiene al contratar el crédito) y la posibilidad de que el cliente caiga en mora en algún momento del plazo.

Entrene un perceptrón multicapa para encontrar una relación tomando como entradas el monto solicitado (normalizado), la carga que implica al salario el pago de la mensualidad y la antigüedad laboral al contratar (normalizada).

Utilice 70-30 de relación entrenamiento-prueba y calcule el accuracy.

El número de neuronas ocultas es a su criterio.

In [33]:

```
import pandas as pd
import numpy as np

def train_one_layer_perceptron(training_data, training_output, L, a, alpha, accuracy
    # INITIALIZE values and functions for one hidden layer
    a = a #a = inclinación de la sigmoide tiene que ser mayor a 0
    alpha = alpha #alpha learning rate tiene que ser entre 0 y 1
    L = L #number of neurons per layer
    accuracy = accuracy
    error = 1

    x = training_data #Q * N matrix. Q = rows, N = columns
    d = training_output #Q * M matrix. Q = rows, M = columns
    N = training_data.shape[1] #number of Xs = entries columns
    M = training_output.shape[1] #number of Ys = number of columns outputs
    Q = training_data.shape[0] #different inputs with known outputs = number of rows

    epoch_count = 0
    np.random.seed(1)
    weights_ItoH = np.random.uniform(-1, 1, (L, N)) #L x N size
    weights_HtoO = np.random.uniform(-1, 1, (M, L)) #M x L size

    while error > accuracy and (epoch_count < max_epochs):
        for row in range(Q):
            #feedforward
            net_h = weights_ItoH @ np.reshape(x[row,:], (N,1))
            y_h = logistic(net_h,a)
            net_o = weights_HtoO @ np.reshape(y_h, (L,1))
            y_o = logistic(net_o,a)

            #backforward
            delta_o = np.reshape( d[row,:] - np.reshape(y_o, (1,M)), (M,1)) * (y_o *
            delta_h = (y_o * (1 - y_o)) * (np.transpose(weights_HtoO) @ delta_o)
            weights_HtoO += alpha * (delta_o * np.transpose(y_h))
            weights_ItoH += alpha * (delta_h * training_data[row,:])

            #error
            error = np.linalg.norm(delta_o)
        epoch_count +=1
    return (weights_HtoO, weights_ItoH, error, epoch_count)

def logistic(x,a):
    return 1/(1 + np.exp(-a*x))
```

```
def logistic_deriv(x,a):
    return a*logistic(x,a) * (1 - logistic(x,a))

def evaluate_one_layer_perceptron(input_matrix, weights_HtoO, weights_ItoH, a):
    # INITIALIZE values
    x = input_matrix
    weights_HtoO = weights_HtoO
    weights_ItoH = weights_ItoH
    a = a
    N = x.shape[1] #number of Xs = entries columns
    Q = x.shape[0] #different inputs with known outputs = number of rows

    output_matrix = []
    output_row = []
    y_matrix = []

    #cicle through rows to evaluate according to weights
    for row in range(Q):
        net_h = weights_ItoH @ np.reshape(x[row,:], (N,1))
        y_h = logistic(net_h,a)
        net_o = weights_HtoO @ np.reshape(y_h, (L,1))
        y_o = logistic(net_o,a)
        # from y's hidden to outputs define if is 0 or 1 and append to rows outputs
        for y in y_o:
            if y > 0.5:
                output_row.append(1)
            else:
                output_row.append(0)

        output_matrix.append(output_row) #append row outputs in a matrix
        output_row = [] #clear to be used again
        y_matrix.append(y_o) #append return

    output_matrix = np.asarray(output_matrix, dtype=np.float64)
    y_matrix = np.asarray(y_matrix, dtype=np.float64)

    return np.block([x, output_matrix]), output_matrix, y_matrix


data = pd.read_excel('C:/Users/nuno/OneDrive - ITESO/Ciencia de Datos'
                     '/IDI II/PercMultAplicado_tarea7.xlsx')
data.head()
```

Out[33]:

	Entidad	Monto	Mensualidad	Plazo (años)	Tasa anual	Ingreso mensual	Antigüedad laboral (meses)	Mora
0	Sinaloa	299200	6277.86	20	0.25	33911.00	58	SI
1	Michoacán de Ocampo	281100	9373.58	20	0.40	112783.48	149	NO
2	Nuevo León	268800	8963.43	20	0.40	33186.96	134	NO
3	Guerrero	256600	9106.81	5	0.35	51118.90	77	NO
4	Yucatán	256500	5381.92	20	0.25	197168.90	5	SI

La table contiene la **Entidad**, **Monto**, **Mensualidad**, **Plazo (años)**, **Tasa anual**, **Ingreso mensual**, **Antigüedad laboral (meses)** que se pueden considerar variables de entrada que determinan **Mora**. Como discutido en clase tenemos que preprocesar los datos.

Primero Entidad, aunque puede ser aportar para saber si el crédito sera moratorio no la podemos usar porque no es un número. Convertirla en número consecutivo no es buena idea por que implicaría que una Entidad tiene mayor peso que otra sin razón aparente. Tendríamos que utilizar alguna técnica estadística para convertirlo a número.

Segundo, mensualidad es dependiente y resultado del Plazo (años), Tasa anual y Monto. Plaza (años) y Tasa anual la podemos descartar.

Tercero, sabemos que Monto y Antigüedad laboral (meses) están limitados a $300,000 y 180 meses (15 años x 12 meses) respectivamente. Normalizamos ambos para que los valores sean (0 a 1).

Normalizamos con una regla de 3, lo cual nos da resultados como si fuera una distribución lineal, aunque no es necesariamente la mejor aproximación ya que los datos podrían no estar distribuidos linealmente. Podrían aproximarse más a una exponencial, logística, normal, etc.

Cuarto, carga al salario se define como la Mensualidad / Ingreso mensual. Nos ayuda porque nos da valores entre 0 y 1 y, contiene información de la mensualidad y el ingreso mensual en términos del tamaño de la deuda para su capacidad.

Quinto, transformar la columna Mora a 0 y 1. SI equivale a 1 y NO es igual a 0.

La final tenemos solo las siguientes columnas como entradas y salida.

In [34]:
```
data['carga_al_salario'] = data.Mensualidad / data['Ingreso mensual']
data.Monto = data.Monto / 300000
data['Antigüedad laboral (meses)'] = data['Antigüedad laboral (meses)'] / 180
data['Mora'] = data['Mora'].replace('SI', 1)
data['Mora'] = data['Mora'].replace('NO', 0)
data = data.drop(['Entidad',
                  'Plazo (años)',
                  'Tasa anual',
                  'Mensualidad',
                  'Ingreso mensual'], axis = 1)
data.head()
```

Out[34]:

	Monto	Antigüedad laboral (meses)	Mora	carga_al_salario
0	0.997333	0.322222	1	0.185128
1	0.937000	0.827778	0	0.083111
2	0.896000	0.744444	0	0.270089
3	0.855333	0.427778	0	0.178150
4	0.855000	0.027778	1	0.027296

Ahora definimos entrenamiento y prueba.

In [35]:
```
training_data = data.sample(frac=0.7)
test_data = data.drop(training_data.index)

training_output = training_data[['Mora']]
training_output = np.asarray(training_output, dtype=np.float64)
training_data = training_data[['Monto', 'Antigüedad laboral (meses)',
                               'carga_al_salario']]
training_data = np.asarray(training_data,dtype=np.float64)
```

```
test_output = test_data[['Mora']]
test_output = np.asarray(test_output, dtype=np.float64)
test_data = test_data[['Monto', 'Antigüedad laboral (meses)',
                       'carga_al_salario']]
test_data = np.asarray(test_data, dtype=np.float64)
```

Lo que quiero hacer primero es definir un límite inferior que indique el resultado que debo obtener con una estimación y modelo más sencillo o conocido o implementado. En este caso lo más sencillo seria definir dicho limite como lo más probable que suceda y darles ese valor a todos los resultados estimados; es decir que tan probable es que el crédito NO entre en mora.

In [36]:
```
base_limit = np.sum(test_output == 0) / len(test_output)
base_limit
```

Out[36]: 0.7466666666666667

Ya que tenemos nuestro límite de 74.3% (o 84.1% para todo el data set), queremos que nuestro modelo sea más preciso. De otra manera podríamos definir que todos los nuevos créditos no caerán en mora sin la necesidad de un perceptrón.

Primero voy a intentar una sencilla combinación en cuanto **a = 1**, **neuronas = 6** y **alpha = 0.1**

In [37]:
```
a = 1
alpha = 0.1
L = 6
accuracy = 0.0001
max_epochs = 100000

(weights_HtoO,
 weights_ItoH,
 error,
 epoch_count) = train_one_layer_perceptron(training_data,
                                           training_output,
                                           L,
                                           a,
                                           alpha,
                                           accuracy,
                                           max_epochs)

(x_and_y,
 transformed_y,
 direct_y) = evaluate_one_layer_perceptron(test_data,
                                           weights_HtoO,
                                           weights_ItoH,
                                           a)
```

Habiendo entrenado y evaluado, sigue calcular la precisión.

direct_y es el vector de salida evaluado con el perceptrón, transformed_y es la transformación a 0 ó 1.

In [38]:
```
vector_accuracy = test_output == transformed_y
accuracy_validation = vector_accuracy.sum() / len(test_data)
accuracy_validation
```

Out[38]: 0.9966666666666667

También queremos saber cuántas veces el perceptrón es indeciso, porque no está muy seguro que el resultado sea 1 o que sea 0 considerando los resultados como probabilidad p y $(1-p)$. Para eso (arbitrariamente) definimos el intervalo 0.1 a 0.9

In [39]:
```
less_sure = np.sum(direct_y[direct_y > 0.1] < 0.9) / len(test_data)
less_sure
```

Out[39]: 0.3466666666666667

El accuracy es bastante bueno cerca de 100%. En cuanto los resultados que no está muy seguro obtuvimos 20%.

Sabemos que queremos que nuestros resultados sean menos ambiguos, sin comprometer la precisión. Entonces ahora intentare con diferentes neuronas = 20, alpha = 0.1 y a = 10.

In [82]:
```
a = 10
alpha = 0.2
L = 12
accuracy = 0.001
max_epochs = 100000

(weights_HtoO,
 weights_ItoH,
 error,
 epoch_count) = train_one_layer_perceptron(training_data,
                                           training_output,
                                           L,
                                           a,
                                           alpha,
                                           accuracy,
                                           max_epochs)

(x_and_y,
 transformed_y,
 direct_y) = evaluate_one_layer_perceptron(test_data,
                                           weights_HtoO,
                                           weights_ItoH,
                                           a)
```

In [83]:
```
vector_accuracy = test_output == transformed_y
accuracy_validation = vector_accuracy.sum() / len(test_data)
accuracy_validation
```

Out[83]: 0.7466666666666667

In [84]:
```
less_sure = np.sum(direct_y[direct_y > 0.1] < 0.9) / len(test_data)
less_sure
```

Out[84]: 0.02

El accuracy 75% y los resultados que no está muy seguro 2%. Empeoro en el accuracy pero mejoro substancialmente en la confiabilidad.

Definitivamente una alta a y más neuronas empeoran el modelo porque está empujando todos los resultados a 0.