Informe_Practica_2

Victor Brown Sogorb

2024-11-24

Introducción

Las redes neuronales, especialmente aquellas implementadas en **Python** utilizando **bibliotecas especializadas**, han demostrado ser herramientas extremadamente poderosas para resolver problemas complejos en este ámbito, como la **clasificación de imágenes**, el **reconocimiento de objetos** y la **segmentación**.

En el núcleo de las **redes neuronales** se encuentra un concepto fundamental: el **perceptrón simple**. El **perceptrón** es un **modelo matemático** inspirado en el funcionamiento de las **neuronas biológicas**. Su objetivo es **clasificar datos** en dos categorías aplicando una **función lineal de decisión**. Aunque su capacidad de resolución es limitada a problemas **linealmente separables**, el **perceptrón** estableció las bases teóricas y prácticas para el desarrollo de **modelos más avanzados**.

En este trabajo, exploraremos cómo **implementar** y **utilizar redes neuronales** en **Python** para abordar la **clasificación** y **entrenamiento**, tanto de **perceptrones propios** como de aquellos provenientes del paquete **sklearn**.

En específico, entrenaremos los diferentes **perceptrones** para que sean capaces de predecir con exactitud qué **letra** representa una **imagen**. Estas imágenes representan **letras escritas a mano** y se encuentran en los archivos en el interior de la carpeta **Data**.

Objetivos del informe

En este informe se interpretaran los resultados obtenidos de cada una de las tareas en las que se divide la práctica y también se comentaran y explicarán las funciones implementadas.

La manera de obtener todos los datos esta realizada y explicada en los Notebooks correspondientes a cada tarea que se pueden encontrar en el repositorio https://github.com/vicBrownS/Practica_Perceptron_SI.

Tarea 1

En la **tarea 1**, cuyo **notebook** correspondiente es el archivo **Tarea_1_Notebook.ipynb**, se debe implementar la función **crea_diccionario**. Esta función tiene como objetivo crear un **diccionario** con el que poder acceder a todos los **caracteres** que se representan en las **imágenes**. Estos **caracteres** vienen dados en **código ASCII** y provienen del archivo **claves_ASCII.txt** en la carpeta **Data**.

Las explicaciones de cómo opera la función se hallan en los propios **comentarios** de la misma.

```
"""
Se implementará la función crea_diccionario
que relaciona una clave del 0 al 46 con el caracter ASCII que representa,
esta relación será dada por el archivo claves_ASCII
que se encuentra en la carpeta DATA.
"""
def crea_diccionario(archivo_claves):
    claves = open(archivo_claves, 'r') #Abre el archivo indicado por el path
    diccionario = {} #Instancia el diccionario donde se guardarán los datos

    for line in claves: #Recorre cada linea en el archivo
        #Se utiliza el metodo strip() para quitar los espacios en blanco
        #Se utiliza el metodo split() para dividir los dos números de la linea
        clave = int(line.strip().split()[0])  #Se accede al primer número
        valor = chr(int(line.strip().split()[1])) #Se accede al segundo número y se le convierte en cha
        diccionario[clave] = valor #Se añade al diccionario
    return diccionario
```

Tarea 2

En la **Tarea 2**, debemos implementar la función **getdataset()**, que devuelve la **matriz de datos X**, la cual contiene la información de las **imágenes**. Específicamente, esta **matriz** es de dos dimensiones, donde las dimensiones corresponden al **número de imágenes x número de píxeles**. Además, la función también devuelve la **matriz de datos y**, que es una **matriz columna** que contiene el **label** de cada **imagen**.

Existe una correspondencia **1 a 1** entre estas **matrices**, lo que será fundamental más adelante para obtener los **conjuntos de entrenamiento**. Esta función se encuentra en el archivo **Tarea_2.py**.

El código de implementación de la función es el siguiente:

```
def getdataset(images, labels, caracteres, num_pix):
    import numpy as np
    from skimage.transform import resize
    """ Obtiene los arrays de numpy con las imágenes y las etiquetas
    Parámetros
    ----------
        imagenes -- estructura de datos que contiene la información de cada una de las imágenes
        etiquetas -- estructura de datos que contiene la información de la clase a la que
        pertenece cada una de las imágenes
        caracteres -- diccionario que contiene la "traducción" a ASCII de cada una de las etiquetas
        num_pix -- valor de la resolución de la imagen (se debe obtener una imagen num_pix x num_pix)

    Devolución
    --------
        X -- array 2D (numero_imagenes x numero_pixeles) con los datos de cada una de las imágenes
        y -- array 1D (numero_imagenes) con el caracter que representa cada una de las imágenes
    """
    import warnings
    warnings.filterwarnings("ignore")
    #Iniciamos los arrays X e Y, el array X contendra 0s y
    # tendra como dimensiones el número de imágenes x el número de pixeles
    X = np.zeros((len(images), num_pix * num_pix))
    #El array Y tendrá una unica dimensión que será el numero de imágenes
```

```
    Y = np.empty(len(images), dtype= str)

    images_resize = [] #Lista donde guardaremos las imágenes con otro tamaño
    images_flat = []

    for i in range(len(images)): #Se recorren todas las imágenes y se las cambia de tamaño
        images_resize.append(resize(images[i], (num_pix, num_pix)))

    for i in range(len(images_resize)):
        images_flat.append(images_resize[i].reshape(images_resize[i].size).tolist())

    #Bloque try para posibles excepciones
    for i in range(len(images_resize)): #Se recorre images_resize
        try:
            Y[i] = caracteres[labels[i]]  # Se añade a Y el label correspondiente
        except ValueError:
            pass
        X[i] = images_flat[i]  # Se aplanan y se añaden a X

    return X, Y
```

En esta función, los métodos **reshape()**, **resize()**, y **tolist()** juegan un papel fundamental en el procesamiento y preparación de los datos de las imágenes para su uso en modelos de aprendizaje automático. A continuación, se describe brevemente su utilidad:

reshape()

Cambia la forma de una imagen redimensionada (matriz 2D) a una estructura 1D o "aplanada". Esto es necesario porque los algoritmos de aprendizaje automático suelen requerir que las imágenes sean representadas como vectores en lugar de matrices. **reshape()** facilita este cambio de forma sin alterar los datos originales.

resize()

El método **resize** de **skimage.transform** se usa para cambiar la resolución de cada imagen a un tamaño uniforme especificado por el parámetro **num_pix**. Esto asegura que todas las imágenes tengan dimensiones consistentes, lo cual es crucial para estructurar la matriz **X** con una forma definida (número de imágenes x número de píxeles). Este paso es especialmente útil para manejar datasets con imágenes de tamaños variados.

tolist()

El método **tolist** de **NumPy** convierte la estructura de datos de tipo array (que es propia de NumPy) a una lista de Python. En este caso, se utiliza para que las imágenes aplanadas puedan ser almacenadas como listas en **images_flat**, que posteriormente se integran en el array **X**.

Tarea 3

En esta tarea debemos implementar las funciones las funciones `entrena_perceptron`, `predice` y `evalua` y comprobar su funcionamiento con los datos de entrenamiento y test.

La implementación de dichas funciones es la siguiente:

entrena_perceptron():

```
def entrena_perceptron(X, y, z, eta, t, funcion_activacion):
    """ Entrena un perceptron simple
    Parámetros
    ----------
        X -- valores de x_i para cada uno de los datos de entrenamiento
        y -- valor de salida deseada para cada uno de los datos de entrenamiento
        z -- valor del umbral para la función de activación
        eta -- coeficiente de aprendizaje
        t -- numero de epochs o iteraciones que se quieren realizar con los datos de entrenamiento
        funcion_activacion -- función de activación para el perceptrón.

    Devolución
    --------
        w -- valores de los pesos del perceptron
        J -- error cuadrático obtenido de comparar la salida deseada con la que se obtiene
            con los pesos de cada iteración
    """

    import numpy as np

    # inicialización de los pesos
    w = np.zeros(len(X[0]))
    n = 0                           # numero de iteraciones se inicializa a 0

    # Inicialización de variables adicionales
    yhat_vec = np.zeros(len(y))     # array para las predicciones de cada ejemplo
    errors = np.zeros(len(y))       # array para los errores (valor real - predicción)
    J = []                          # error total del modelo

    while n < t:
        for i in range(0,len(X)):
            # Hayamos el producto escalar entre el vector peso y el de entrada
            # De forma que hayamos el sumatorio del producto entre los elementos
            escalar_prod = np.dot(w, X[i])
            # Aplicamos la función escalon al producto escalar
            resultados_prediccion = funcion_escalon(escalar_prod,z)
            if resultados_prediccion != y[i]:
                for weights in range(0,len(w)):
                    w[weights] = w[weights] + eta * (y[i] - resultados_prediccion) * X[i][weights]  # a

        n += 1  # se incrementa el número de iteraciones
        # calculo del error cuadrático del modelo
        # esto no es más que la suma del cuadrado de la resta entre el valor real
        # y la predicción que se tiene en esa iteración
        for i in range(0, len(y)):
            errors[i] = (y[i] - yhat_vec[i]) ** 2
        J.append(0.5 * np.sum(errors))

    return w, J
```

La función `entrena_perceptron` entrena un modelo de perceptrón simple ajustando sus pesos mediante un proceso iterativo. Inicializa los pesos en ceros y, durante cada epoch, calcula las predicciones usando el producto escalar entre los pesos y las entradas, aplicando una función de activación.

Si la predicción difiere de la salida esperada, actualiza los pesos según la regla de aprendizaje del perceptrón. Al final de cada epoch, calcula y registra el error cuadrático total, permitiendo evaluar el desempeño del modelo. La función devuelve los pesos ajustados y la evolución del error durante el entrenamiento.

predice():

```
def predice(w,X,z,funcion_escalon):
    import numpy as np
    """ Función de para la predicción
        Parámetros
        ----------
            w -- array con los pesos obtenidos en el entrenamiento del perceptrón
            x -- valores de x_i para cada uno de los datos de test
            z -- valor del umbral para la función de activación
            funcion_activacion -- función de activación para el perceptrón.

        Devolución
        --------
            y -- array con los valores predichos para los datos de test
        """
    resultado_array = np.zeros(len(X)) #Inciamos
    for x in range(0, len(X)):
        escalar_prod = np.dot(X[x], w)
        resultado_prediccion = funcion_escalon(escalar_prod, z)
        resultado_array[x] = resultado_prediccion
    return resultado_array
```

La función `predice` realiza predicciones utilizando un modelo de perceptrón previamente entrenado. Toma los pesos del modelo, las entradas de prueba, un umbral, y una función de activación. Para cada entrada, calcula el producto escalar entre los pesos y las características, aplicando luego la función de activación para obtener la predicción correspondiente. Estas predicciones se almacenan en un array y se devuelven como resultado.

evalua():

```
def evalua(y_test, y_pred):
    """ Función de activación
    Parámetros
    ----------
        y_test -- array con los valores salida conocidos para los datos de test
        y_test -- array con los valores salida estimados por el perceptrón para los datos de test

    Devolución
    --------
        acierto -- float con el valor del porcentaje de valores acertados con respecto al total de elem
    """
```

```
    acierto = float(0)
    for i in range(0,len(y_test)):
        acierto+=1 if y_test[i] == y_pred[i] else 0
    return acierto/len(y_test)
```

La función `evalua` calcula el porcentaje de aciertos de un modelo de perceptrón al comparar las predicciones realizadas con los valores reales. Toma como entrada los valores verdaderos (**y_test**) y las predicciones del modelo (**y_pred**). Compara cada par de valores y suma los aciertos, es decir, los casos en los que la predicción coincide con el valor real. Al final, devuelve el porcentaje de aciertos respecto al total de elementos en el conjunto de prueba.

Resultados

Los datos correspondientes a las imágenes cuyo label es la letra A o el número 3 son divididos en conjuntos de entrenamiento y de test que serán usados para comprobar la funcionalidad de estas funciones.

Se realizará una comparación entre las funciones de entrenamiento propias del perceptrón de sklearn como fit() y la creada por nosotros.

```
from Funciones.Tarea_3 import entrena_perceptron,evalua, predice, funcion_escalon #Importamos las funciones

z = 0.0     # umbral
eta = 0.1   # learning rate
t = 50      # número de iteraciones

#Obtenemos los pesos y el error
weights, errors = entrena_perceptron(X2C_train, y2C_train, z, eta, t,funcion_escalon)
#Luego de entrenarlo predecimos con los valores de test
y_pred_2C_own=predice(weights,X2C_test,z,funcion_escalon)
#Comparamos la predicción con los reales
porcentaje_acierto_perceptron_own=evalua(y2C_test, y_pred_2C_own)

print(porcentaje_acierto_perceptron_own)
[3]

 0.9930555555555556
```

Figure 1: Código de ejecución y métricas perceptrón propio

Perceptrón de Sklearn:

{} Code | M↓ Markdown

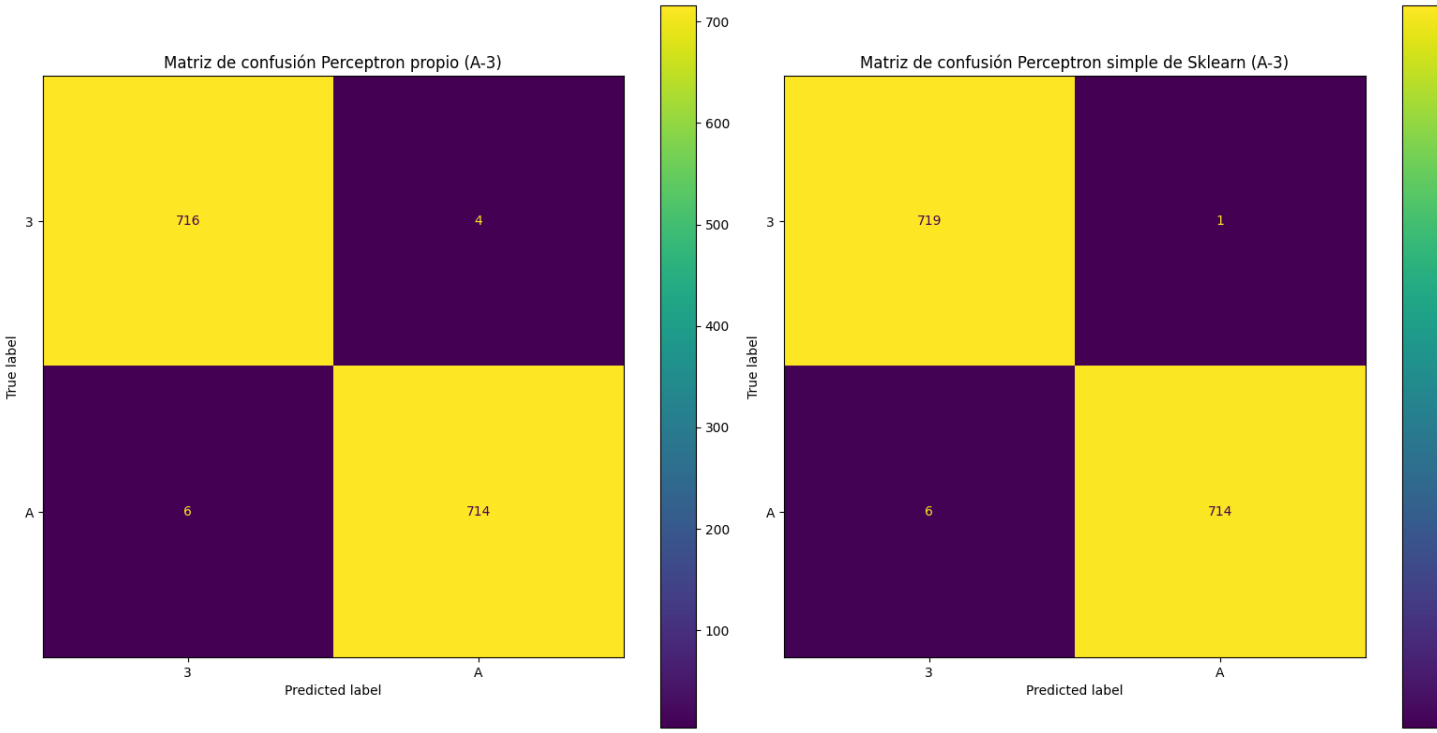
```
from sklearn.linear_model import Perceptron
from Funciones.Tarea_3 import evalua
#Inicializamos el perceptrón
clf = Perceptron(random_state=None, eta0= 0.1, shuffle=True, fit_intercept=True)
#Lo entrenamos con la función fit()
clf.fit(X2C_train, y2C_train)
#Sacamos las predicciones
y_pred_2C_sk = clf.predict(X2C_test)
#Comparamos la predicción con el real
porcentaje_acierto_perceptron_sk=evalua(y2C_test, y_pred_2C_sk)

print(porcentaje_acierto_perceptron_sk)
[4]
```

```
0.9951388888888889
```

Figure 2: Código de ejecución y métricas perceptrón sklearn

Se observa como la precisión es ligerisimamente menor en el caso del perceptrón propio, de la magnitud de las milésimas. Lo cual es lógico, dado que incluso para datasets sencillos las funciones propias del módulo sklearn son mucho mejores que las nuestras.

A continuación las matrices de confusión de ambas predicciones:



Tarea 4:

En la tarea 4 debemos observar en análisis de las métricas del perceptrón cuando se le entrena repetidamente con el mismo conjunto de datos. Las métricas que definen el comportamiento de un perceptrón y que estudiaremos tanto en esta tarea como en tareas posteriores son:

- **Exactitud (accuracy)**: Porcentaje de predicciones correctas. Se corresponde con el valor obtenido de nuestra función `evalua`.

- **Precisión (precision)**: Porcentaje de predicciones positivas correctas.

- **Sensibilidad (recall)** representa la tasa de verdaderos positivos. Es la proporción entre los casos positivos bien clasificados por el modelo, respecto al total de positivos.

Este análisis de las métricas se realizará en paralelo entre el perceptrón propio utilizado en la tarea 3 y el perceptrón de sklearn. Se comparará la evolución de las métricas en ambos perceptrones. Los conjuntos de datos utilizados serán los mismos que se usaron en la tarea 3.

Los resultados de los análisis son los siguientes:

```
----------Round 0------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9916666666666667
Precisión_sk: [0.99719101 0.98626374]
Sensibilidad_sk: [0.98611111 0.99722222]
----------Round 1------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9930555555555556
Precisión_sk: [0.99305556 0.99305556]
Sensibilidad_sk: [0.99305556 0.99305556]
----------Round 2------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9840277777777777
Precisión_sk: [1.         0.96904441]
Sensibilidad_sk: [0.96805556 1.        ]
----------Round 3------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.99375
Precisión_sk: [0.9972028  0.99034483]
Sensibilidad_sk: [0.99027778 0.99722222]
----------Round 4------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9923611111111111
Precisión_sk: [0.99167822 0.9930459 ]
Sensibilidad_sk: [0.99305556 0.99166667]
----------Round 5------------------
```

```
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9951388888888889
Precisión_sk: [1.         0.99037139]
Sensibilidad_sk: [0.99027778 1.        ]
----------Round 6------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9944444444444445
Precisión_sk: [0.99859944 0.99035813]
Sensibilidad_sk: [0.99027778 0.99861111]
----------Round 7------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.99375
Precisión_sk: [0.99306519 0.99443672]
Sensibilidad_sk: [0.99444444 0.99305556]
----------Round 8------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.99375
Precisión_sk: [0.99859748 0.98899587]
Sensibilidad_sk: [0.98888889 0.99861111]
----------Round 9------------------
Exactitud_own: 0.9923611111111111
Precisión_own: [0.9944212  0.99031812]
Sensibilidad_own: [0.99027778 0.99444444]
Exactitud_sk: 0.9944444444444445
Precisión_sk: [0.99307479 0.99582173]
Sensibilidad_sk: [0.99583333 0.99305556]
```

Los resultados muestran como la exactitud del perpeptrón propio no varia entre iteraciones, esto puede ser debido a la simplicidad del método `entrena_perceptron()` que cuando se llama repetidas veces con el mismo dataset siempre da los mismos resultados. Se observa un contraste con el perceptrón de sklearn cuta exactitud sí que aumenta ligeramente entre iteraciones.

Desde el punto de vista de de la precisión y de la sensibilidad se observa como el perceptrón propio sigue sin cambiar de métricas, debido a que el resultado de los entrenamientos es el mismo, como hemos visto en el caso de la exactitud. La sensibilidad y precisión del perpeceptrón de sklearn van fluctuando desde el 1 hasta el 0.990.

Tarea 5

En la Tarea 5 se deben analizar las métricas y matriz de confusión de todas las combinaciones posibles entre caracteres. Esta tarea se realizará sobre el perceptron de sklearn. Para esa tarea se ha implementado la función analisis_binario() que se encuentra en el archivo Tarea_5.py.

Las métricas que vamos a observar son las ya mencionadas en la Tarea_4:

- **Exactitud (accuracy)**: Porcentaje de predicciones correctas. Se corresponde con el valor obtenido de nuestra función `evalua`.

- **Precisión (precision)**: Porcentaje de predicciones positivas correctas.

- **Sensibilidad (recall)** representa la tasa de verdaderos positivos. Es la proporción entre los casos positivos bien clasificados por el modelo, respecto al total de positivos.

Además se mostrará la matriz de confusión de cada par de caracteres.

analisis_binario():

A continuación la explicación de la función `analisis_binario()`:

```
def analisis_binario(Ximage,yimage, caracteres):
    """
    La función análisis binario realizará un análisis en las métricas
    y en los resultados de entrenar al perceptron simple de sklearn
    con cada combinación binaria posible de los caracteres en el diccionario
    caracteres.

    Luego se visualizará la matriz de confusión de aquellos pares de caracteres
    con los que el perceptrón tenga un peor desempeño.

    :param Ximage: dataset con las imágenes y sus pixeles.
    :param yimage: dataset con los labels de cada imágen
    :param caracteres: diccionario con todos los caracteres que aparecen en las imágenes
    """
    from sklearn.linear_model import Perceptron

    from sklearn import metrics

    from sklearn.model_selection import train_test_split

    import numpy as np

    from matplotlib import pyplot as plt

    #Para evitar la repetición al comparar caracteres que nos den los mismos resultados
    # (e.j comparación de {O,o} y de {o,O}
    #Se comprobará primero si aquel par de caracteres ya ha sido comparado

    lista_caracteres_ya_pareados = [] #Lista donde se guardaran los caracteres pareados
    ya_comparados = False

    for caracter_1 in caracteres.values():
        for caracter_2 in caracteres.values(): #Cada caracter itera sobre todos los demás

            set_caracteres = {caracter_1, caracter_2} #Set con los caracteres a comparar
            if caracter_1 == caracter_2: #Si son iguales también se salta la iteración
                continue
            for par in lista_caracteres_ya_pareados: #Se itera sobre la lista de ya_pareados
                if set_caracteres == par: #Si los sets son iguales
                    ya_comparados = True
                    break #Se rompe el primer bucle for
```

```
            if ya_comparados: #Si ya han sido comparados se sale del segundo bucle
                ya_comparados = False
                continue
            lista_caracteres_ya_pareados.append(set_caracteres) #Se añade el par a la lista

            #Se comienza con la separacion del dataset en función de los caracteres
            X_binario = Ximage[((yimage == caracter_1)| (yimage == caracter_2))] #X con solo los caract
            y_binario = yimage[((yimage == caracter_1) | (yimage == caracter_2))] #Y con solo los carac
            y_binario_num = (y_binario == caracter_1).astype(int)  #Se pasa Y a binario 0,1

            #Se divide X_binario e y_binario_num en los sets de entrenamiento y test
            X_train, X_test, y_train, y_test = train_test_split(X_binario, y_binario_num, stratify=y_bi

            clf = Perceptron(random_state=None, eta0= 0.1, shuffle=True, fit_intercept=True) #Creamos e
            clf.fit(X_train, y_train) #Se entrena el perceptron
            y_pred_binario = clf.predict(X_test) #Se guarda la predicción

            accuracy= round(metrics.accuracy_score(y_test,y_pred_binario),3) #Se obtiene la accuracy
            sensibilidad = round(metrics.recall_score(y_test,y_pred_binario),3) #Se obtiene la sensibil
            precision = round(metrics.precision_score(y_test,y_pred_binario),3) #Se obtiene la precisio

            print("Exactitud:" + str(accuracy)+ "de los valores("+ str(caracter_1)+"," + str(caracter_2
            print("Sensibilidad:" + str(sensibilidad) + "de los valores(" + str(caracter_1) + "," + str
            print("Precision:" + str(precision) + "de los valores(" + str(caracter_1) + "," + str(carac

            #Visualizamos la matriz de confusión
            labels=np.unique(y_test)
            conf_mat= metrics.confusion_matrix(y_test,y_pred_binario,labels=labels)
            cm_display=metrics.ConfusionMatrixDisplay(confusion_matrix = conf_mat,display_labels= [str(

            fig,ax =plt.subplots(figsize=(10,10))
            cm_display.plot(ax=ax)

            plt.title("Matriz de confusión Perceptron simple de Sklearn(" + str
            (caracter_1)+"," + str(caracter_2)+ ")")
            plt.xticks(rotation='vertical')
            plt.show()
```

La función `analisis_binario` realiza un análisis del desempeño de un perceptrón simple de **sklearn** entrenado para distinguir pares de caracteres en un dataset. Comienza generando todas las combinaciones binarias posibles de caracteres en el diccionario proporcionado, asegurándose de no repetir pares previamente evaluados. Para cada par de caracteres, filtra el conjunto de datos para incluir únicamente las muestras asociadas a esos caracteres, transformando las etiquetas en valores binarios (0 o 1).

A continuación, divide el subconjunto de datos filtrado en conjuntos de entrenamiento y prueba utilizando una proporción especificada. Luego, entrena un modelo de perceptrón simple sobre los datos de entrenamiento y genera predicciones para el conjunto de prueba. Se calculan métricas clave como precisión, sensibilidad y exactitud, que se imprimen para cada par de caracteres evaluados.

Además, la función genera y visualiza la matriz de confusión, a la hora de interpretar los resultados se mostrarán aquellos pares que tengan un peor desempeño

Resultados

Aquellos pares que han tenido pero desempeño son:

(0,o)

Exactitud:0.58

Sensibilidad: 0.24

Precisión: 0.737

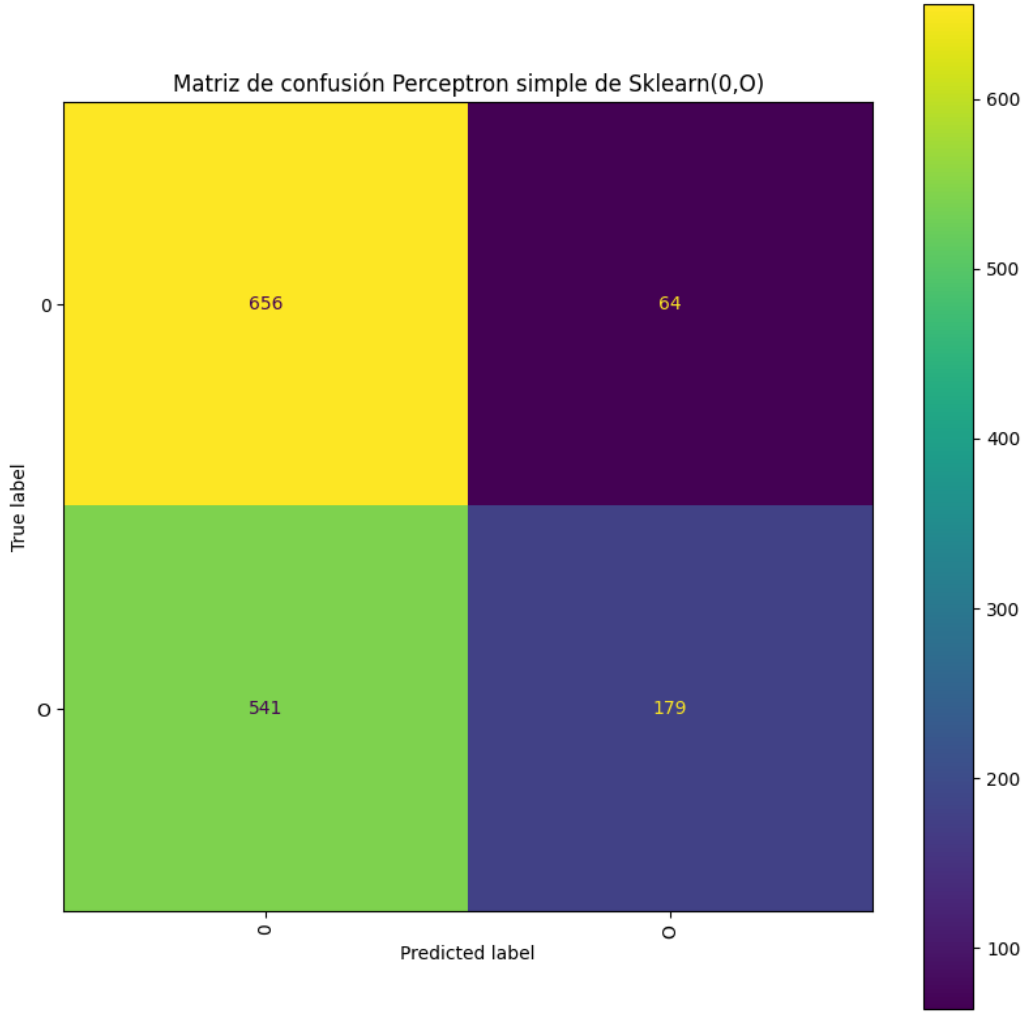


Figure 3: MC (0,o)

(1,I)

Exactitud: 0.596

Sensibilidad: 0.939

Precisión: 0.557

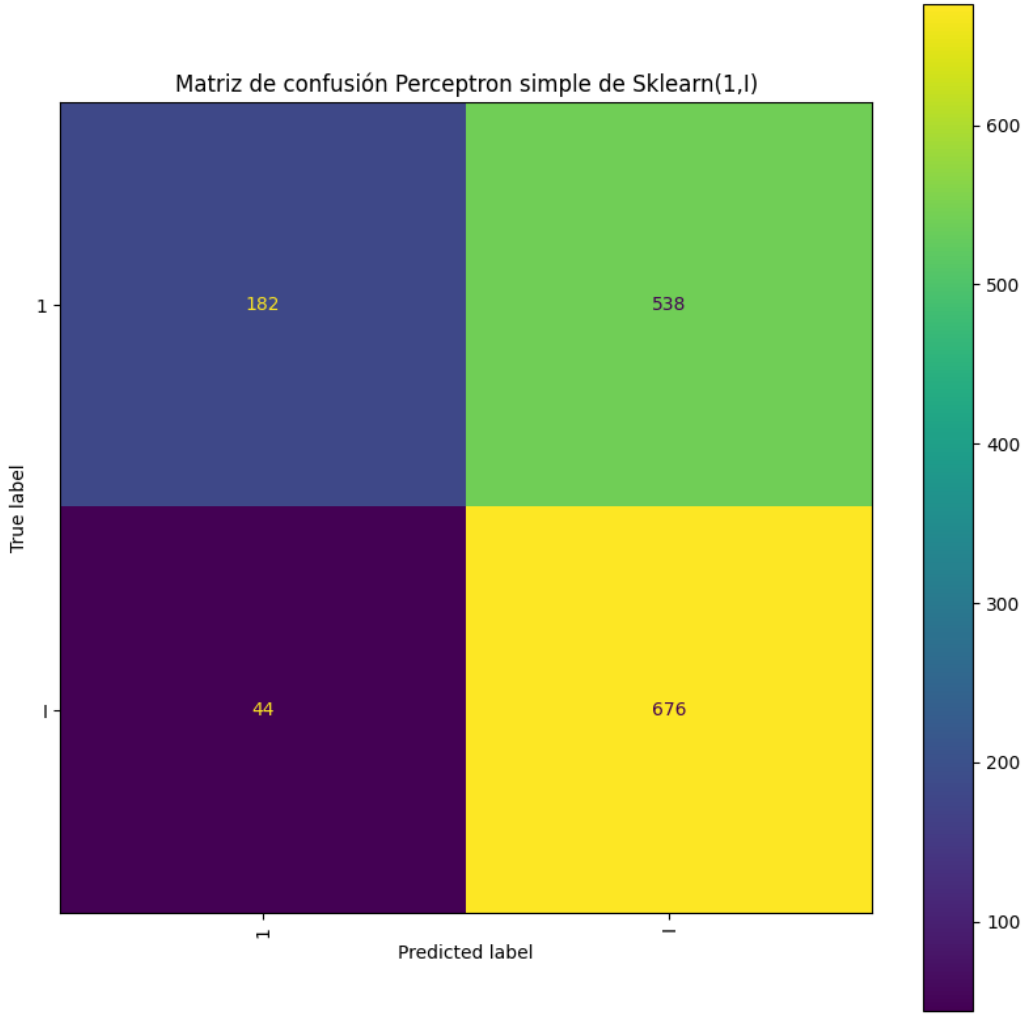


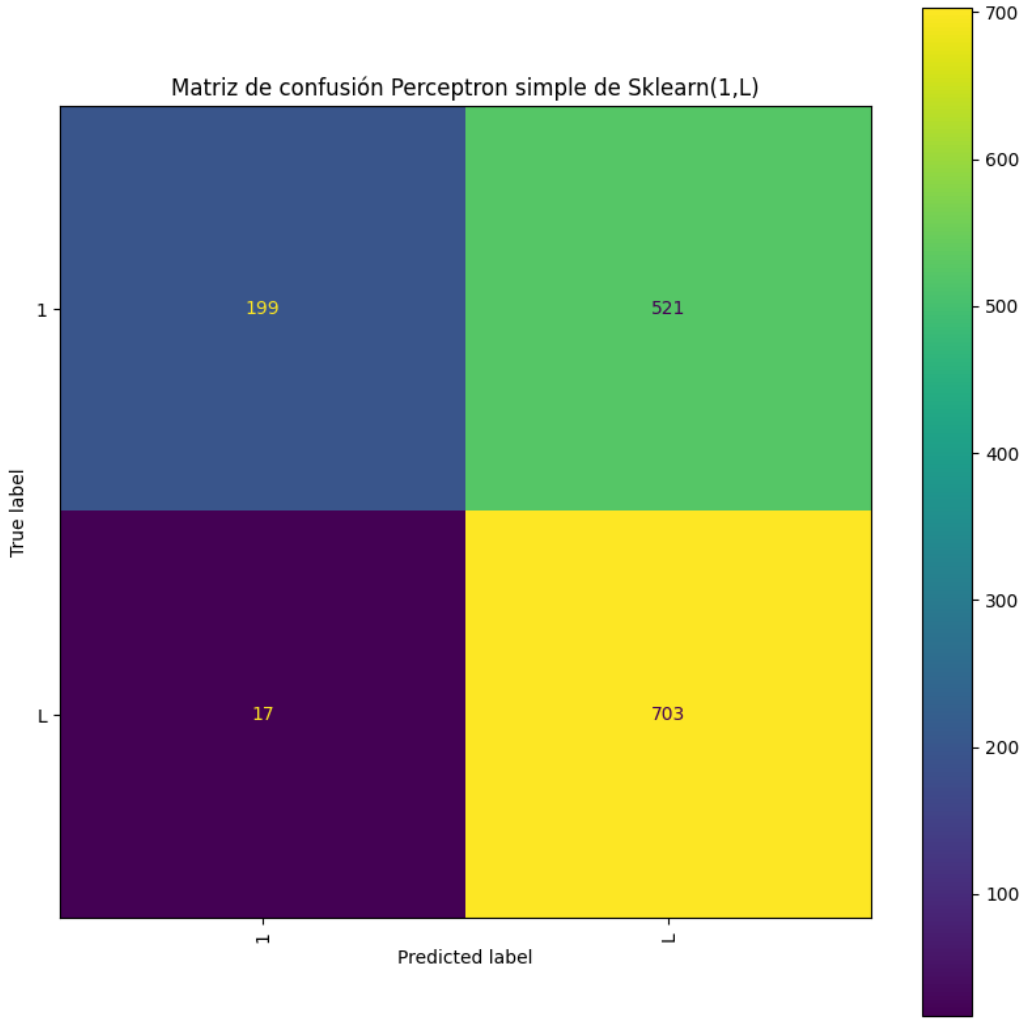
Figure 4: MC (1,I)

(1,L)

Exactitud: 0.626

Sensibilidad: 0.976

Precisión: 0.574



(9,q)

Exactitud: 0.756

Sensibilidad: 0.832

Precisión: 0.722

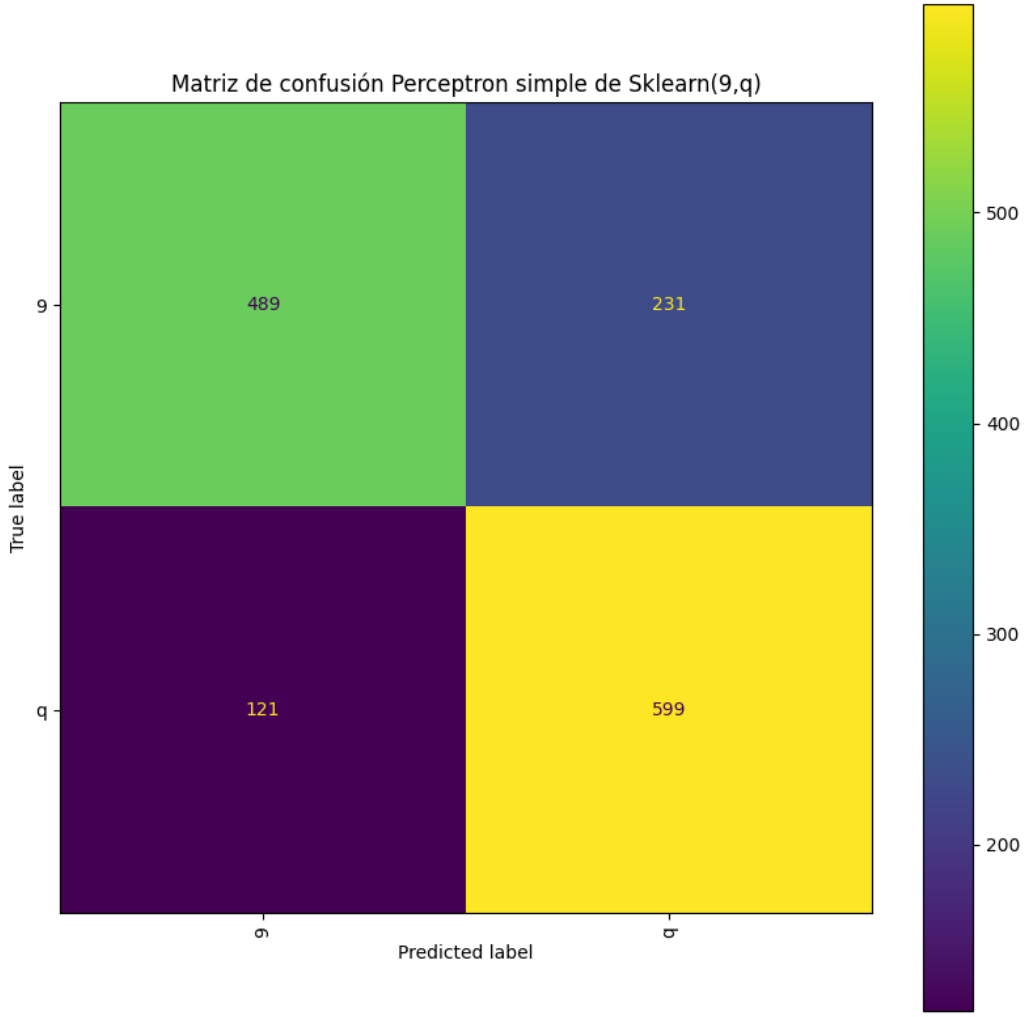


Figure 5: MC (9,q)

(F,f)

Exactitud: 0.549

Sensibilidad: 0.761

Precisión: 0.534

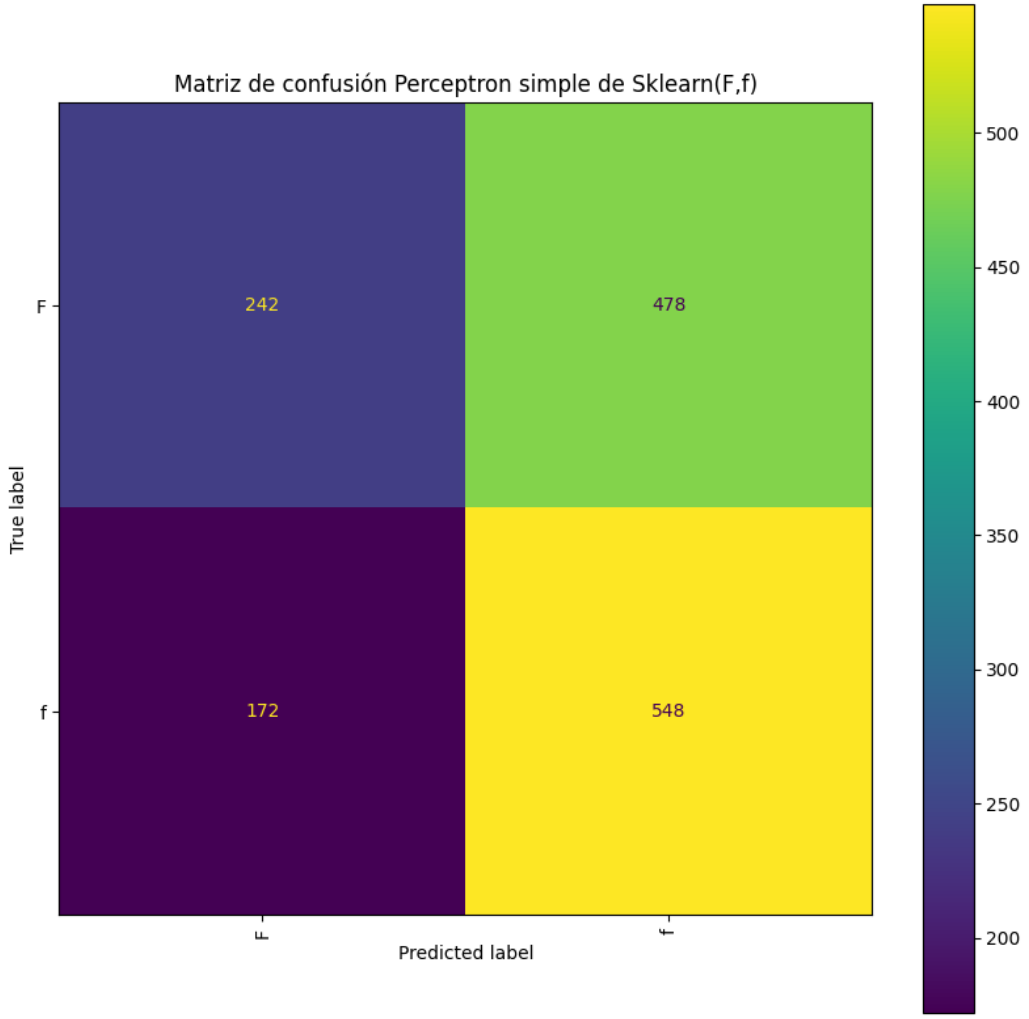


Figure 6: MC (F,f)

(I,L)

Exactitud: 0.624

Sensibilidad: 0.479

Precisión: 0.674

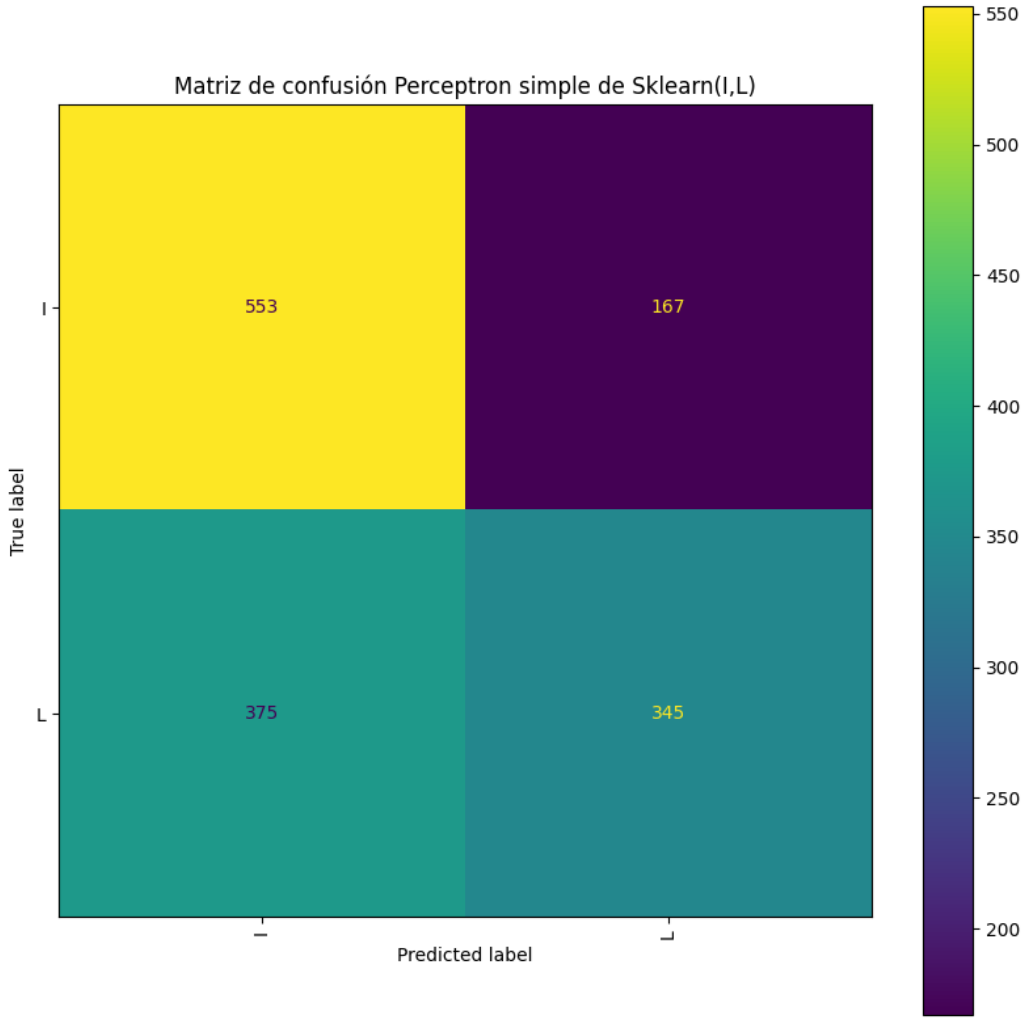


Figure 7: MC (I,L)

(g,q)

Exactitud: 0.671

Sensibilidad: 0.494

Precisión: 0.764

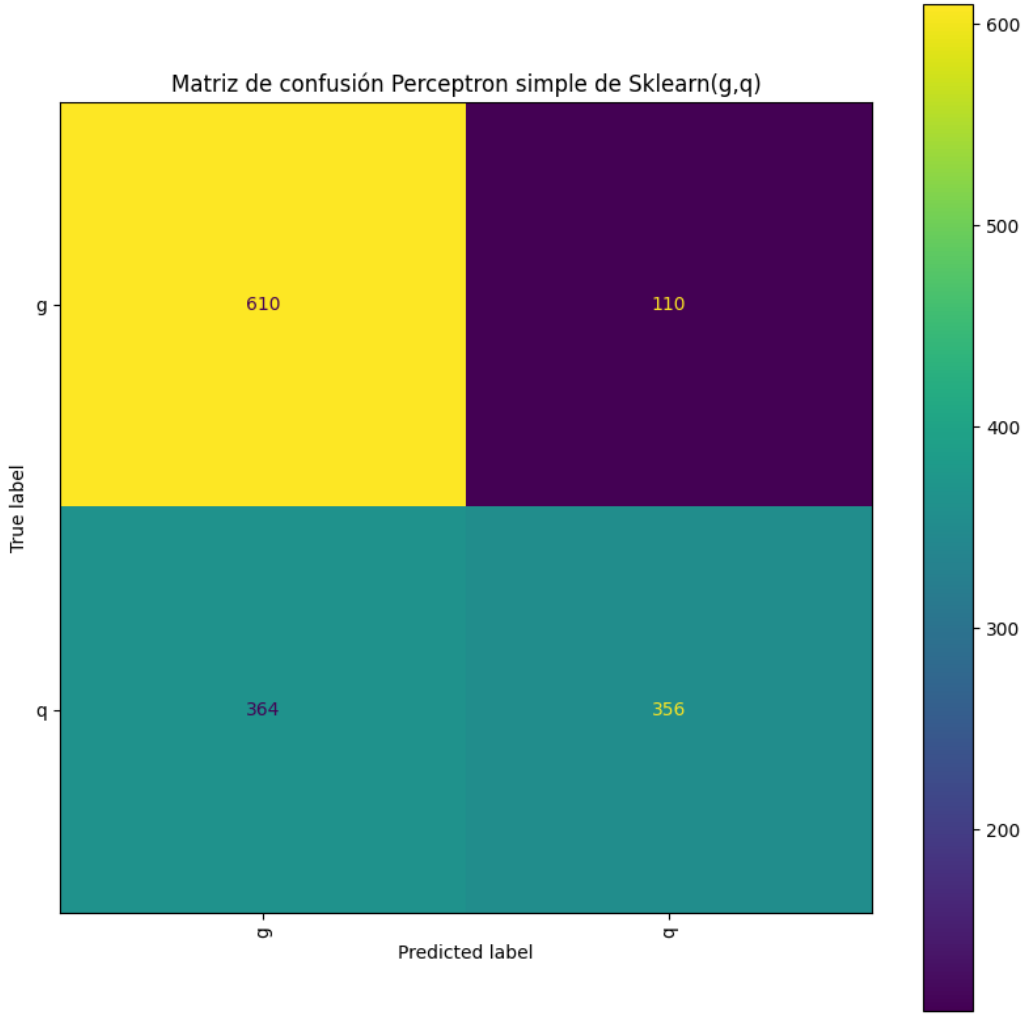
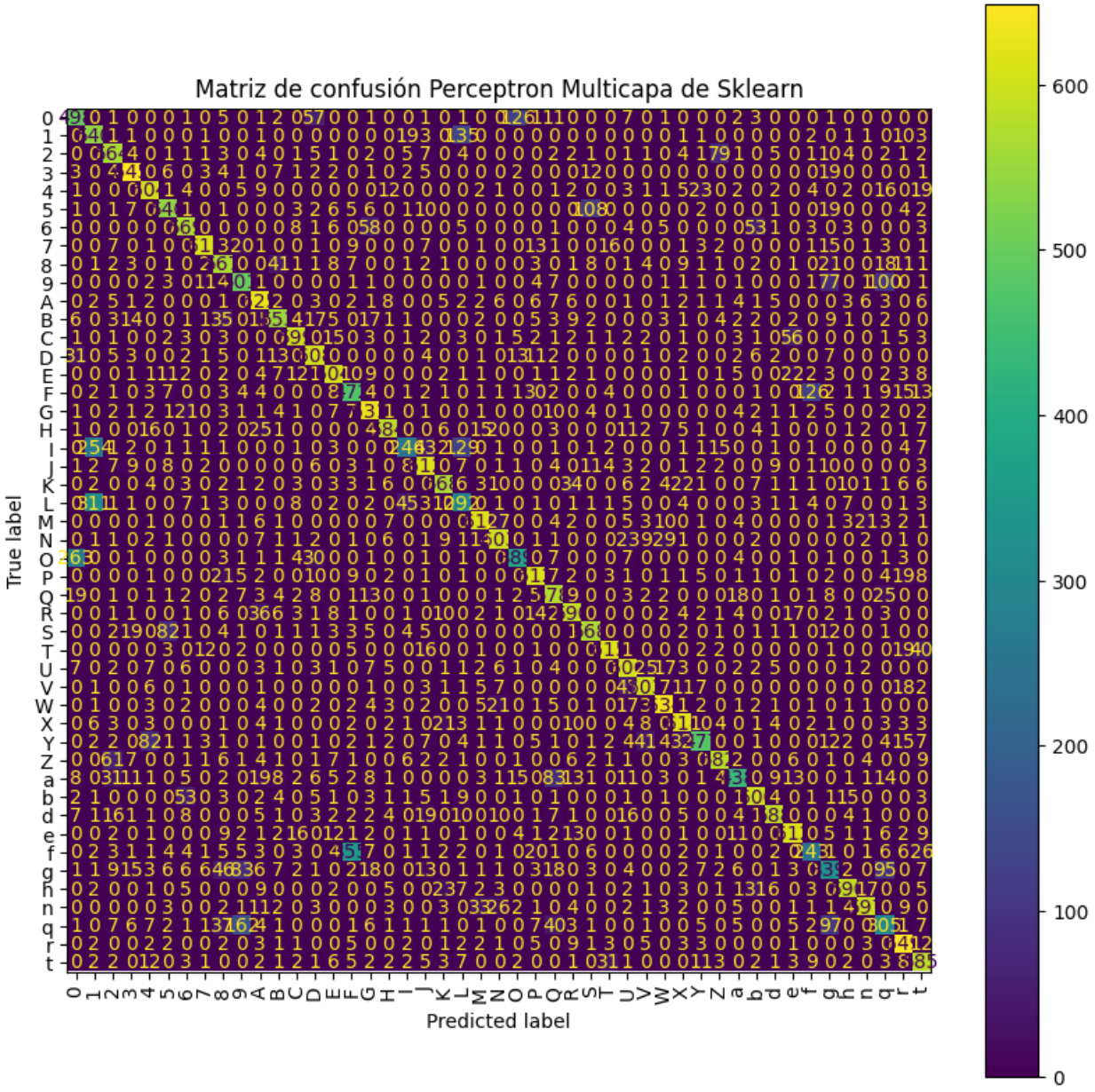


Figure 8: MC (g,q)

Tarea 6

En la tarea 6 debemos utilizar el perceptrón multicapa (MLP) de sklearn y comparar las métricas halladas de entrenar este clasificador y hacer la predicción con los datos usados anteriormente en la práctica con aquellas métricas halladas de entrenar el MLP con *todos* los datos provenientes de Ximage e yimage y hacer el test con unos nuevos datos.

La matriz de confusión del perceptrón multicapa de sklearn entrenado con el split de datos X_Train proveniente de Ximage cuya predicción se ha hecho con el split de datos X_Test también proveniente de Ximage es la siguiente:



Las métricas obtenidas son las siguientes:

```
Exactitud: 0.759

Precisión:
[0.585, 0.475, 0.753, 0.855, 0.784, 0.786, 0.794, 0.926, 0.711, 0.623,
0.752, 0.820, 0.882, 0.777, 0.838, 0.514, 0.768, 0.890, 0.703, 0.779,
0.844, 0.458, 0.857, 0.801, 0.688, 0.812, 0.707, 0.816, 0.773, 0.900,
0.758, 0.841, 0.861, 0.817, 0.835, 0.824, 0.822, 0.823, 0.900, 0.818,
0.590, 0.496, 0.894, 0.899, 0.483, 0.795, 0.718]

Sensibilidad:
[0.692, 0.750, 0.783, 0.896, 0.839, 0.750, 0.790, 0.851, 0.788, 0.696,
0.868, 0.768, 0.832, 0.844, 0.839, 0.656, 0.876, 0.811, 0.342, 0.850,
0.789, 0.406, 0.856, 0.838, 0.540, 0.850, 0.803, 0.819, 0.789, 0.854,
0.842, 0.838, 0.881, 0.858, 0.663, 0.808, 0.608, 0.833, 0.817, 0.850,
0.338, 0.471, 0.819, 0.829, 0.424, 0.901, 0.813]
```

Vemos que la exactitud de esta predicción es notablemente más baja que en la mayoría de los casos binarios que hemos visto en la tarea 5, esto se debe a que el perceptrón predice con exactitud ciertos datos y otros lo hace de peor forma, de manera que la exactitud global es de un 75, 9%.

Rango precisión: 0.459 -0.926

Rango Sensibilidad: 0.338 -0.901

La matriz de confusión del perceptrón multicapa de sklearn entrenado con TODOS los datos provenientes de Ximage y cuya predicción se ha hecho con los nuevos datos proporcionados ha sido:

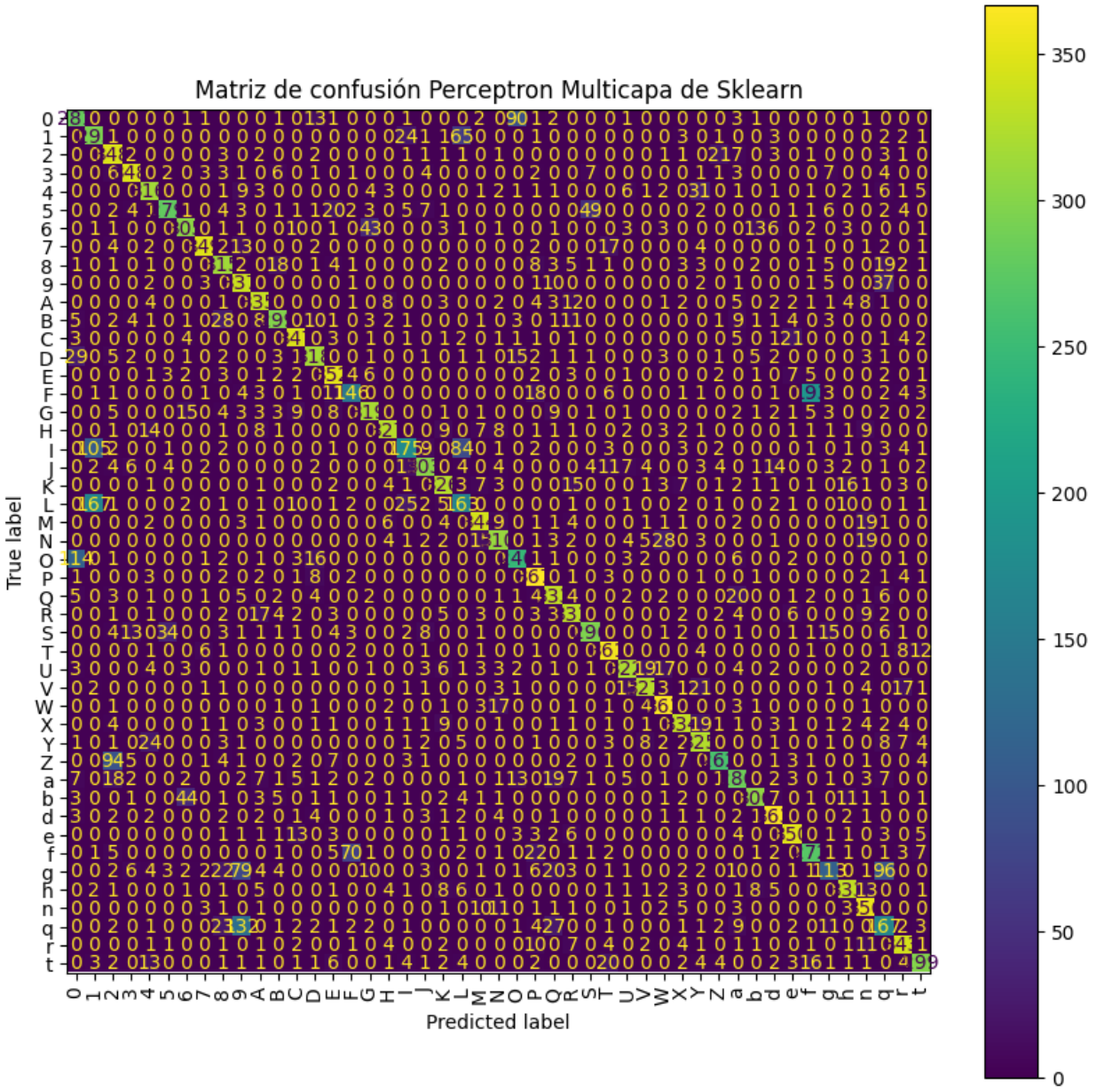


Figure 9: Matriz de confusión de MLP

Las métricas obtenidas son las siguientes

```
Exactitud: 0.760

Precisión:
[0.616, 0.509, 0.662, 0.885, 0.790, 0.853, 0.794, 0.930, 0.711, 0.562, 0.805, 0.842,0.836, 0.815, 0.801
0.650, 0.775, 0.748, 0.786, 0.818, 0.826, 0.856, 0.870, 0.825, 0.850, 0.756, 0.853,
0.724, 0.887, 0.848, 0.852, 0.532, 0.617, 0.845, 0.753, 0.428, 0.817, 0.831]
```

```
Sensibilidad:
[0.703, 0.738, 0.870, 0.870, 0.790, 0.698, 0.763, 0.873, 0.788, 0.843, 0.838, 0.748,
0.868, 0.795, 0.880, 0.365, 0.798, 0.818, 0.438, 0.758, 0.815, 0.408, 0.860, 0.775,
0.615, 0.918, 0.833, 0.828, 0.740, 0.903, 0.805, 0.818, 0.908, 0.835, 0.813, 0.655,
0.715, 0.765, 0.908, 0.875, 0.680, 0.283, 0.833, 0.890, 0.418, 0.858, 0.748]
```

En comparación con las métricas obtenidas en el anterior entrenamiento del MLP no se observa una diferencia muy notable, la exactitud solo aumenta en un 0,1 % mientras que los rangos de precisión se quedan iguales y los de sensibilidad aumentan.

Rango precisión: 0.430 - 0.930

Rango Sensibilidad: 0.283 - 0.918

Tarea 7

En la tarea 7 debemos realizar una validación cruzada con el perceptrón simple y con el MLP, y comparar sus resultados, para ello utilizaremos el conjunto de datos que solo tiene las imágenes correspondientes a la letra A y al número 3.

Antes, explicaremos el concepto de validación cruzada:

La **validación cruzada** es una técnica estadística utilizada para evaluar la capacidad de generalización de un modelo, como una red neuronal, y evitar problemas como el **sobreajuste** (cuando el modelo aprende demasiado bien los datos de entrenamiento, pero falla con datos nuevos). Su objetivo principal es medir cómo se comportará el modelo con datos no vistos, proporcionando una estimación más confiable de su desempeño.

La validación cruzada divide el conjunto de datos disponible en varios subconjuntos o **"folds"**. Uno de estos subconjuntos se utiliza como datos de validación, mientras que los demás se utilizan para entrenar el modelo. Este proceso se repite varias veces, rotando el subconjunto utilizado para validación, de modo que cada fold se emplee como conjunto de validación al menos una vez. Al final, se promedian los resultados obtenidos en cada iteración para obtener una estimación robusta del desempeño del modelo.

Por ejemplo, en una validación cruzada de 3 pliegues (**3-fold cross-validation**):

1. Los datos se dividen en 3 partes iguales.
2. En la primera iteración, se utiliza la primera parte como conjunto de validación, y las otras 2 para entrenar el modelo.
3. En la segunda iteración, se utiliza la segunda parte para validar, y así sucesivamente hasta cubrir todos los pliegues.

Usaremos este tipo de validación cruzada para el perceptrón simple y el MLP, aquí la matriz de confusión que se obtiene:

Perceptrón simple:

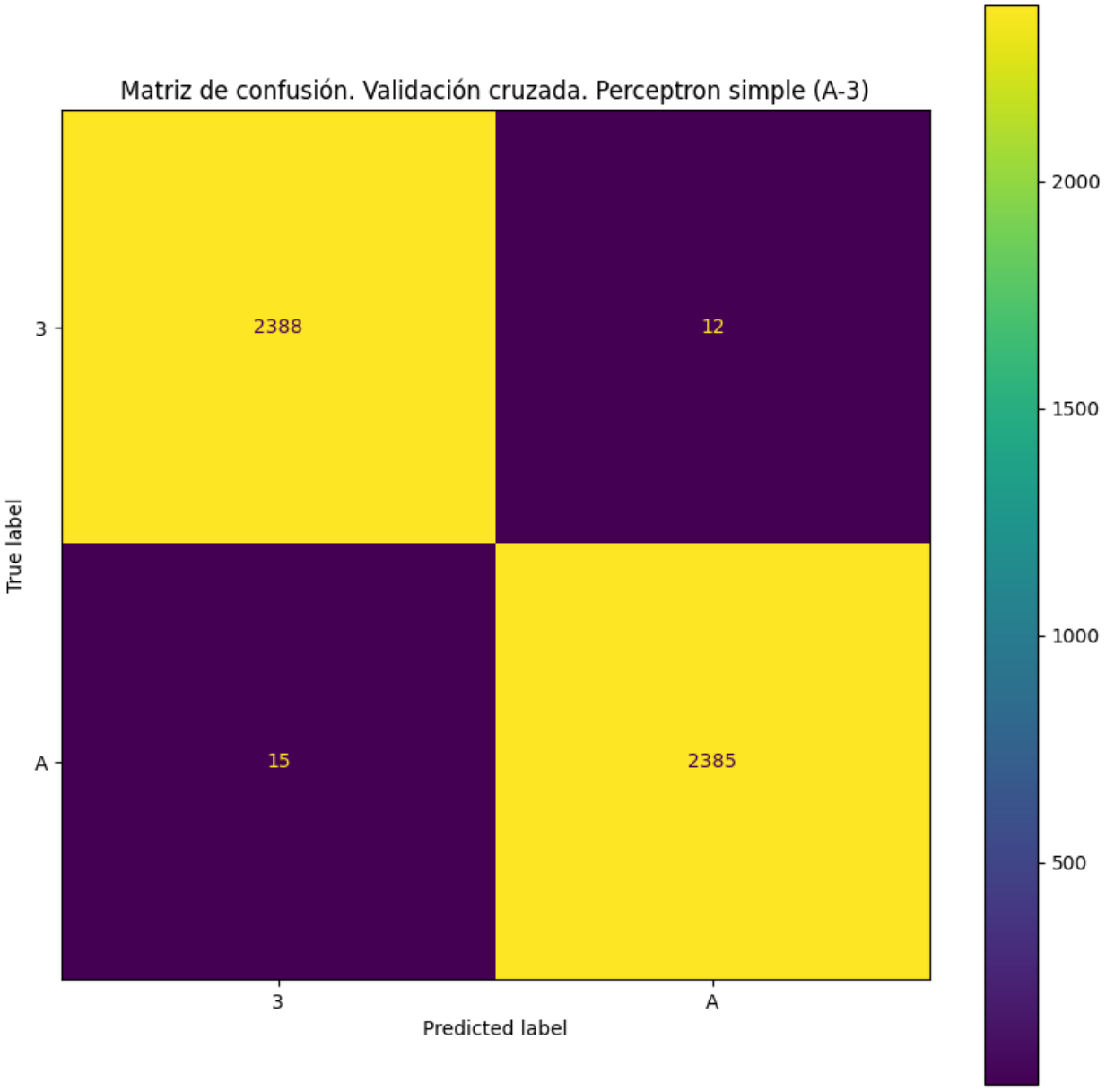


Figure 10: Matriz de confusión de cross-validation PS

Perceptrón multicapa:

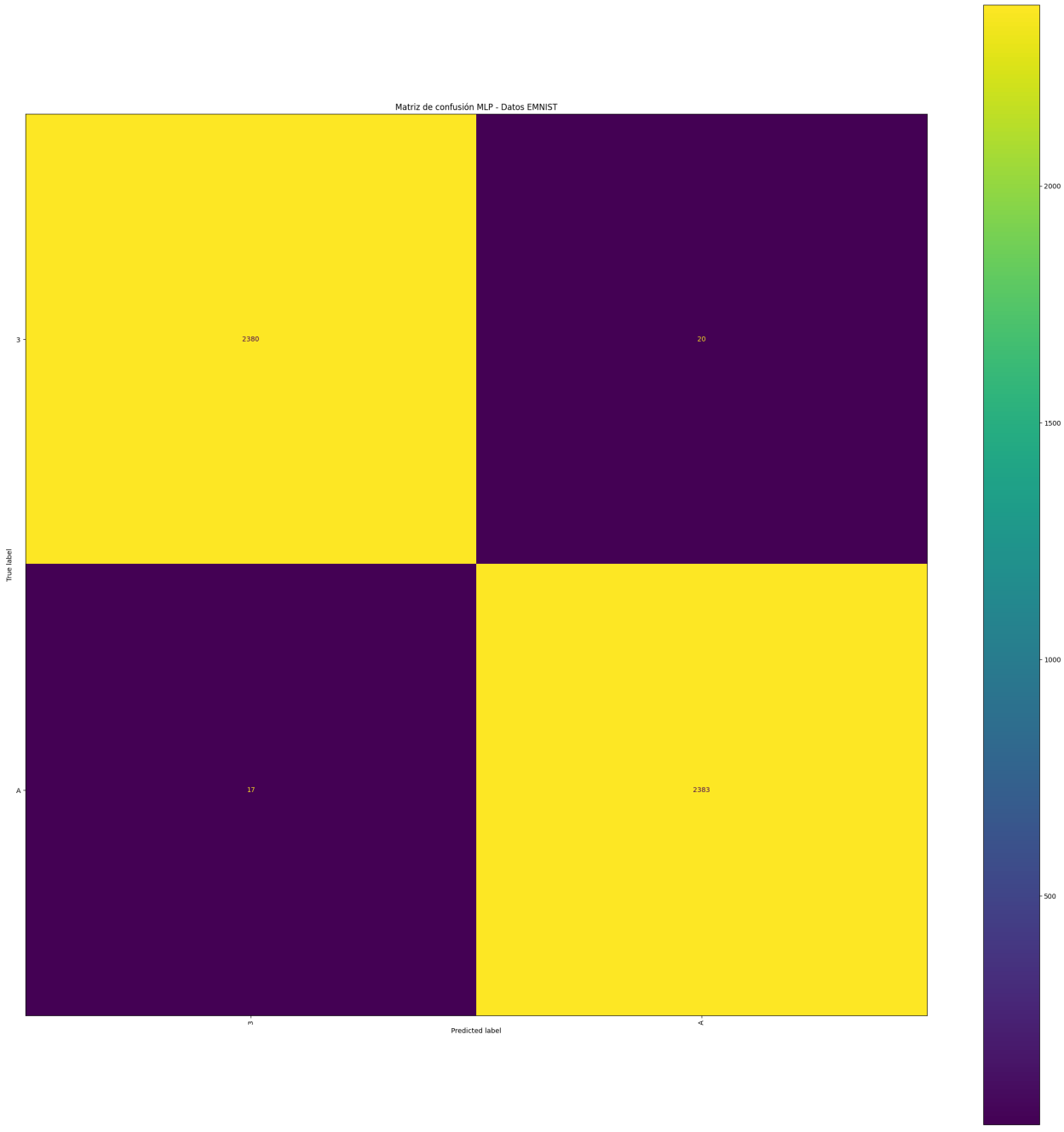


Figure 11: Matriz de confusión de cross-validation MLP

Se observa como los resultados son similares entre estas matrices de confusión, esto es debido a la alta exactitud que tienen estos dos modelos dado el dataset (A-3), sin embargo, el número de predicciones hechas es mucho mayor, llegando a los 2300 TP, en comparación de los 714 TP en la predicción normal.

Dicho esto, la validación cruzada dará más fiabilidad en los resultados pero su uso en un dataset grande puede consumir mucho tiempo en completar.

(Con todo el dataset tardo 30 minutos en mi portartil)