DigitReco; reconocimiento de números manuscritos

Copyright © 2024 por Lorena Sánchez. Todos los derechos reservados.

Tabla de contenido

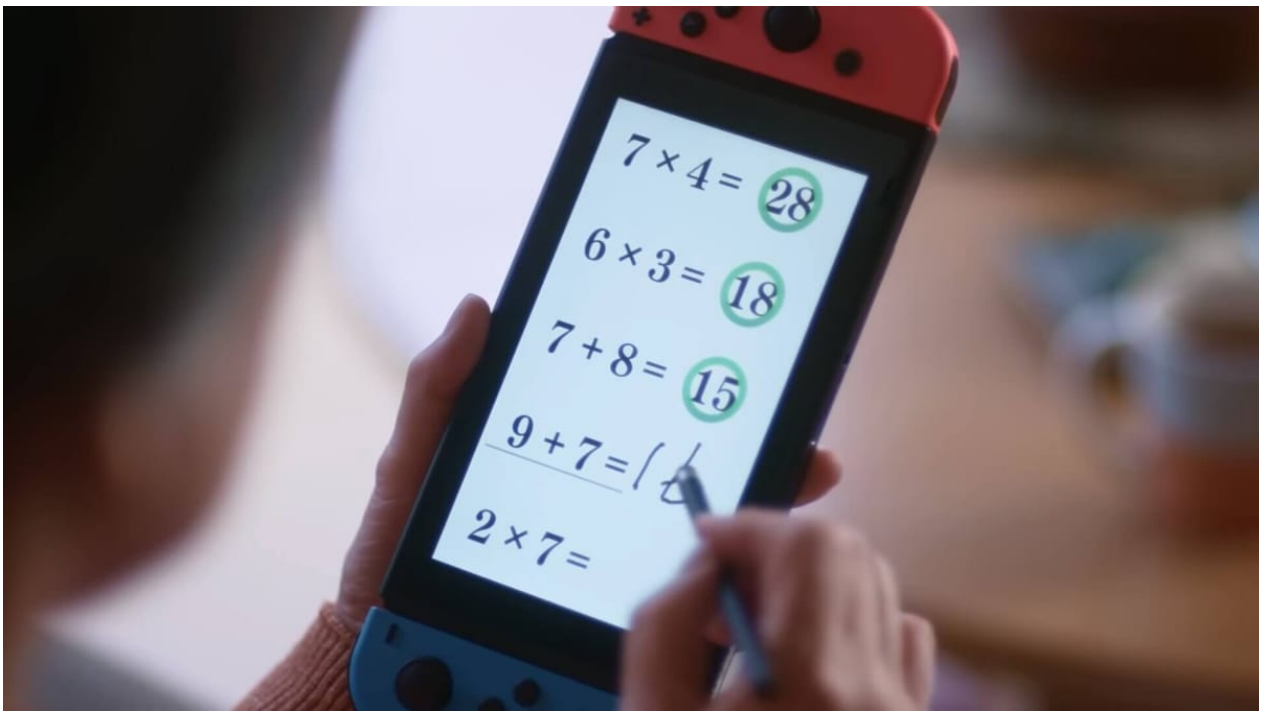
Introducción 3
Primeras nociones 4
Información Técnica 5
Generación del modelo 5
Cargado del modelo 9
Implementación con .NET 10
De líneas a bytes 13
Interfaz gráfica 17
Manual de usuario 19
Requisitos previos 19
Interfaz 19
Solución de errores 22
Más ayuda 25

Introducción

DigitReco es un programa encargado de reconocer dígitos escritos por el usuario y realizar una predicción en base a los píxeles que conforman el dígito.

Origen de la idea

Esta idea de proyecto surgió como un *sideproject* que quería realizar a mitad de curso.
No suelo dedicarle mucho tiempo a los videojuegos, por lo que cuando lo hago busco una experiencia que no requiera más trasfondo que saber a lo que estás jugando en el momento. No me gusta la necesidad de tener que *pillar* la mecánica conforme inviertes horas o el tener que recordar la historia de un videojuego que probablemente lleve meses sin jugar.
Es por este motivo que disfruto mucho las demos; tienen una duración perfecta y te aportan todo lo necesario para disfrutar el videojuego en el momento.
Una de las demos que probé fue *Dr Kawashima's Brain Training;* qué nostalgia.
El *Brain Training* es uno de esos juegos que aunque no se le echaba muchas horas todos lo teníamos en la DS hace años. ¡Pues parece ser que había una demo para la Switch!
Me decidí a probarla y me pareció maravillosa; consiguen mantener un formato que cualquiera puede entender, con mucha simpleza pero a la vez ofreciendo una experiencia de usuario buenísima.
La idea de aplicación no parece muy compleja; distintas actividades para entrenar tu lógica y memorización y un *trackeo* diario de ella para ver cómo vas mejorando. Siendo una idea como esta seguro que tenían que haber muchas alternativas para Android, ¿verdad?
A la conclusión que llegué tras probar un par de ellas es que cumplen su función pero no logran esa inmersión que Nintendo sí; en Android al fin y al cabo obtenías los resultados de cada actividad pero no era la misma sensación simplemente por cómo se presentaba al usuario.
Nintendo a pesar de mantenerse simple en interfaz proporcionaba una inmersión muy buena.



Específicamente me gustó el de cálculo de números, por lo que pensé en centrarme en este aspecto más que en una aplicación completa con distintos ejercicios de lógica.
Quería desarrollar yo misma una aplicación con esa UX que buscaba en otras, pero, ¿cómo puede mi programa entender que las 2 rayas que ha dibujado el usuario son el número 9?

Creado con el Personal Edition de HelpNDoc: Agilice su proceso de documentación con la plantilla HTML5 de HelpNDoc

Primeras nociones

Para entrenar el modelo de Inteligencia Artificial que utilizaremos en DigitReco necesitamos unas cuantas imágenes, pero no 4 ni 400, sino miles.
Para ello haremos uso de un dataset.
Un dataset es un conjunto de datos utilizado en el campo de la inteligencia artificial para entrenar y probar algoritmos de aprendizaje.
Gracias a este dataset podremos crear un algoritmo capaz de aprender patrones y hacer predicciones en base a ello.
Kaggle es una plataforma bien conocida en la comunidad del *Machine Learning* en la cual encontraremos herramientas útiles.
De esta misma página fue de donde obtuve el dataset de MNIST en formato CSV. Este CSV está formado por 60.000 líneas, y cada línea tiene 785 números.

¿Por qué 785?

A un modelo de aprendizaje no podemos enseñarle miles de imágenes y esperar que aprenda de ellas.
Nuestro modelo va a necesitar información en bits, por lo que si tomamos como referencia imágenes de 28

píxeles obtendremos 784 valores que corresponden a cada pixel. (28 x 28 = 784).
Cada uno de los 784 números por línea será un número del 0 al 255 haciendo referencia a una escala de grises; siendo 0 negro y 255 blanco.
Mediante estos números estamos haciendo saber al modelo en qué pixeles se encuentran nuestros trazos.
El número extra, el 785, es simplemente el propio número (del 0 al 9) que representa la imagen. De esta manera el modelo tras hacer su predicción verá si el valor real coincide con lo sacado. En base a esto cambiará el algoritmo para aprender de una manera que ofrezca mayor precisión.

Una alternativa a descargar el dataset en un *CSV* es importarlo desde *Tensorflow*.

Creado con el Personal Edition de HelpNDoc: Transforma tu documento de Word en un libro electrónico de calidad profesional con HelpNDoc

Información Técnica

Creado con el Personal Edition de HelpNDoc: Cree sin esfuerzo documentación profesional con la interfaz de usuario limpia de HelpNDoc

Generación del modelo

El modelo a entrenar está programado en Python.
Como alternativa a cargar el CSV local podemos utilizar la librería de datasets de Tensorflow para cargar el MNIST únicamente con 1 línea de código:

```
import tensorflow_datasets as tfds

dataset, metadata = tfds.load('mnist', as_supervised=True, with_info=True)
```

La función *load()* de Tensorflow devuelve 2 valores; el propio dataset y metadata (lo cual nos puede ser útil para obtener información extra del conjunto de datos).

Una vez almacenado el dataset en la variable *dataset*, vamos a dividir el set en 2.
Una parte será el conjunto de entreno y otro el de pruebas. Esto se hace porque si utilizásemos los mismos datos de entreno para hacer las pruebas los resultados del modelo final podrían ser más imprecisos.

```
train_dataset, test_dataset = dataset['train'], dataset['test']
```

Podemos obtener el tamaño del ambos conjuntos con los metadatos. Esta información la utilizaremos más adelante.

```
num_train_examples = metadata.splits['train'].num_examples
num_test_examples = metadata.splits['test'].num_examples
```

Una vez divididos los datos necesitamos normalizarlos.
Recordemos que los valores que se encuentran en el dataset son números del 0 al 255, pero para que nuestro programa Python lo entienda queremos pasarle o 0 o 1.
Creamos una función encargada de normalizar estos datos casteando los datos (para poder operar con ellos) y posteriormente dividiéndolos entre 255.
Los datos normalizados serán 0 cuando ese pixel específico sea negro y 1 cuando blanco.

```
def normalize(images, labels):
    images = tf.cast(images, tf.float32)
    images /= 255
    return images, labels


train_dataset = train_dataset.map(normalize)
test_dataset = test_dataset.map(normalize)
```

Arquitecturas de redes neuronales

En este programa el modelo está creado con una red neuronal recurrente (RNN).
Las redes neuronales recurrentes se conforman por la capa inicial, la final y las ocultas. Las capas ocultas son todas aquellas intermedias entre la inicial y la final.
En estas capas ocultas obtenemos como entrada la salida de la capa anterior. Además, tienen una copia en memoria que permite cambiar la predicción en base a esta.

Por ejemplo, si detectar el número 5 fuese detectar la secuencia 101, la capa oculta recibirá como entrada el 1 y lo guardará en memoria. Posteriormente cuando reciba los siguientes números podrá ajustar la predicción al conjunto total. El modelo no sabrá qué número es aquel que comienza por 1, pero sí podrá intentar predecir cuál es aquel que empieza por 1 y le siguen X números.

Implementación de la RNN

Para crear una RNN e indicar las capas que la componen lo hacemos indicando las capas que la componen. Observamos que en este caso son 4.

La primera capa es ***Flatten*** porque así nos permite convertir los datos de entrada en un formato unidimensional de 28x28 compatible con los datos del dataset. El número final indica los canales de

colores. Si la imagen tuviese color querríamos que este número fuese el 3 (R,G, B) pero en este caso solo necesitamos 1 al tener las imágenes en escala de grises.

El resto de capas son **densas** (*Dense*) lo cual indica que todas las neuronas de la capa están conectadas con todas las de la capa anterior.
Como primer parámetro se indica el **número de neuronas** de esa capa; nos encontramos con 64 y con 10.
Este número podemos variarlo en base a nuestras necesidades, pero para un conjunto de datos de este tamaño 64 neuronas son suficientes.
Cuantas más neuronas asignemos a las capas mayor será el uso de recursos y el tiempo de entrenamiento del modelo.

La **capa final** tiene 10 neuronas, 1 por cada predicción posible (del 0 al 9).

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28,1)), # 28 x 28 píxeles cada imagen
    tf.keras.layers.Dense(64, activation=tf.nn.relu), #Capa oculta 1
    tf.keras.layers.Dense(64, activation=tf.nn.relu), #Capa oculta 2
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)
])
```

Funciones del modelo

Con el método *compile()* estamos configurando el modelo para que pueda ver de mejor manera los patrones en los datos pasados.

Adam es el algoritmo de optimización que se utilizará durante el entrenamiento. Tenemos mucho donde elegir; este específico permite modificar la tasa de aprendizaje conforme cambian las condiciones.

Como función de pérdida (*loss*) tenemos asignada *sparse_categorical_crossentropy* porque es la adecuada para entradas de números enteros.

Como métrica a obtener establecemos la proporción de predicciones correctas realizadas en relación con el número total de predicciones; la *accuracy.*

```
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

Preparación del dataset

Agrupamos los datos en grupos de 64 para entrenar al modelo de manera eficiente y los aleatorizamos y repetimos infinitamente.
Más tarde veremos durante cuánto tiempo (cuántas *epochs; épocas*) entrenamos al modelo, pero dentro de este entreno jamás nos quedaremos sin datos (gracias al *repeat()*), por más largo que sea el tiempo de entreno.

```
BATCHSIZE = 64
train_dataset = train_dataset.repeat().shuffle(num_train_examples).batch(BATCHSIZE) #Aleatorizar los datos
test_dataset = test_dataset.batch(BATCHSIZE)
```

Entreno del modelo

Al comenzar a entrenar un modelo este tiene valores aleatorios en cada conexión entre neuronas. El hecho de comparar las salidas del programa con los resultados correctos permite mejorar el algoritmo sucesivamente mientras el modelo ajusta distintos pesos.
Como parámetros al método *fit()* indicamos el dataset con los datos de entreno, las épocas y los pasos por época.

En cada época (*epoch*) se entrena el conjunto de datos completo; por lo que en 50 épocas se entrena 50 veces toooodo el set.

Los *steps per epoch* son la cantidad de veces que el modelo actualizará sus pesos durante cada época. DIvidiendo el tamaño del set entre el *batchsize* obtenemos el número necesario para cubrir todos los ejemplos.

```
model.fit(
    train_dataset, epochs=50,
    steps_per_epoch=math.ceil(num_train_examples/BATCHSIZE)
)
```

Guardado del modelo

Por último, el modelo es guardado en la ruta en la que se encuentra el script. Para el funcionamiento de la aplicación este modelo ya estará en la propia carpeta del proyecto de Visual Studio.
El modelo puede guardarse tanto en formato *h5* como *keras*.

```
model.save('C:/Users/loren/Downloads/dataset/modelo.h5')
```

Creado con el Personal Edition de HelpNDoc: Haz que la revisión de la documentación sea muy sencilla con el analizador de proyectos avanzado de HelpNDoc

Cargado del modelo

Una vez obtenido el archivo *h5* con el modelo entrenado, se crea otro script Python encargado de llamar a este modelo.

Tras muchas pruebas he visto que lo que mejor ha funcionado es pasarle la lista de bytes por argumentos al script. Esta lista será el que representa el color de cada pixel del dibujo del usuario.

Una vez obtenida la lista se convierte en una pero con un formato específico de *Numpy* (una librería de Python) para trabajar con esta.
Le hacemos un *reshape* a la lista para asegurarnos que tiene el formato requerido por nuestro modelo, siendo 1 el número de imágenes a predecir de manera simultánea, 28 los píxeles de alto, 28 los de ancho y el argumento final indica de nuevo el canal de color (gris).

```
def predecir_exc(arr):
    try:
        modelo = tf.keras.models.load_model("ruta/modelo.h5")

        arr = np.array(arr)
        arr = arr.reshape(1, 28, 28, 1)
```

Realizar la predicción

Llamamos a la función *predict()* del modelo previamente cargado. Le pasaremos como argumento la lista pasada por parámetros y con formato *Numpy* y el número de imágenes a predecir.

```
        prediction_values = modelo.predict(arr, batch_size = 1)
```

Devolución de datos

En *prediction_values* se encuentran todos los números junto a su porcentaje de *accuracy*. Es por esto que si queremos sacar el número que es más probable de ser el dibujado por el usuario queremos sacar el máximo. Esto lo conseguimos con la función *argmax()* de *Numpy.*

```
       prediction = str(np.argmax(prediction_values)) #Extrae la que mayor certeza tiene
```

```
        print("Predicción final:", prediction)
```

El hecho de imprimirlo por pantalla nos será útil para más tarde implementarlo en .NET.

Ejecución del programa

En el main nos encontramos una variable que almacena la lista introducida por argumentos.
Para que la aplicación funcionase mucho más rápido la lista está serializada en JSON (más adelante veremos cómo), porque pasar cada número como argumento independiente (784 argumentos) no era nada óptimo y costaba mucho rendimiento.

Es por ello que lo deserializamos importando la librería *json* y llamando a su función *loads()*

```
def main():
    serialized_array = sys.argv[1]

    # Deserializar el array desde JSON
    arr = json.loads(serialized_array)

    # Llamar a la función de predicción con el array deserializado
    predecir_exc(arr)
```

Creado con el Personal Edition de HelpNDoc: Actualice su proceso de documentación con una herramienta de creación de ayuda

Implementación con .NET

El *frontend* de esta aplicación está realizado utilizando tecnología *.NET;* específicamente *MAUI*.
MAUI *(Multi-Platfrom App UI)* es una tecnología que ofrece la gran ventaja de ser multiplataforma, permitiéndonos así correr las aplicaciones en Windows, Mac, Android e iOS.
Debido a problemas con la implementación del script en Python este programa **corre únicamente en Windows** (si miramos el código tendremos varios warnings del estilo *Esto podría no funcionar en X plataforma*)**.**

Implementación de Python en C#

Pero si el script que se encarga de realizar predicciones está desarrollado en Python, ¿cómo podemos desde Visual Studio llamar a este?
Hay paquetes de Visual Studio que podemos instalar fácilmente los cuales nos prometen compatibilidad con Python. Razón no les falta, con muy pocas líneas de código puedes integrar fácilmente tu script desarrollado en Python, pero el problema viene cuando necesitamos cargar librerías externas. Es decir, cuando tenemos varios *import* en el script. En este caso tenemos el de *Tensorflow, Numpy* y *JSON.*

Herramientas de implementación

IronPython y *Pythonnet* son herramientas muy utilizadas pero con gran cantidad de problemas al tratar de implementarlas en MAUI. Codificaciones incorrectas, escasez de permisos para acceder a ciertos archivos y problemas de memoria son algunos de los problemas que me encontré con ellas.
Curiosamente la solución de muchos de estos problemas estaba en cambiar la plataforma del proyecto y no dejar la predeterminada *Any CPU*, pero al estar realizando la aplicación con tecnología *MAUI* este cambio de plataforma no tenía sentido y daba problema por todos lados.

¿Entonces?

Alternativas

La alternativa a tantos problemas de compatibilidad era directamente ejecutar una terminal por debajo pero sin mostrársela al usuario.
Esto podemos lograrlo así:

```
                string pythonScript = @"ruta\cargarModelo.py";

                ProcessStartInfo start = new ProcessStartInfo();
                start.FileName = @"ruta\python3.11.exe"; // Comando para ejecutar
Python

                start.Arguments = $"{pythonScript} {serializedArray}"; // Pasar el
array serializado como argumento al script Python

                start.UseShellExecute = false; // No usar shell para la ejecución
                start.RedirectStandardOutput = true; // Redireccionar salida estándar
                start.CreateNoWindow = true; // No crear ventana
```

Captura de la salida y mostrado en la interfaz

Si capturamos la salida del programa con *ReadToEnd()*, se almacenará todo lo mostrado por pantalla por el script de Python. Es por esto que anteriormente comentamos que el *print* con la predicción final nos sería de ayuda.

Crearemos una expresión regular para únicamente mostrar en nuestra UI el número con mayor predicción. En Python lo mostrábamos por pantalla como *Predicción final: N*, por lo que la expresión regular filtrará por un número tras 2 puntos (: [0-9]).
Obtenemos el número y con un *replace()* reemplazamos estos 2 puntos por un espacio vacío.
Ahora nuestra variable *reemplazar* contiene únicamente el número con mayor *accuracy*.

```
                // Inicia el proceso
```

```
using (Process process = Process.Start(start))
{
    // Leer la salida estándar del proceso Python
    string output = process.StandardOutput.ReadToEnd();
    //reemplazar = output;

    // Esperar a que el proceso Python termine
    process.WaitForExit();

    // Mostrar la salida del script Python
    string pattern = @": [0-9]";
    //string patternOthers = @"(?<=STARTHERE.*?)aa(?=.*?ENDHERE)";


    MatchCollection matches = Regex.Matches(output, pattern);
    reemplazar = matches[1].Value.Replace(":", "").ToString();
```

Todo este proceso requiere que el script se ejecute con el tiempo que ello conlleva. Para evitar que el programa se congele mientras se calcula la predicción (sobre unos 3 segundos), encapsulamos esto en una función y esta en un hilo.
Al llamar a esta función (que la llamaremos al accionar el botón de predicción) se muestra un indicador de carga (*ActivityIndicator*), y al obtener el resultado final del script este se vuelve a detener, haciéndole saber así al usuario que la predicción ha terminado.

Creado con el Personal Edition de HelpNDoc: Convertir documentos de Word en libros electrónicos: una guía paso a paso con HelpNDoc

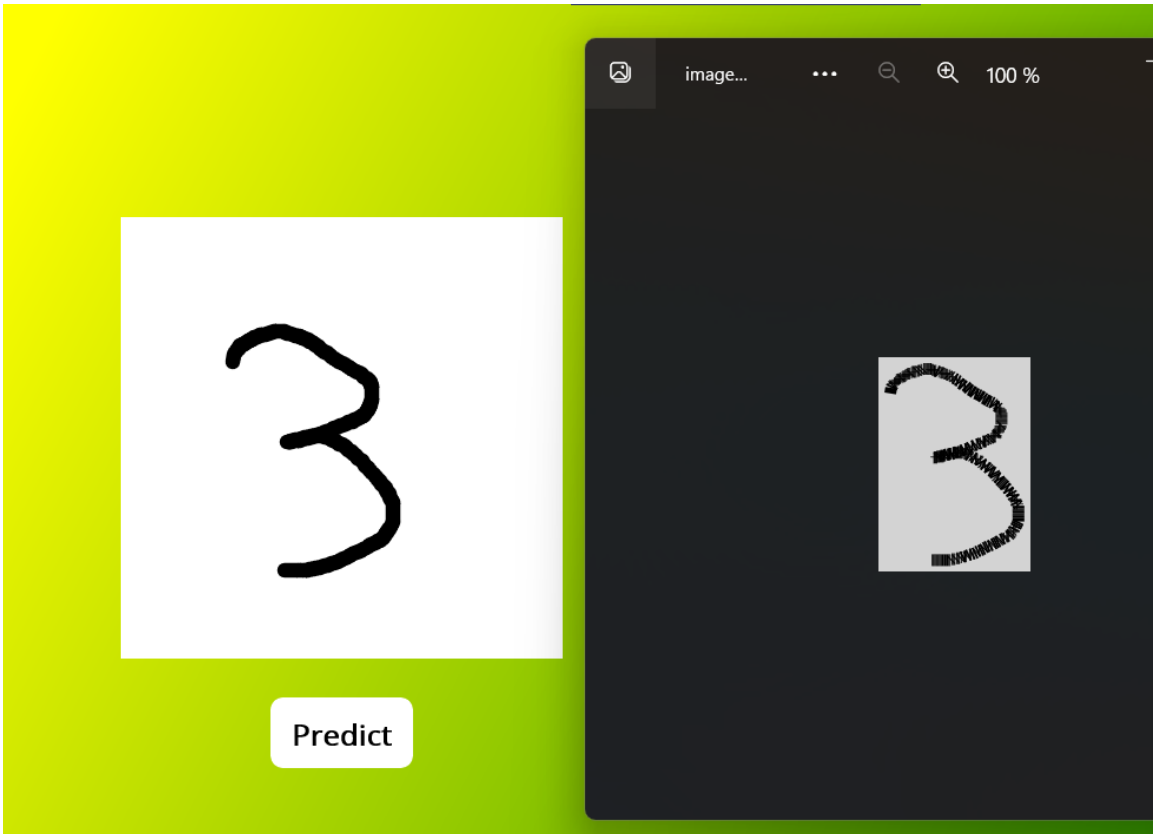
De líneas a bytes

Ya tenemos la implementación para poder pasarle la lista al script pero, ¿cómo obtenemos esta lista de números?
Recordemos que en la lista deben encontrarse 784 números con valor o 0 o 1.

Para ello en la función llamada al hacer clic sobre el botón *Predict* obtenemos el flujo dibujado por el usuario, lo cargamos a un flujo de memoria y este flujo lo pasamos a una imagen.

```
    private async void OnPredictClicked(object sender, EventArgs e)
    {
        try
        {
            int pixelesRequeridos = 28;
            using var stream = await drawingCanvas.GetImageStream(pixelesRequeridos,
pixelesRequeridos);
            using var memoryStream = new MemoryStream();
            stream.CopyTo(memoryStream);

            stream.Position = 0;
            memoryStream.Position = 0;
            await System.IO.File.WriteAllBytesAsync(@"C:
\Users\loren\Downloads\imagen.png", memoryStream.ToArray());
```

Sin embargo esta imagen es guardada a partir del trazo del propio usuario; no respeta los márgenes. Esto suponía un problema ya que las imágenes con las que ha sido entrenado el modelo son imágenes con márgenes en todos los lados.



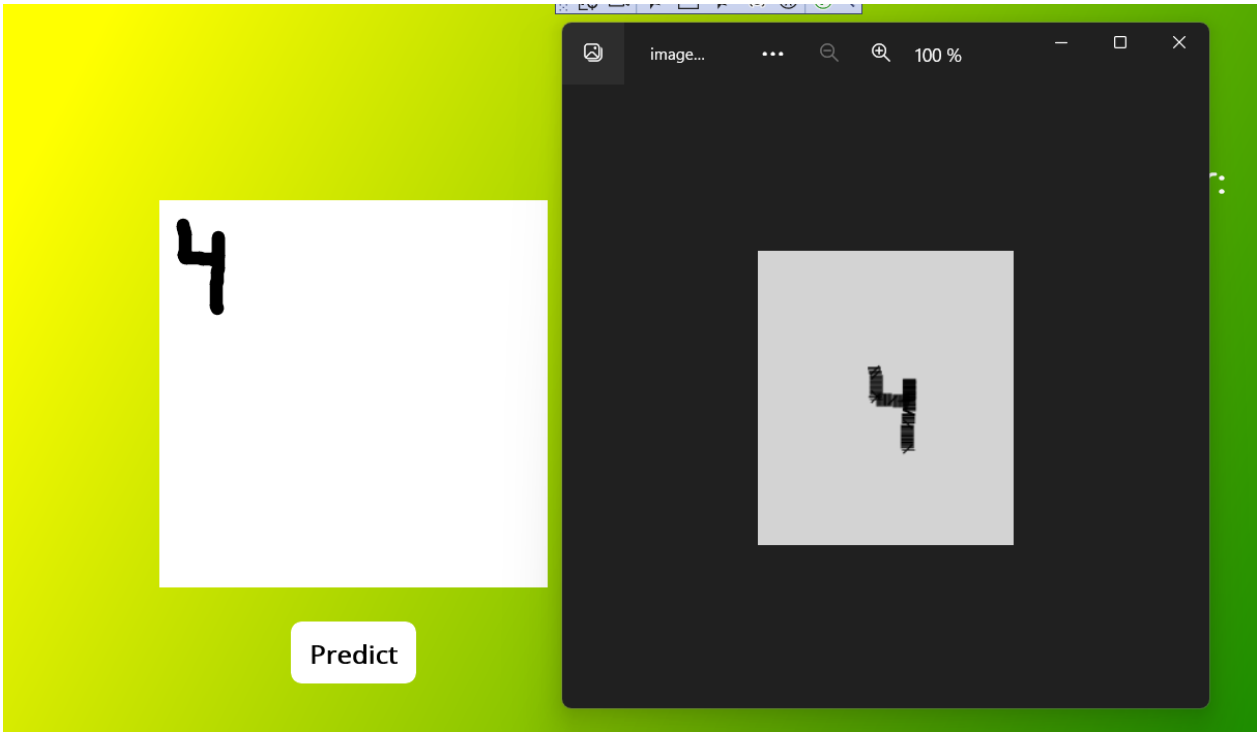
Es por este motivo que necesitamos añadir manualmente los márgenes.
Para ello creamos una nueva imagen haciendo uso de la clase *Bitmap* con la misma anchura y altura que la original pero con 200 píxeles más. Estos píxeles podemos adaptarlos a cada caso, pero con 200 he obtenido buenos resultados ya que tomamos como imagen original la del canvas con tamaño 300 x 300; antes de redimensionarlas a 28 x 28.

Con la clase *Graphics* podemos centrar la imagen original en la nueva y guardarla en memoria para utilizarla a la hora de pasar la imagen a bytes.

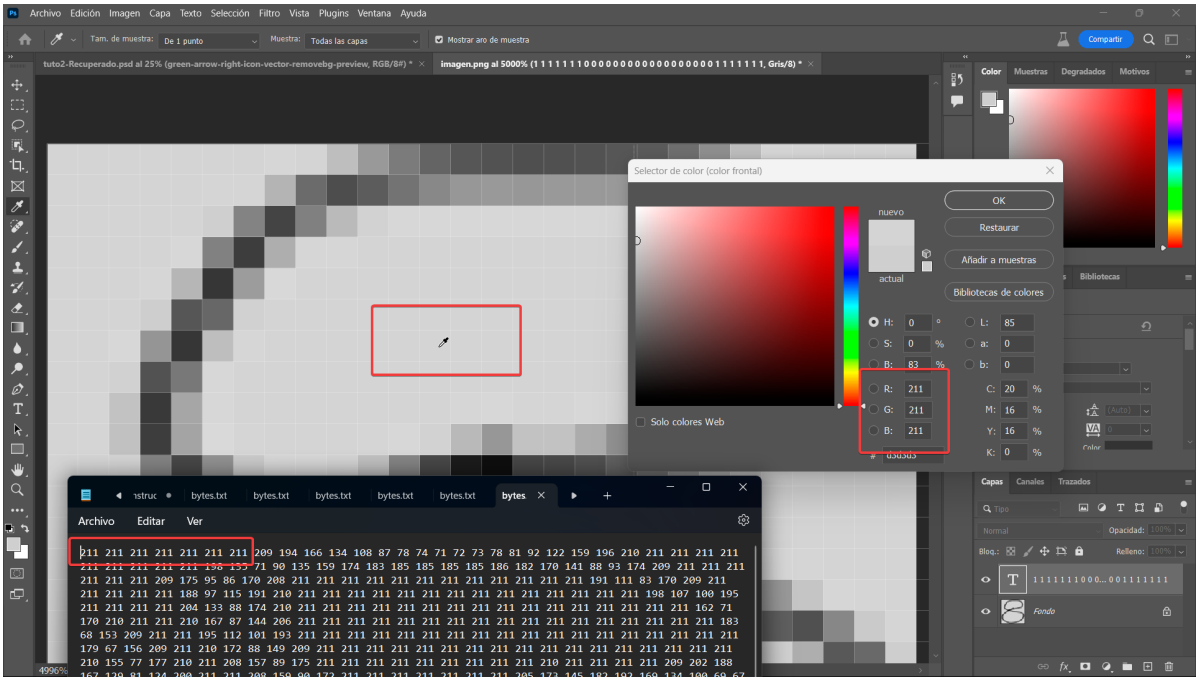
```
            using var originalImage = System.Drawing.Image.FromStream(memoryStream);

            int newWidth = originalImage.Width + 200; // 200 píxeles de padding a cada
lado
            int newHeight = originalImage.Height + 200; // 200 píxeles de padding
arriba y abajo
            using var paddedImage = new Bitmap(newWidth, newHeight);

            // Dibujar la imagen original en la nueva imagen centrada dentro del
padding
            using (Graphics g = Graphics.FromImage(paddedImage))
            {
                g.Clear(System.Drawing.Color.LightGray); // Color del padding mismo
```

```
que la imagen original
                int x = (newWidth - originalImage.Width) / 2;
                int y = (newHeight - originalImage.Height) / 2;
                g.DrawImage(originalImage, x, y);
            }
            paddedImage.Save(@"ruta\imagen_con_padding.png",
System.Drawing.Imaging.ImageFormat.Png);
```

Además, de esta manera no nos influye tanto en la predicción la posición en la que el usuario dibuje el número, ya que sea como sea la imagen original siempre será colocada en el centro y sus márgenes extras añadidos.
Aún así la posición y tamaño siguen siendo factores a tener en cuenta, ya que si el número es tan pequeño cuando reescalemos la imagen a 28 x 28 píxeles el modelo perderá mucha precisión.



No llegaba a entender los valores que se imprimían al intentar convertir los píxeles individuales de la imagen a números; el valor 211 se repetía mucho cuando debía ser el 255 (el blanco).
Al abrir la imagen reescalada a 28 píxeles con Photoshop me di cuenta de que (aunque es algo que se ve a simple vista no me había percatado) por algún motivo al obtener la imagen de lo dibujado con *GetImageStream(),* ¡el fondo siempre es gris! No es blanco como en el canvas.
Por lo tanto, el 211 es el gris del fondo.
Haciendo una leve modificación en el código haremos que los píxeles a tomar en cuenta como pintados sean los que tengan valor inferior a 200.




```
            using (var fileStream = new FileStream(@"ruta\imagen_con_padding.png",
FileMode.Open))
            {
                using (var ms = new MemoryStream())
                {
                    await fileStream.CopyToAsync(ms);
                    fileStream.Position = 0;
                    ms.Position = 0;

                    byte[] pixeles = Convert(ms);
                    //ShowArray(pixeles);

                    for (int i = 0; i < pixeles.Length; i++)
                    {
                        if (pixeles[i] < 200) //<200 son los de color negro, el resto
forman parte del fondo gris
                        {
                            pixeles[i] = 1;
                        }
                        else
                        {
                            pixeles[i] = 0;
                        }
```

```
}
```

Una vez normalizado el array de píxeles (pasados los valores a 0 o 1), lo convertimos en un array de *float* para no tener problemas de compatibilidad al pasarlo como argumento al script.
Lo pasamos en la función *ExecuteScript(),* la cual es la que anteriormente comentábamos que ejecutaba por debajo una terminal.

```
float[] floats = new float[pixeles.Length];
for (int i = 0; i < pixeles.Length; i++)
{
    floats[i] = pixeles[i];
}

await ExecuteScript(floats);
```

Lo mostramos en la interfaz estableciendo el *Text* del label con el texto pasado por su expresión regular correspondiente.
Al mismo tiempo desactivamos el indicador de carga.

```
lblPrediction.Text = reemplazar;

loadingIndicator.IsRunning = false;
```

Creado con el Personal Edition de HelpNDoc: Experimenta el poder y la simplicidad de la interfaz de usuario de HelpNDoc

Interfaz gráfica

La interfaz gráfica está formada por un *grid* metido en un *ScrollView*. Este *grid* contiene 2 columnas y 8 filas.
En estas filas están distribuídos componentes como el label que conforma el título, la imagen indicativa del funcionamiento básico, el *canvas,* el botón y el *label* destinado a mostrar la predicción.

MAUI no tiene de manera nativa un *canvas*, por lo que este ha sido añadido con la *MAUI Community ToolKit.*
Esta puede obtenerse como paquete NuGet, pero deberemos añadirla manualmente en el fichero *MauiProgram.cs*

```
public static MauiApp CreateMauiApp()
{
    var builder = MauiApp.CreateBuilder();
    builder
        .UseMauiApp<App>()
```

```
                .UseMauiCommunityToolkit()
```

Además añadiremos el espacio de nombres en el fichero *XML* de nuestra interfaz.

```
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:toolkit="clr-
namespace:CommunityToolkit.Maui.Views;assembly=CommunityToolkit.Maui"
```

Ya podremos añadir tanto este como cualquier componente del kit de la comunidad de *MAUI*.

```
                <toolkit:DrawingView
                    LineColor="Black"
                    x:Name="drawingCanvas"
                    HeightRequest="300"
                    WidthRequest="300"
                    LineWidth="10"
                    BackgroundColor="White"
                    Grid.Row="6"
                    Grid.Column="0"
                    />
```

El fondo gradiente se ha conseguido con un *Frame* y creando una brocha con un gradiente lineal de color amarillo desde la posición 0.08 (8%) hasta el color verde en la posición 1 (el 100% del frame).
Este gradiente se aplica como fondo al *Frame*, lo que nos permite conseguir un fondo que cambia gradualmente de amarillo a verde.

```
    <Frame BorderColor="White"
       HasShadow="True"
       CornerRadius="12">
        <Frame.Background>
            <LinearGradientBrush>
                <GradientStop Color="Yellow"
                              Offset="0.08" />
                <GradientStop Color="Green"
                              Offset="1" />
            </LinearGradientBrush>
        </Frame.Background>
```

Se han utilizado 2 fuentes personalizadas almacenadas en la carpeta *Fonts* y añadidas en el *MauiProgram.cs*.

```
                    fonts.AddFont("LTSaeada-Light.otf", "Saeada");
                    fonts.AddFont("GalleroVintage-DemoVersion-Regular.otf",
"GalleroVintage");
```

Creado con el Personal Edition de HelpNDoc: Facilita la documentación con la interfaz de usuario limpia y eficiente de HelpNDoc

Manual de usuario

Creado con el Personal Edition de HelpNDoc: Transforme su proceso de creación de archivos de ayuda CHM con HelpNDoc

Requisitos previos

Para que el programa funcione de manera óptima necesita las **librerías** indicadas en el *requirements.txt.*

Estas son la versión 2.12 de ***Tensorflow*** y la 1.23.5 de ***Numpy***.

Podemos realizar la instalación de ellas con ***pip install -r requirements.txt***.

Además, la **versión de Python debe ser la 3.11**. Esta puede instalarse desde la web oficial.

Es importante que durante la instalación marquemos la opción de añadir Python al Path. En caso de no tener Python en el Path, deberá modificarse el código con la ruta de instalación del propio Python.

Actual:

```
                start.FileName = "python3.11.exe"; // Comando para ejecutar Python
```

Cambio si no está en el path:

```
                start.FileName = @"C:
\Users\loren\AppData\Local\Microsoft\WindowsApps\python3.11.exe"; // Comando para
ejecutar Python
```

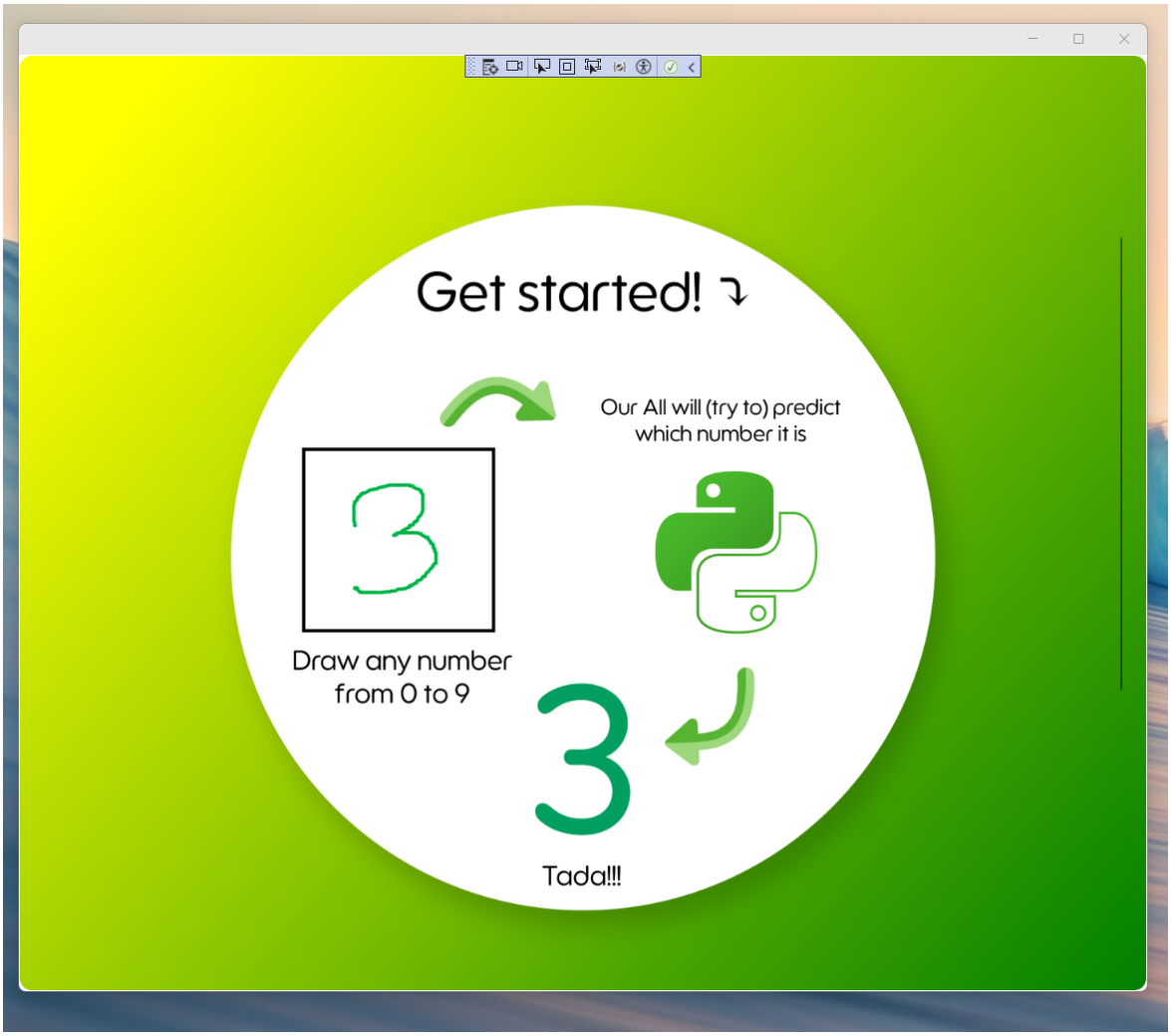
Creado con el Personal Edition de HelpNDoc: Por qué Microsoft Word no está recortado para la documentación: los beneficios de una herramienta de creación de ayuda

Interfaz

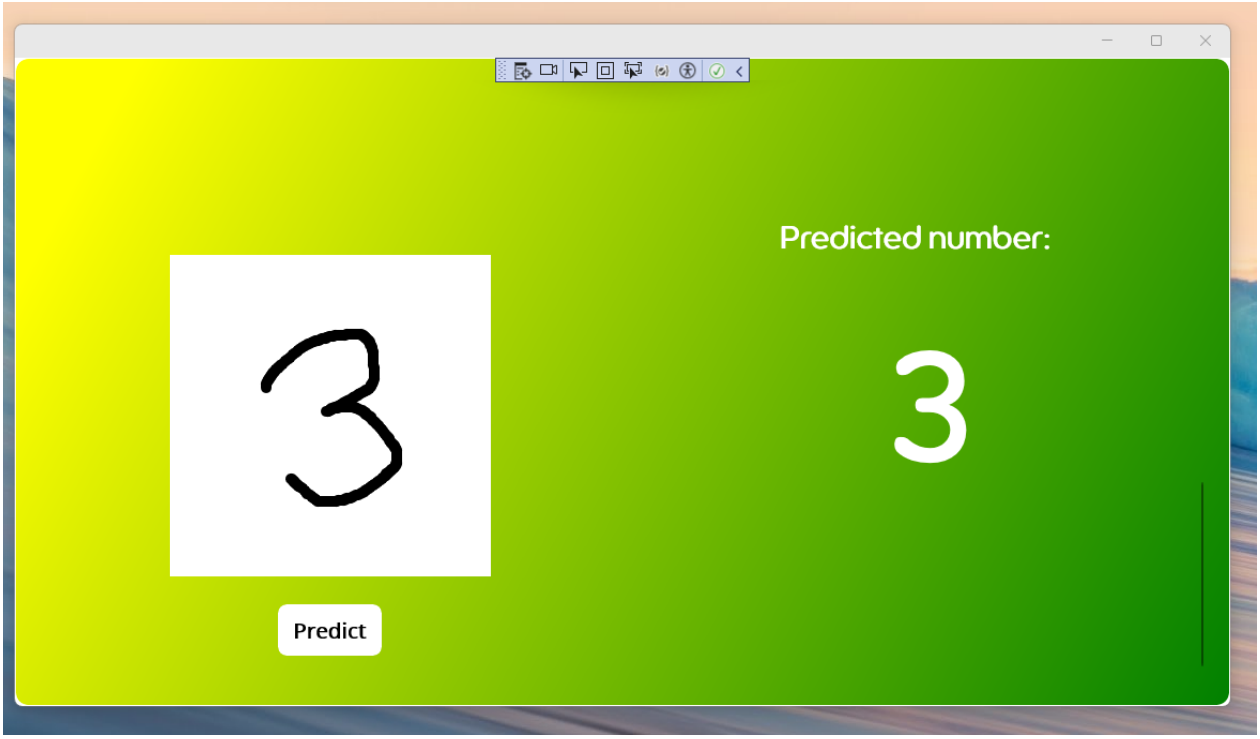
Usabilidad

La interfaz de la aplicación es sencilla; tenemos una UI que nada más abrirla nos invita a deslizar hacia abajo.

Al deslizar nos encontramos una noción del proceso; de un vistazo se le muestra al usuario que debe dibujar en el lienzo y el programa mostrará el resultado predecido por pantalla.



Si el usuario sigue desplazando se encuentra el lienzo y el botón de predicción bajo este.
Al presionar el botón aparecerá una animación de carga mientras se esté calculando la predicción. Esta desaparecerá cuando el resultado esté listo.



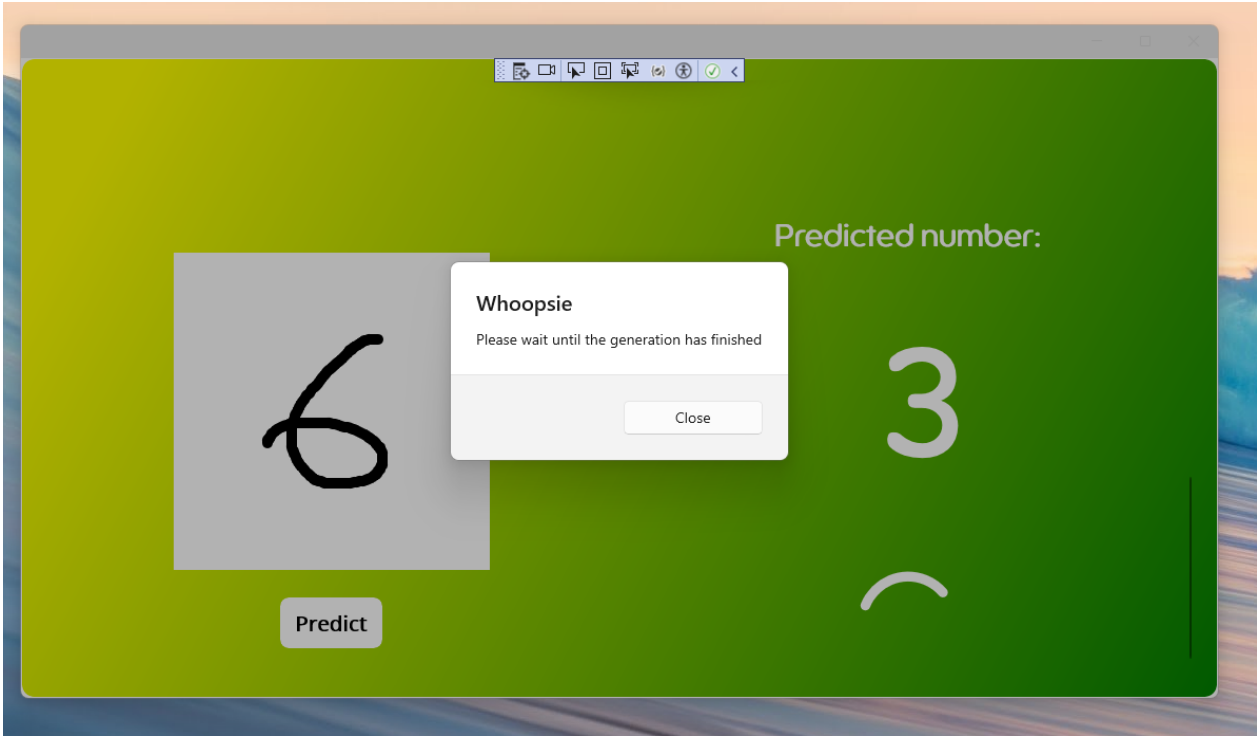
Creado con el Personal Edition de HelpNDoc: Generador de documentación con todas las funciones

Solución de errores

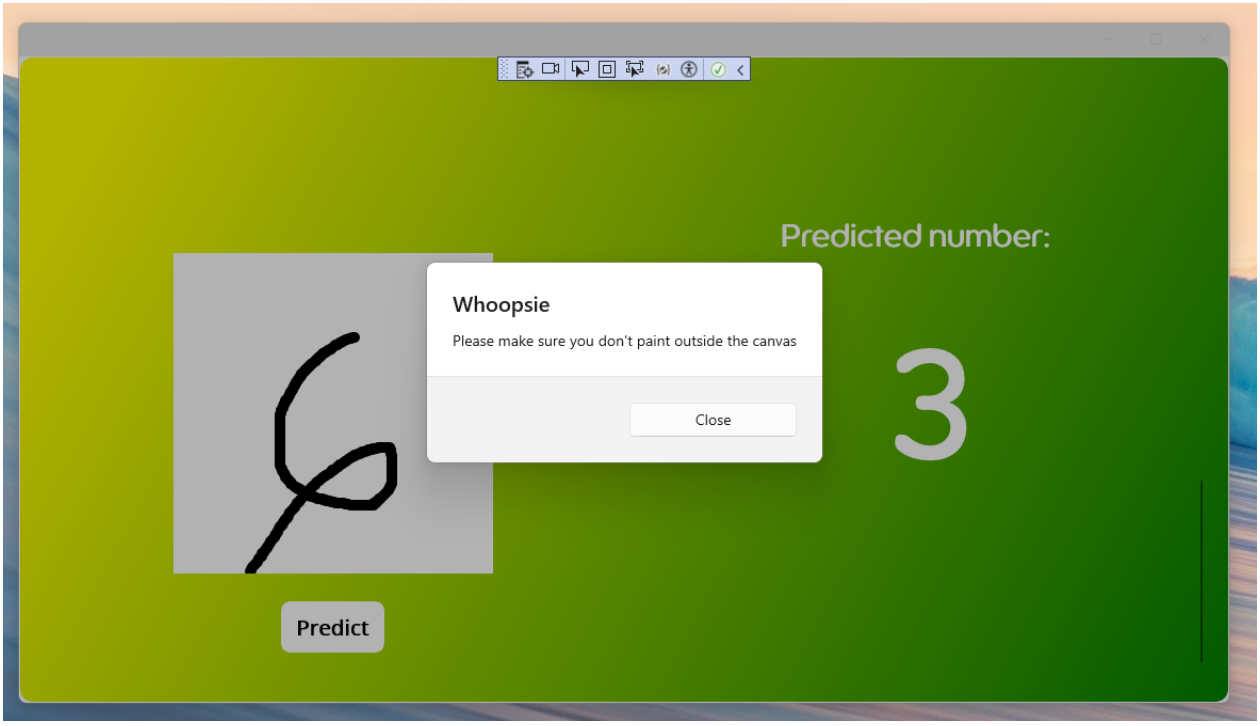
Solución de errores

Se han observado 4 errores específicos que pueden ocurrir al interactuar con la aplicación.

- **Predicción antes de la finalización de la anterior**: Si se pulsa el botón mientras se está realizando otra predicción se produce una excepción. Esta se ha controlado y nos muestra un mensaje indicando que debemos esperar antes de volver a pulsar el botón.

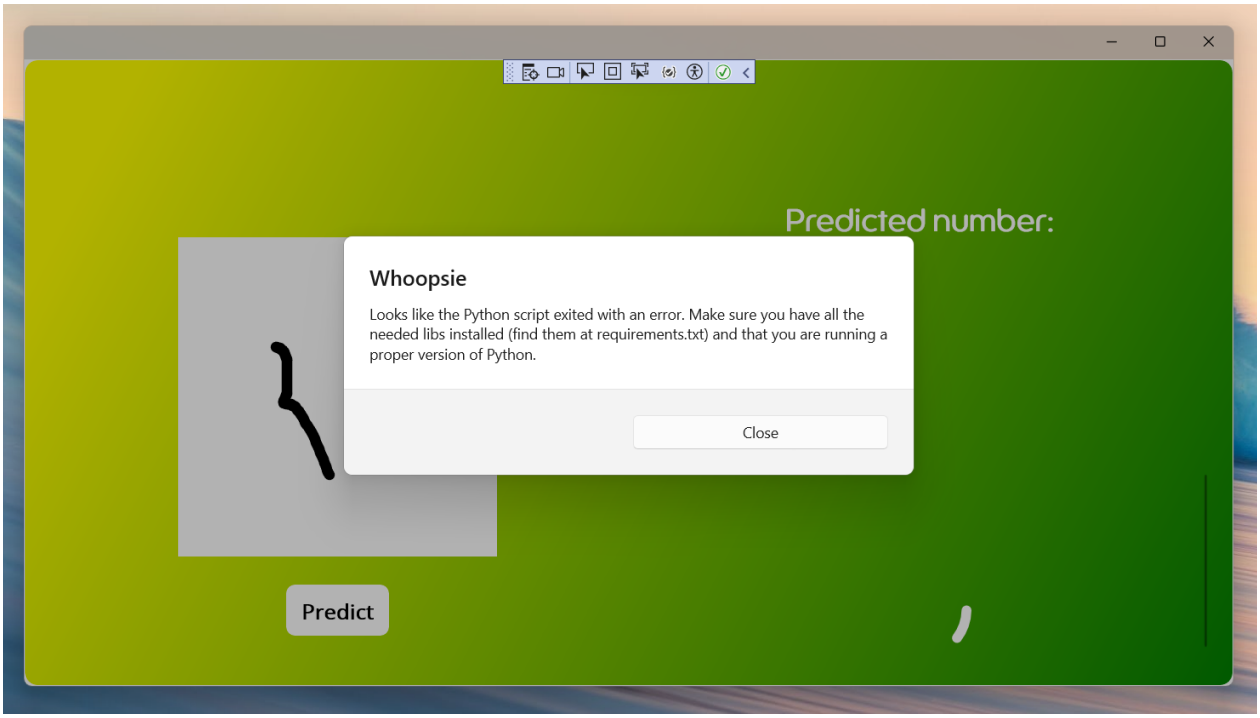


- **Pintado fuera del *canvas*:** Si el usuario mantiene el pincel mientras se sale del canvas, aunque esté delimitado por los bordes esto genera una excepción. De ser así, el dibujo no sería válido y se controla de la misma manera.

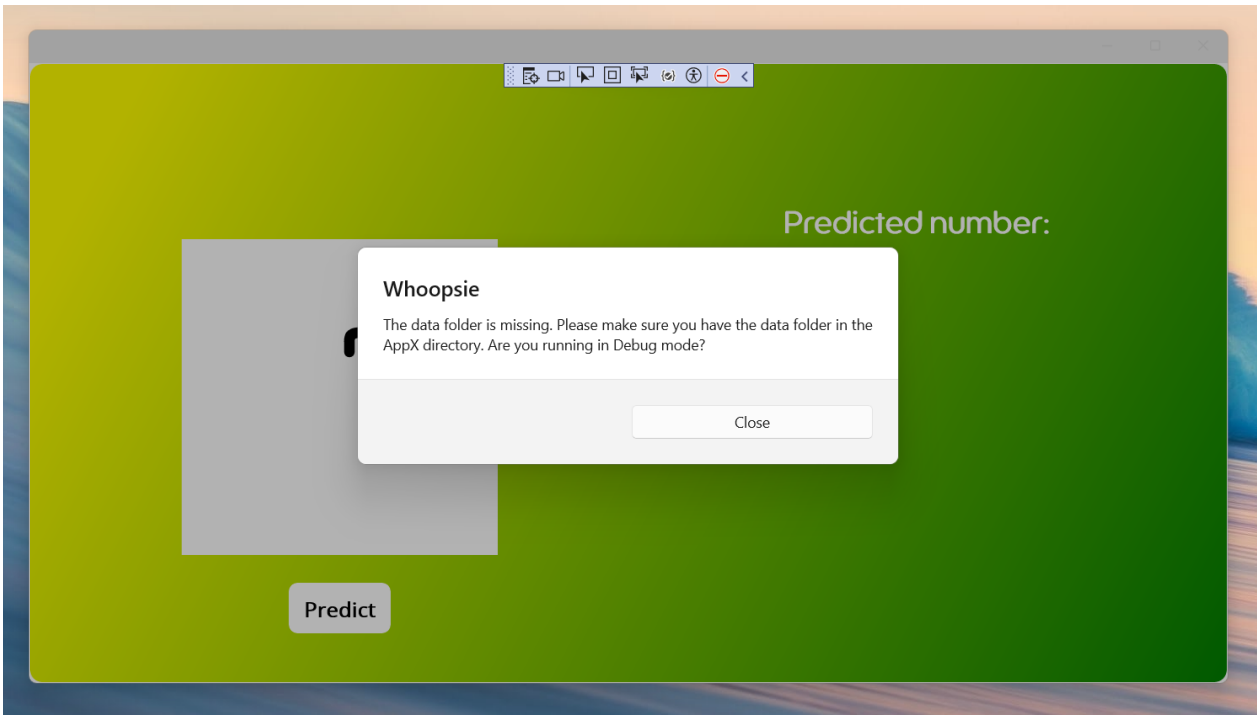


- **Fallo en la ejecución:** Si por cualquier motivo la predicción no puede realizarse, no se mostrará ningún número como resultado. El usuario puede solucionar esto instalando las librerías necesarias (las cuales se encuentran en el fichero *requirements.txt*). Instalarlas solo requiere ejecutar el

comando *pip install -r requirements.txt* en la terminal. Además, se debe ejecutar con la versión 3.11 de Python. Para ver la versión del sistema podemos ejecutar *python --version*.



- **No se ha encontrado el directorio *data:*** En él es donde se encuentran tanto el modelo como el script necesario para hacer las predicciones. Esta carpeta viene creada previamente al realizar la descarga del proyecto; prueba a eliminar el existente y descárgalo de nuevo. También puede ocurrir si se ejecuta desde otra ruta la cual no es la esperada; distinta a la ruta desde la que se ejecuta en modo *Debug.* Este error puede ocurrir al ejecutar el programa en *Release*.



Creado con el Personal Edition de HelpNDoc: Maximiza las capacidades de tu archivo de ayuda CHM con HelpNDoc

Más ayuda

Si han surgido problemas no contemplados en este manual puedes contactarme en el correo *lorenasanchezdev@gmail.com* o en *Github*.

Creado con el Personal Edition de HelpNDoc: Creación de ayuda CHM, PDF, DOC y HTML desde un solo lugar