**MODELADO ML PARA DETECCIÓN DE OBJETOS EN EL PLANO.**

**PLANTEAMIENTO**

En primera instancia se plantea un problema de regresión lineal que cumpla con criterios de posicionamiento y rotación de objetos en un plano tal que permita procesar imágenes de vista superior grabadas en tiempo real para alimentar un sistema que arroja por producto las coordenadas del vector v(x,y) y su rotación tan(β)=y/x respecto al eje de coordenadas predefinido. El sistema debe ser capaz de leer y extraer características típicas de una imagen de una figura predeterminada para dar con su ubicación y rotación desde el punto de referencia de una marca roja en uno de sus vértices, solucionando un problema de regresión con aprendizaje automático.

**METODOLOGÍA**

**Flujo de sistema**

El flujo lógico será lo correspondiente a la lectura de las imágenes de entrenamiento, el acondicionamiento de las mismas y preparación del set de datos, configuración, alimentación y entrenamiento del sistema, posteriormente, una evaluación a corde a la selección de métricas de rendimiento para penalización de errores atípicos y prueba de modelo con datos nuevos.

El sistema analiza una imagen que reduce a 128x128(px), aplana y normaliza de 0 a 1, identificando características típicas de un objeto y un punto de coordenadas pintado de color rojo en uno de sus vértices. Así que el sistema agrega esta información como característica a un set de datos de forma automática combinando conversión de imágenes a texto plano y las salidas esperadas correspondientes a cada característica, esto para la etapa de entrenamiento supervisado. Inicialmente el sistema debe poder identificar las coordenadas del punto rojo para determinar la ubicación en el plano, pero adicional, identificar el ángulo de rotación respecto al eje horizontal y mostrar gráficamente dichos valores en una interfaz gráfica simple utilizando api web sockets con flask y react.

**Tecnologías y dependencias**

Las tecnologías a utilizar son…

**Modelo.**

El modelo seleccionado es….

**Métricas de desempeño**.

Analizar el desempeño de predicción del modelo…

**DESARROLLO**

**Creación del entorno (dependencias, librerías y estructura)**

Para crear el entorno en python que contiene todas las dependencias básicas para ejecutar un proyecto.

* python -m venv venv

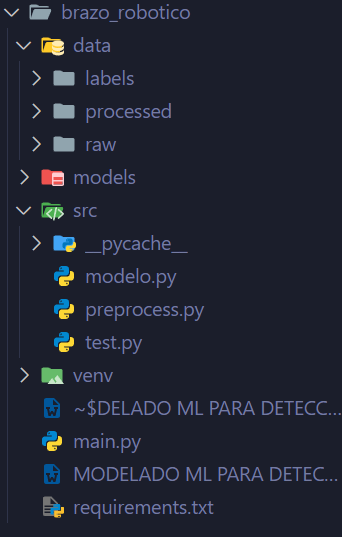
Para instalar las librerías que se van a utilizar en el proyecto( opencv , sklearn , pandas, numpy, mathplotlib)

* pip install opencv-python scikit-learn matplotlib pandas numpy

Para correr el archivo main.py.

* python main.py

La estructura de carpetado debe contener los fragmentos de cada etapa de programa, asi, se divide en: data, models, src, venv, main y requirements.



La data es donde se almacenan los datos de inicio de entrenamiento, los datos procesados y las etiquetas con la información de cada imagen, por tanto, queda dividido así:



Donde labels, son las coordenadas (x,y) y la rotación de cada cubo en cada imagen; processed son los archivos procesados y raw son las imágenes en crudo, de tipo jpg que ingresan al sistema inicialmente.

La carpeta models, contiene los parámetros y resultados de aprendizaje del modelo ML que aprendió con un conjunto de datos y una configuración determinada. Allí se puede inferir el modelo para ser consumido.

La carpeta src contiene todos los recursos como funciones y lógica de proceso.

La carpeta venv contiene todas las configuraciones, características y dependencias del entorno python.

El archivo main.py contiene la lógica de ejecución de todos los archivos de recurso de la carpeta src.

El archivo requirements.txt contiene todas las dependencias y sus versiones utilizadas en el sistema operativo y en el entorno de programación para cuando se desee migrar el modelo a un servidor en la nube para poder ser consumido

Librerias:

* Scikitlearn: librería para desplegar, entrenar y evaluar modelos ML básicos
* Opencv: librería para cargar, procesar y extraer características de las imágenes
* Pandas: librería de python para almacenar, estructurar, depurar, importar o exportar sets de datos.
* Numpy: librería de python para cálculos con arreglos multidimensionales y demás.
* Mathplotlib: librería para visualización estadística de datos.

**Muestreo, acondicionamiento y filtrado de datos (datacleaning)**

1. Se cargan las imágenes relativas al entrenamiento, o el set de datos de entrenamiento en la carpeta “raw” en formato jpg.
2. Se carga un archivo .csv con los datos de los labels para todas las imágenes, es decir, las diferentes características de cada imagen con su respectiva etiqueta.
3. Se crea un archivo de preprocesado en la carpeta src que contenga la lógica para extraer las imágenes reducirlas de pixelado, aplanarlas y normalizarlas en archivos. npy para luego ser guardadas dentro de la carpeta processed.

*import* os

*import* cv2

*import* numpy *as* np

*import* pandas *as* pd

def convertir\_imagen(*ruta*,*i*):

    imagen = cv2.imread(*ruta*) *#Cargar imagen y la almacena en la variable imagen*

*if* imagen is None:

*raise* FileNotFoundError(f'No se pudo cargar la imagen {*ruta*}')

    imagen = cv2.resize(imagen,(128,128)) *#Redimensionar la imagen a 128x128*

*# imagen = cv2.cvtColor(imagen, cv2.COLOR\_BGR2GRAY) #Convertir la imagen a escala de grises*

    imagenmpy = imagen.flatten() *#Aplanar la imagen y devolverla. flatten convierte la matriz en un arreglo unidimensional*

    imagenmpy = imagenmpy/255.0 *#Normalizar la imagen dividiendo cada pixel por 255.0*

*return* {"imagen1":imagenmpy}

def procesar\_imagen(*dir\_entrada*, *dir\_salida*):

    i = 0

    archivos = os.listdir(*dir\_entrada*)

    archivos\_ordenados = sorted(archivos, *key*=lambda *x*: int(''.join(filter(str.isdigit, *x*))))

    print(archivos\_ordenados)

*if* not archivos:

*raise* FileNotFoundError(f'No se encontraron archivos en el gestor de muestras')

*for* file *in* archivos\_ordenados:

*if* not file.endswith('.jpg'):

*raise* FileExistsError(f'El archivo {file} no es una imagen jpg')

        ruta = os.path.join(*dir\_entrada*, file)

        imagen\_procesada= convertir\_imagen(ruta,i)

        np.save(os.path.join(*dir\_salida*, f"muestra{i}\_reducida.npy"), imagen\_procesada["imagen1"]) *## se guarda la imagen con numpy porque numpy ofrece una forma de guardar y cargar matrices dimensionales eficiente.*

        i+=1

        print(f'Imagen {file} guardada en {*dir\_salida*}')

1. una vez se obtenga los datos de archivo .npy guardados, se extraen por medio de otra función que se encarga de apilarlos como un arreglo de diccionarios con “key” = “imagenes”. Se convierte en dataframe con ayuda de la librería pandas y se almacena en una variable.
2. En principio para obtener un dataset completo, es decir cada fila de una característica tanto con los labels (salidas o variables dependientes) como con las imágenes planas y normalizadas, se deben tener dos dataframes tipo pandas. Por tanto se usa la función de la librería que permite leer el archivo csv y convertirlo a dataframe.
3. Se juntan ambos dataframes para obtener un set completo y correpondiente. Después de esto existen técnicas de depuración y data cleaning por si el set es muy extenso o puede estar con vacios, incongruencias o elementos indeterminados.

def crear\_dataframe(*dir\_processed*,*dir\_labels*):

    procesado = []

    i=0;

    archivos\_procesados = os.listdir(*dir\_processed*)

    archivos\_ordenados = sorted(archivos\_procesados, *key*=lambda *x*: int(''.join(filter(str.isdigit, *x*))))

*for* file *in* archivos\_ordenados:

*if* not file.endswith('.npy'):

            print (f'El archivo {file} no es un archivo numpy')

*continue*

        dir\_de\_cada\_archivo = os.path.join(*dir\_processed*, file)

        numpyarray= np.load(dir\_de\_cada\_archivo)

        procesado.append(numpyarray)

        print(f'Imagen {file} cargada en el dataframe')

        i+=1

    procesado = np.array(procesado)

    columnas\_pixeles =[f'pixel\_{i}' *for* i *in* range(procesado.shape[1])]

*#data frame con las imagenes procesadas*

    imagenes = pd.DataFrame(procesado, *columns*=columnas\_pixeles)

*#data frame con los labels o salidas*

    labels = pd.read\_csv(*dir\_labels*, *sep*=';')

*#dataframe completo*

    fullDataFrame = pd.concat([labels,imagenes], *axis*=1)

*return* fullDataFrame

**## main.py**

*import* os

*from* src.preprocess *import* procesar\_imagen, crear\_dataframe

*from* src.modelo *import* preentrenamiento, entrenamiento\_final

*from* src.test *import* test\_model

*from* src.data\_augmentation *import* data\_augmentation

*import* numpy *as* np

*import* pandas *as* pd

dir\_processed = 'data/processed'

dir\_raw = 'data/raw'

dir\_labels = 'data/labels/images.csv'

dir\_test = 'data/test'

dir\_raw\_augmentation = 'data/raw\_augmentation'

*if* not os.path.exists(dir\_processed):

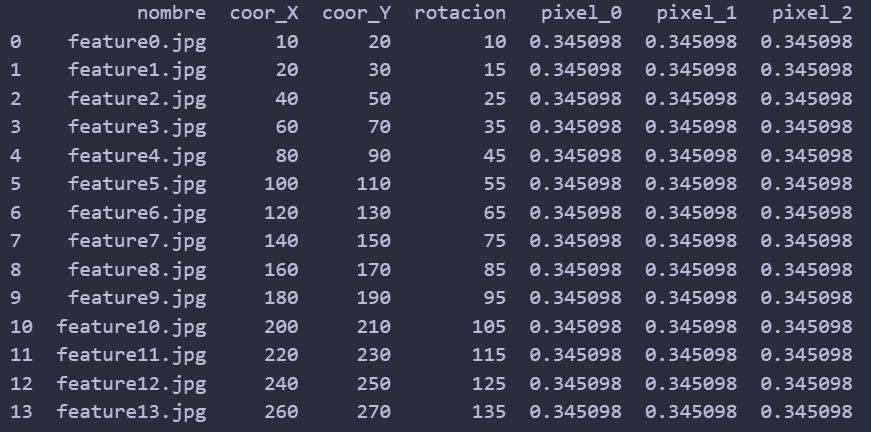
    os.makedirs(dir\_processed)

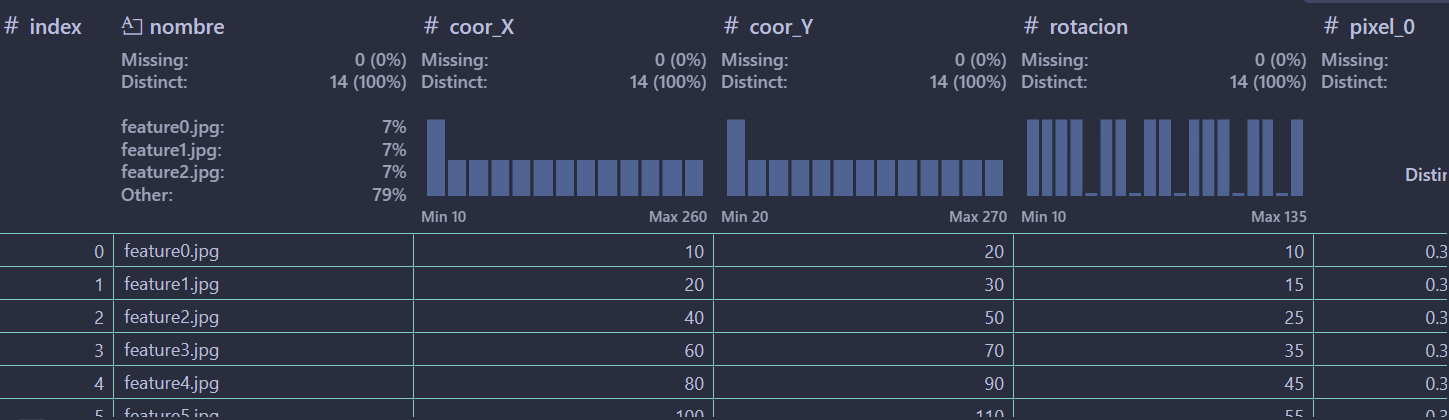
procesar\_imagen(dir\_raw, dir\_processed)

df= crear\_dataframe(dir\_processed,dir\_labels)

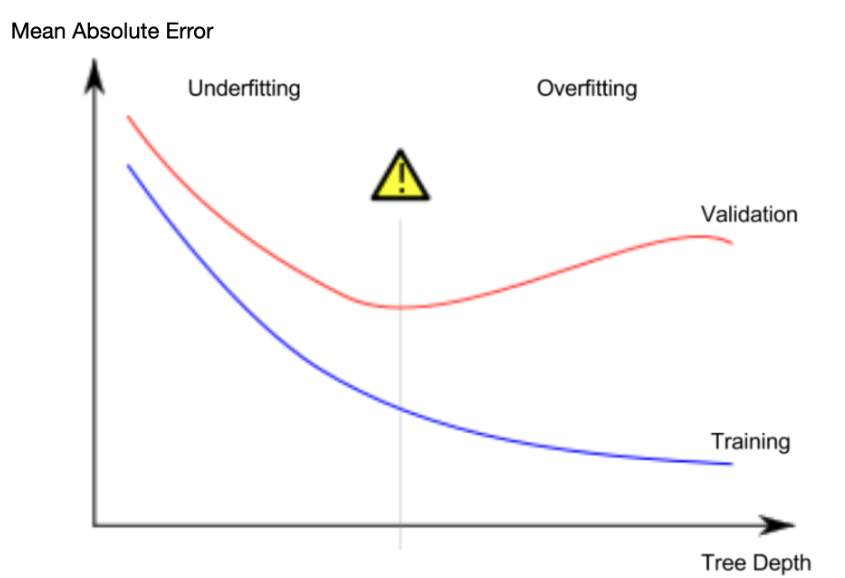
**Preparación de set de prueba y pre-entrenamiento**

1. Una vez obtenido el dataframe puede ser visualizado con ayuda de la extensión wrangler, compilando la fracción de código de main.py que devuelve todo el dataframe completo. La herramienta sirve para analizar columna por columna gráficamente y la estadística correspondiente.

****

****

1. ¿Nos preguntaremos porque hay 49151 columnas? Los modelos de machine learning derivados de la familia sklearn como árboles o bosques, reciben las características separado por columnas para poder distinguir patrones con mayor claridad y asertividad que cuando se empaquetan en una sola palabra o lista. El procesamiento de imágenes requeire datos planos y normalizados, por tanto, la dimensión de pixeles se empaqueta como texto plano en el rango de 0 a 1 para facilitar el procesamiento y el entrenamiento de características. Lo siguiente es asegurarse de la integridad completa del dataset, asegurarse de obtener datos consistentes y no indefinidos, asi como datos sin errores atípicos o desfases.
2. Cuando el dataset ya esta listo para ser ingerido por el modelo, se crea un sistema cíclico representativo que permita generar y estudiar la profundidad del árbol, es decir la calidad de sus predicciones ( que tanto generaliza o que tanto especifica). La calidad de sus respuestas dependerá de la cantidad y objetividad de los datos pero dependerá también de que tanta profundidad o en otras palabras, de cuantas hojas tenga para generar una respuesta. La relación profundidad y error de un árbol aleatorio esta descrito por la siguiente gráfica.



Lo que buscamos es la mayor profundidad a la menor cantidad de error. La evaluación, las métricas y los índices de rendimiento se verán un poco mas abajo.

Para evaluar y sacar la grafica de error-profundidad de nuestro modelo con datos de entrenamiento y prueba se realiza un nuevo archivo llamado “modelo”

*from* sklearn.tree *import* DecisionTreeRegressor

*from* sklearn.model\_selection *import* train\_test\_split

*import* matplotlib.pyplot *as* plt

*from* sklearn.metrics *import* mean\_absolute\_error

*import* pandas *as* pd

*import* joblib

*# def separar\_datos(df):*

*#     y= df.iloc[:,0:4]*

*#     x = df.iloc[:,4:]*

*#     X\_train, X\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.05)*

*#     return X\_train, X\_test, y\_train, y\_test*

def separar\_datos(*df*):

    y\_train = *df*[['coor\_X','coor\_Y','rotacion']]

    x\_train = *df*.iloc[:,4:]

*return* x\_train, y\_train

def get\_mean\_absolute\_error(*mln*,*x\_train*, *y\_train*):

    model = DecisionTreeRegressor(*max\_leaf\_nodes*=*mln*,*random\_state*=1)

    model.fit(*x\_train*, *y\_train*)

    prediction = model.predict(*x\_train*)

*return* mean\_absolute\_error(*y\_train*, prediction)

def graficar\_resultados(*errores*):

    df = pd.DataFrame(*errores*, *columns*=['max\_leaf\_nodes', 'mean\_absolute\_error'])

    df.to\_excel('errores.xlsx', *index*=False,*float\_format*="%.2f",*sheet\_name*='Errores de modelo')

    print(df)

*# x = df['max\_leaf\_nodes']*

*# y = df['mean\_absolute\_error']*

*# plt.plot(x,y)*

*# plt.xlabel('max\_leaf\_nodes')*

*# plt.ylabel('mean\_absolute\_error')*

*# plt.title('Decision Tree Regressor')*

def preentrenamiento(*df*):

    x\_train, y\_train = separar\_datos(*df*)

    errores = []

*for* mln *in* range(5,100,5):

        error = get\_mean\_absolute\_error(mln,x\_train, y\_train)

        errores.append([mln,error])

    graficar\_resultados(errores)

1. Cuando analizamos el comportamiento del árbol a profundidades de entre 5 a 100 hojas, buscamos el índice con el menor error asegurándonos de haber entrenado el modelo tanto con datos de entrenamiento como con datos de prueba. Estos últimos son fundamentales para evaluar el rendimiento de predicción del modelo frente a nuevos datos.
2. Una vez definida la profundidad del modelo, necesitamos entrenar oficialmente el sistema, para eso se realiza una nueva función dentro del archivo “modelo” configurado con la profundidad seleccionada y además, guardando el modelo con ayuda de la librería “joblib”. Guardamos los datos del modelo entrenado en una carpeta vecina a la carpeta “SRC”. Aquí se almacenarán todos nuestros posibles modelos y variantes para luego ser llamados en proyectos de testeo o de inferencia.

def entrenamiento\_final(*mln*,*df*):

    x\_train, y\_train = separar\_datos(*df*)

    print(x\_train)

    model = DecisionTreeRegressor(*max\_leaf\_nodes*=*mln*,*random\_state*=1)

    model.fit(x\_train, y\_train)

    prediction = model.predict(x\_train)

    error = mean\_absolute\_error(y\_train, prediction)

    joblib.dump(model, 'models/modelo\_entrenado.pkl')

    print('Modelo entrenado con un error de:', error)

**“MAIN.PY”**

*import* os

*from* src.preprocess *import* procesar\_imagen, crear\_dataframe

*from* src.modelo *import* preentrenamiento, entrenamiento\_final

*from* src.test *import* test\_model

*from* src.data\_augmentation *import* data\_augmentation

*import* numpy *as* np

*import* pandas *as* pd

dir\_processed = 'data/processed'

dir\_raw = 'data/raw'

dir\_labels = 'data/labels/images.csv'

dir\_test = 'data/test'

dir\_raw\_augmentation = 'data/raw\_augmentation'

*if* not os.path.exists(dir\_processed):

    os.makedirs(dir\_processed)

procesar\_imagen(dir\_raw, dir\_processed)

df= crear\_dataframe(dir\_processed,dir\_labels)

print(df)

*# entrenamiento parcial para encontrar la mejor profundidad del arbol*

preentrenamiento(df) *#comentar en produccion*

*# entrenamiento final con la mejor profundidad*

entrenamiento\_final(20,df)

**Inferencia local de modelo**

1. Para consumir el modelo es preciso hacer el llamado al modelo usando la librería joblib. Para esto creamos un nuevo archivo dedicado a la inferencia y prueba de modelo, “Test”. Adicional al archivo de ejecución del modelo, solo para efectos prácticos se crea una carpeta en data/test para extraer las imágenes que se desean predecir.

*import* joblib

*import* os

*import* cv2

*import* numpy *as* np

*import* pandas *as* pd

def test\_model(*dir\_test*):

    archivo = os.listdir(*dir\_test*)

*if* not archivo:

*raise* FileNotFoundError(f'No se encontraron archivos en el gestor de muestras')

*if* not archivo[0].endswith('.jpg'):

*raise* FileExistsError(f'El archivo {archivo[0]} no es una imagen en formato jpg')

    path = os.path.join(*dir\_test*, archivo[0])

    imagen = cv2.imread(path)

    imagen = cv2.resize(imagen, (128, 128))

    imagen = imagen.flatten()

    imagen = imagen/255.0

    feature =[]

    feature.append(imagen)

    feature = np.array(feature)

    columnas\_pixeles = [f'pixel\_{i}' *for* i *in* range(feature.shape[1])]

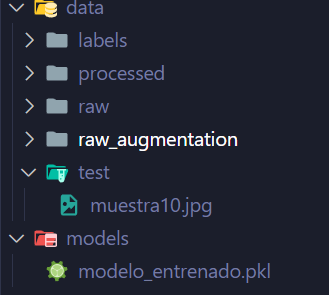
    x\_test = pd.DataFrame(feature, *columns*=columnas\_pixeles)

    model = joblib.load('models/modelo\_entrenado.pkl')

    prediccion = model.predict(x\_test)

*return* prediccion

finalmente, el sistema de archivos queda así:

****

**Data Augmentation**

Aumentar los datos es vital si requerimos un modelo bien generalizado y eficiente, para obtener un dataset rico en características, se debe recopilar todas las posibles variantes en formatos de imagen y en formatos descriptivos de extensión csv o lo que llamamos labels (etiquetas o salidas). Para hacer la aumentación de los datos, hay muchas técnicas usando librerías de extracción y manipulación de datos gráficos, en este caso, se usará opencv para generar imágenes a partir de capas, generando ubicaciones y rotaciones consecutivas. En otras palabras, se separa la figura en una capa, se separa el fondo o plano vacío en otra capa diferente para luego apilarse, configurarse de forma distinta y guardándose por cada iteración.

El siguiente código puede procesar una imagen y separar de ella el cuadro amarillo, el cual se usara para rotar y ubicar en todas las posibles posiciones.  
asi como puede identificarse del plano vacío con el contorno en rojo que encierra la zona real de trabajo.

def data\_augmentation(*dir\_raw\_augmentation*,*dir\_raw*):

*if* not os.path.exists(*dir\_raw\_augmentation*):

        os.makedirs(*dir\_raw\_augmentation*)

*##SACAR LA FIGURA*

*# 1) definir el rango de tonos que se desea filtrar en la imagen*

    tono\_bajo = np.array([20,100,100]) *#tonos amarillos con la menor intencidad y brillo bajo de amarillo*

    tono\_alto = np.array([30,255,255]) *#tonos amarillos con la mayor intencidad de luz alta*

*# 2) convertir la imagen en un espacio de color HSV para poder ser filtrada, y obtener la altura y ancho de la imagen en pixeles*

    imagen\_con\_mi\_cuadrado = cv2.imread(f'{*dir\_raw*}/muestra0.jpg')

*# high, widht = imagen\_con\_mi\_cuadrado.shape[:2]*

    hsv = cv2.cvtColor(imagen\_con\_mi\_cuadrado, cv2.COLOR\_BGR2HSV)

*# 3) crea una mascara para filtrar los colores en el rango definido*

    mask = cv2.inRange(hsv, tono\_bajo, tono\_alto)

*# 4) aplica la mascara a la imagen original*

    contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE) *#encuentra los contornos de la mascara*

    c = max(contours, *key*=cv2.contourArea) *#selecciona el contorno mas grande*

    x\_shape, y\_shape, w\_shape, h\_shape = cv2.boundingRect(c) *#crea el rectangulo mas pequeño alrededor del contorno*

    cuadro\_amarillo = imagen\_con\_mi\_cuadrado[y\_shape:y\_shape + h\_shape, x\_shape:x\_shape + w\_shape] *#recorta la imagen original con el rectangulo mas pequeño*

    cv2.imwrite(os.path.join(*dir\_raw\_augmentation*, 'cuadro\_amarillo.jpg'), cuadro\_amarillo) *#guarda la imagen recortada en la carpeta de raw\_augmentation*

*#SACAR EL PLANO*

    tono\_rojo\_bajo1 = np.array([0, 100, 100])

    tono\_rojo\_alto1 = np.array([10, 255, 255])

    tono\_rojo\_bajo2 = np.array([160, 100, 100])

    tono\_rojo\_alto2 = np.array([180, 255, 255])

*#convertir la imagen a HSV*

    imagen\_del\_plano = cv2.imread(f'{*dir\_raw*}/plano\_vacio.jpg')

    hsv\_plane = cv2.cvtColor(imagen\_del\_plano, cv2.COLOR\_BGR2HSV)

*#crear la mascara con los colores rojos*

    mask\_plane\_1 = cv2.inRange(hsv\_plane, tono\_rojo\_bajo1, tono\_rojo\_alto1)

    mask\_plane\_2 = cv2.inRange(hsv\_plane, tono\_rojo\_bajo2, tono\_rojo\_alto2)

*#generar una mascara general, entre las dos mascaras de rojo*

    mask\_plane = cv2.bitwise\_or(mask\_plane\_1, mask\_plane\_2)

*#encontrar el contorno de la  imagen mas grande*

    contours\_plane, \_ = cv2.findContours(mask\_plane, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    c\_plane = max(contours\_plane, *key*=cv2.contourArea)

*#se crea una mascara de ceros con el mismo tamaño de la mascara de los colores rojos*

    mask\_internal = np.zeros\_like(mask\_plane)

    cv2.drawContours(mask\_internal, [c\_plane], -1, (255), *thickness*=cv2.FILLED)

    plano\_vacio = cv2.bitwise\_and(imagen\_del\_plano, imagen\_del\_plano, *mask*=mask\_internal)

    x\_plane, y\_plane, w\_plane, h\_plane = cv2.boundingRect(c\_plane)

    plano\_vacio\_cutted = imagen\_del\_plano[y\_plane:y\_plane + h\_plane, x\_plane:x\_plane + w\_plane]

    cv2.imwrite(os.path.join(*dir\_raw\_augmentation*, 'plano\_vacio.jpg'), plano\_vacio\_cutted)

    high\_plano, width\_plano = plano\_vacio\_cutted.shape[:2]

    print(f'alto: {high\_plano}, ancho: {width\_plano}')

**Paso 1: Cargar los datos**

1. **Cargar las imágenes y etiquetas**: Carga tanto los archivos .npy de las imágenes como los datos de etiquetas (coordenadas y ángulos) en una estructura que permita su uso como datos de entrada y salida.
2. **Normalizar las imágenes**: Si no lo has hecho ya, normaliza las imágenes (por ejemplo, escalando los valores de píxeles entre 0 y 1).

**Paso 2: Preparar los datos para entrenamiento y prueba**

1. **Dividir en entrenamiento y prueba**: Divide el conjunto de datos en entrenamiento y prueba (e.g., 80% entrenamiento y 20% prueba).
2. **Convertir en tensores o arreglos compatibles**: Asegúrate de que las imágenes y etiquetas están en un formato compatible con el modelo (usualmente numpy para modelos con scikit-learn).

**Paso 3: Configurar el modelo**

1. **Elegir el modelo**: Usarás un modelo de árbol de decisión o un bosque aleatorio profundo.
2. **Definir el bosque aleatorio profundo**: Puedes empezar con un modelo como RandomForestRegressor en scikit-learn y ajustar los hiperparámetros para obtener una estructura profunda.

**Paso 4: Entrenar el modelo**

1. **Entrenar el modelo con las imágenes y etiquetas**: Proporciona las imágenes y etiquetas para que el modelo aprenda a mapear las características visuales a las coordenadas y el ángulo de rotación.
2. **Evaluar el rendimiento**: Almacena las métricas de rendimiento para evaluar qué tan bien predice el modelo.

**Paso 5: Validar y ajustar el modelo**

1. **Validación cruzada**: Utiliza validación cruzada para verificar la consistencia del modelo y prevenir el sobreajuste.
2. **Ajuste de hiperparámetros**: Considera ajustar los hiperparámetros (número de árboles, profundidad máxima) para mejorar el rendimiento.

**Paso 6: Prueba y predicción**

1. **Probar con nuevas imágenes**: Una vez que esté entrenado, prueba con imágenes nuevas para ver cómo predice las coordenadas y el ángulo.
2. **Analizar los resultados**: Evalúa si el modelo predice correctamente o si necesita más ajuste o datos adicionales.