



UNIVERSIDAD COMPLUTENSE MADRID

NTIC Master

Facultad de Comercio y Turismo

Máster en Data Science, Big Data y Business Analytics

Desarrollo de un Sistema Basado en Visión por Computadora para la Detección y
Clasificación de Residuos No Orgánicos en Playas

Autores:

Hao Qi Xu

Jesús Esteban López Peña

Carlos Fernando Fernández de Castro Bautista

Erick Julián Villamizar Contreras

Diego Alejandro Torres Cubillos

Juan Diego Uribe Bolaños

Madrid a 19 de septiembre de 2024

Índice

1. Problemática:	1
1.1. La generación y gestión de residuos en el mundo.....	1
1.2. Basuras en las playas.....	2
1.3. Identificación y clasificación de basuras.....	3
2. Estado del Arte.....	4
3. Objetivos.....	7
3.1. Objetivo Principal.....	7
3.2. Objetivos Específicos.....	7
4. Metodología y Desarrollo.....	7
4.1. DataSets.....	7
Tabla 1.....	9
4.2. Entrenamiento.....	9
Tabla 2.....	10
Tabla 3.....	11
Tabla 4.....	12
5. Resultados:.....	13
Tabla 5.....	14
6.2. Utilidad.....	15
6.3. Próximos pasos.....	17
7. Bibliografía.....	18
8. Anexos.....	19
8.1. Anexo A: Imágenes de Datasets.....	19
8.2. Anexo B: Gráficos de Evaluación.....	21
8.3. Anexo C: Evaluación en Imágenes de Prueba.....	23
8.4. Anexo D: Código.....	26

1. Problemática:

1.1. La generación y gestión de residuos en el mundo

El manejo de residuos ha presentado un gran reto para la humanidad actualmente, tal como dijo Sameh Wahba, director de Desarrollo Urbano y Territorial, Gestión de Riesgos de Desastres, y Resiliencia del Banco Mundial “La gestión inadecuada de los desechos está produciendo la contaminación de los océanos del mundo, obstruyendo los drenajes y causando inundaciones, transmitiendo enfermedades, aumentando las afecciones respiratorias por causa de la quema, perjudicando a los animales que consumen desperdicios, y afectando el desarrollo económico, por ejemplo, al perjudicar el turismo” (Wahba S. 2018)¹. Es por tanto la gestión de residuos en los entornos naturales una problemática que abarca esferas más allá del medio ambiente, afectando también a los sectores de la salud pública y la economía en sí misma sustentada a través del turismo y la experiencia de las personas con respecto a su entorno.

Tal y como se relata en el informe What a Waste 2.0² (Los desechos 2.0) del Banco Mundial, son más de 2010 toneladas de desechos sólidos los que se generan en el mundo anualmente y al menos el 33 % de ellos no tiene una gestión adecuada para la preservación del medio ambiente (Banco Mundial. 2018). En este mismo informe se proyecta como la rápida urbanización y el crecimiento exponencial de la población en las distintas zonas, tanto centrales como turísticas del mundo, en conjunto con el desarrollo económico esperado, generarán un aumento del 70% en la cantidad de desechos a nivel mundial para los próximos 30 años pudiendo llegar a un volumen de 3400 millones de toneladas de desechos generados anualmente.

¹ Artículo publicado en la página del Banco Mundial en el 2018, titulado Los desechos: un análisis actualizado del futuro de la gestión de los desechos sólidos, disponible en: <https://www.bancomundial.org/es/news/immersive-story/2018/09/20/what-a-waste-an-updated-look-into-the-future-of-solid-waste-management>

² What a Waste es el informe sobre el que se construyó el artículo del Banco Mundial y sobre el que se sustentan todos los datos de este título “La Generación y Gestión de Residuos en el Mundo”. El informe se encuentra disponible en: https://datatopics.worldbank.org/what-a-waste/?_gl=1*1dfmkny*_gcl_au*MzIxNTU2OTExLjE3MjY0MTY0NjY

Así mismo, el Banco Mundial destaca que el 23% de los desechos generados anualmente pertenecen a la región de Asia oriental y el Pacífico, así como la tercera parte del total de desechos, el 34%, son generados en conjunto en los países con mayores ingresos, a pesar de representar solo el 16% de la población mundial. Aun con ello, se proyecta que en los países de ingreso mediano/bajo se registre el mayor aumento en la generación de desechos sólidos, debido al crecimiento poblacional y económico que se ha estimado en los mismos.

En materia de recolección de los residuos generados, el Banco Mundial añade que los países de ingreso medio/alto son los que proveen la mayoría de los servicios a nivel mundial, igualmente más de la tercera parte de los residuos generados por los países de altos ingresos son recuperados a través de estos procesos de reciclaje. En los países con menores ingresos el 48% los desechos generados en las ciudades son recogidos, mientras que en sus zonas rurales solo se recoge el 26% y de todo esto solo el 4% es reciclado a nivel nacional, de hecho, en general a nivel mundial solo el 13,5% de los desechos son reciclados y de estos el 5,5% es compostado.

1.2. Basuras en las playas

Cuando analizamos el contexto específico de las playas, nos encontramos con que la situación de desechos sólidos en las mismas están estrechamente relacionadas con la contaminación y presencia de diferentes tipos de desechos en el mar. Según el estudio de la ONU from pollution to solution³, los residuos no orgánicos que no son desechados adecuadamente pueden terminar tarde o temprano en el mar y luego en las diferentes playas del mundo, entre los cuales el más abundante y problemático hasta la fecha ha sido el plástico, el cual representa el 85% del total de los desechos marinos (ONU, 2021), Los cuales concluyen en un daño irreparable a las fuentes hídricas y la biodiversidad de los diferentes sectores colindantes a las zonas acuíferas a nivel mundial.

³ From pollution to solution es un informe de la ONU publicado en el 2021, el cual describe la situación de los residuos y desechos sólidos en el océano. la información de este artículo se encuentra disponible en: <https://www.unep.org/interactives/pollution-to-solution/>

En materia de costos, en un estudio de National Geographic⁴ se estimó que “el coste de limpiar las playas para los gobiernos de 90 localidades de Washington, Oregón y California fue de más de 500 millones de dólares. Pero las playas limpias aportan beneficios económicos a las comunidades locales en forma de turismo” (National Geographic, 2018). Es de hecho en este estudio que se cita como un 66% de personas encuestadas, las cuales frecuentaban las zonas costeras de los sitios mencionados, afirmaban que la ausencia de basuras y la calidad del aire era muy importante a la hora de decidir si visitar o no una playa.

Según un estudio internacional liderado por el Instituto de Ciencia y Tecnología Ambiental de la Universidad Autónoma de Barcelona (ICTA-UAB) el uso recreativo por parte del turismo en las playas del Mediterraneo genera hasta el 80% de los residuos marinos durante el verano, así mismo, este estudio concluye que la mayor parte de los desechos son microplásticos que se generan por la fragmentación de desechos de mayor tamaño de este mismo material, el cual como hemos visto antes, ocupa la mayor cantidad de los desechos sólidos en las playas, así mismo aseguran que “El 65 % de la basura marina que se acumula en las playas está formada por colillas, pajitas o latas de bebidas y un 15 % más son microplásticos de mayor tamaño, ya que los artículos de plástico abandonados se fragmentan por la irradiación solar y la fricción con la arena, acelerada por el alto volumen de visitantes.”⁵ (ICTA-UAB, 2021).

1.3. Identificación y clasificación de basuras

Dado que el tiempo de descomposición de los residuos varía según sus características, y el uso de productos químicos para acelerar este proceso podría generar un mayor impacto negativo en el medio ambiente, la solución más sostenible radica en su recolección,

⁴ Esto fue publicado en un artículo de la mencionada revista escrito por la periodista Laura Parker el cual se encuentra disponible en: <https://www.nationalgeographic.es/medio-ambiente/2018/10/estudio-limpiezas-playas-muestra-alcance-global-contaminacion-plastico>

⁵ El informe directamente del que habla el artículo no se encuentra disponible, por lo que la información citada es únicamente la que se menciona en el mismo artículo que se encuentra disponible en: <https://www.lavanguardia.com/natural/20210208/6231172/turismo-genera-basura-playas-mediterraneo.html>

clasificación y reciclaje. Este enfoque optimiza la gestión de residuos, contribuyendo a reducir su impacto ambiental y promoviendo un ciclo de vida más eficiente de los materiales.

La diversidad de los materiales, las limitaciones tecnológicas y la falta de infraestructura adecuada son factores que juegan un papel importante en el proceso de reciclaje de los residuos. Estos desafíos no solo afectan la eficiencia en el proceso de reciclaje, sino que también impactan a la sostenibilidad ambiental y a la gestión de recursos disponibles. Esto puede implicar que:

- La presencia de una amplia gama de materiales en una zona específica dificulta la separación eficiente y su posterior reciclaje, provocando que los centros especializados se vean obligados a implementar sistemas de clasificación de materiales dentro de sus instalaciones.
- La falta de tecnología avanzada para la clasificación y separación de los residuos puede resultar en una menor tasa de reciclaje ya que una gran mayoría de los residuos podrían ser enviados a vertederos siendo eventualmente incinerados generando un impacto negativo en el medio ambiente.
- La ausencia de una infraestructura adecuada para la recolección y clasificación conlleva una inadecuada clasificación de los residuos, dificultando el proceso de reciclaje.

2. Estado del Arte

En una primera etapa de la investigación, se realizó una revisión de las empresas y/o asociaciones que tuvieran una propuesta similar en materia de identificación y clasificación de residuos, en este caso en procesos industriales de distribución y clasificación de los residuos entre las que se destacaron Recycleye y Atria.

RECYCLEYE⁶ es una empresa dedicada al desarrollo e implementación de tecnologías para el reciclaje. En esta nota hablan específicamente de dos tecnologías, Recycle Vision y Recycle Robotics. La primera se basa en un software para la detección y clasificación de residuos a través de la visión por computadora la cual proponen implementar específicamente en los sectores industriales del reciclaje, la segunda se basa en las máquinas físicas que apoyan la separación y distribución de los residuos para facilitar su proceso de reciclaje. Si bien estos productos los pueden ofrecer por separado, también enfatizan el los beneficios que tienen el implementar ambos para un mismo proceso. Este proyecto nos ha permitido ver que en la industria la visión por computadora si es una práctica adoptada y aceptada por varias entidades, lo que permite hacerse una idea de lo que se viene haciendo en el sector industrial frente a nuestro trabajo.

En una dirección similar, Atria⁷ En materia de clasificación y separación de residuos en la industria del reciclaje destaca en específico dos proyectos SIARA y SEPARA. El primero es la clasificación de residuos que van por la cinta, en la fábrica de reciclaje, centrándose primordialmente en plásticos de gran tamaño y residuos como films de plástico o barquillas usadas en el transporte de las frutas, aunque también buscaban localizar residuos de cartón y electrónicos, por lo que más que una clasificación general de los residuos, hacían una búsqueda de residuos específicos para separarlos. El segundo proyecto parece una progresión del primero, pues se basa en la misma detección y clasificación de residuos sobre la cinta, pero en este caso utilizan más tecnologías para la detección de los mismos, como lo son las cámaras 3D y cámaras de visión hiperespectral, para así poder capturar objetos de menor tamaño en la imagen, en síntesis obtener más datos de la cámara para usarlos en el modelo.

Otros trabajos que han resultado interesantes a la hora de definir la dirección del presente proyecto, han sido desarrollados a manos de grupos de investigación y desarrollo de

⁶ Empresa dedicada a formular y diseñar soluciones para los procesos industriales de reciclaje, para mas información visitar su pagina web: <https://recycleye.com/es/la-ia-y-el-reconocimiento-des-residuos-por-que-funciona-tan-bien/>

⁷ Atria también esta dedicada al desarrollo de soluciones y construcción de herramientas orientadas a la optimización de los procesos industriales de reciclaje. para mas yor información visitar su pagina web: <https://atriainnovation.com/servicios/sostenibilidad/clasificacion-residuos/>

herramientas focalizadas al reciclaje, la limpieza del medio ambiente y los ODS. En el 2016 el grupo de investigación IEAB'S Artificial Vision Group⁸ de la Universidad Nacional de Colombia desarrolló un prototipo automático que captura imágenes de las basuras con una cámara web, las procesa y las clasifica, para así indicar el lugar donde se debe depositar cada desecho. Sumado a ello, el prototipo tiene integrado un dispositivo de audio con interfaz visual para que la persona que lo use entienda el proceso de clasificación.

Esta propuesta estuvo entre las 21 iniciativas reconocidas por la OEA en la competencia Intel ISEF 2016, en Phoenix Arizona, en donde el proyecto colombiano ‘Automatic Classification of Waste Using Artificial Vision and Programmed Electronic’ (Desarrollo de un prototipo electrónico que permite la clasificación automática de residuos) obtuvo el segundo lugar en la categoría ‘Desarrollo integral de las Américas’

Por su parte, en el 2020 surgió como propuesta del Equipo Dorado⁹ el proyecto RECICLA.IA, en el marco de la edición LATAM de Saturday AI del mencionado año, el cual consistió en el desarrollo de una API para la clasificación de residuos en la Industria del reciclaje. Al igual que las empresas descritas anteriormente se basa en Visión Artificial para la detección y clasificación de los residuos y un brazo robótico para la separación de los mismos, aunque el proyecto en sí mismo se centra en el desarrollo de la API. Para este modelo se evaluaron previamente dos modelos ya entrenados que fueron el VGG-16 y el InceptionRestNetV2 pero finalmente optaron por utilizar el DenseNet121 siendo este el que mejores resultados arrojó en la evaluación de modelos.

Para el entrenamiento utilizaron 2 datasets uno de 2,527 imágenes distribuidas en 501 imágenes para vidrio, 594 para papel, 403 para cartón, 482 para plástico, 410 para metal y 137 como relleno sanitario; y el otro de 22,500 imágenes, clasificadas en dos grupos:

⁸ Este grupo de investigación específicamente opera en la sede de Medellín de la Universidad Nacional de Colombia, su enfoque se encuentra en el desarrollo de soluciones para el medio ambiente y los objetivos de desarrollo sostenible a través de la ingeniería. para mas informacion consultar: <https://iresiduo.com/noticias/colombia/universidad-nacional-colombia/16/05/30/vision-artificial-y-algoritmos-clasificar>

⁹ El Equipo Dorado es un grupo conformado principalmente para participar en la edición LATAM de Saturday AI del 2020, para la cual desarrollaron una API de reconocimiento de residuos sólidos basado en visión artificial. la informacion sobre su proyecto se puede encontrar en: <https://medium.com/saturdays-ai/recicla-ia-sistema-para-clasificaci%C3%B3n-de-residuos-s%C3%B3lidos-urbanos-e-implementaci%C3%B3n-con-sistema-bdf648809025>

orgánicas y reciclables, para lo cual reclasificaron dichas imágenes en sus propias categorías para poder usarlas en el entrenamiento. Finalmente el set de datos utilizado se dividió en 7 clases diferentes: Vidrio, papel, cartón, plástico, metal, orgánico y relleno sanitario, obteniendo un total de 1,000 imágenes por clase.

3. Objetivos

3.1. Objetivo Principal

- Diseñar una herramienta que identifique y clasifique desechos sólidos en entornos naturales.

3.2. Objetivos Específicos

- Encontrar y seleccionar Datasets ya construidos con base en imágenes de desechos previamente etiquetados.
- Definir categorías apropiadas para asignar a los desechos de forma eficiente.
- Seleccionar y recategorizar las imágenes de los Datasets elegidos de forma tal que coincidan con las categorías seleccionadas.
- Seleccionar los parámetros apropiados para entrenar el modelo clasificador de desechos en entornos naturales.

4. Metodología y Desarrollo

4.1. DataSets

Se han utilizado, para entrenar el modelo, un total de 4,157 imágenes provenientes de diferentes fuentes:

TACO (Trash Annotations in Context): conjunto de datos diseñado para la detección de basura en imágenes que muestran diversos contextos, como calles, parques y áreas naturales, aunque con pocas imágenes de playas. De 6004 imágenes originales, se

seleccionaron 1613, ya que la mayoría eran variaciones de la misma imagen (técnicas de data augmentation) para conservar sólo las imágenes únicas. Este conjunto de datos es el único de los cuatro que incluía archivos de texto con anotaciones detalladas de las basuras en cada imagen. Cada imagen está etiquetada con la posición exacta mediante cajas delimitadoras (bounding boxes) y clasificada en 18 categorías distintas. Estas categorías fueron posteriormente reducidas a las 4 del interés del proyecto (plastic, paper/cardboard, glass, metal).

- Plastic: plastic bag - wrapper, plastic container, styrofoam piece, straw, lid, cup, bottle, bottle cap, other plastic. (Plástico)
- Cardboard: paper, carton. (Papel y cartón)
- Glass: broken glass, glass bottles. (Vidrio)
- Metal: aluminum foil, cans. (Metal)

Open Litter Map: plataforma global abierta que permite a los usuarios cargar imágenes y registrar la ubicación de residuos en diversas áreas. Los datos incluyen una amplia variedad de basura, desde plásticos y colillas de cigarrillos hasta envoltorios, botellas y latas, en contextos urbanos y rurales. Se realizó una selección manual de 1,892 imágenes de residuos en diferentes entornos pero que en su mayoría eran imágenes de desechos sólidos en la playa.

Google Images: Se recopilaron un total de 151 imágenes de basura en playas, todas con licencias de "Creative Commons" seleccionadas a través de búsquedas filtradas. Se decidió orientar la búsqueda en encontrar imágenes que muestran grandes cantidades de basura amontonada y solapada en diversos tipos de playas: desde arenas rocosas hasta arenas finas de diferentes colores. Esto se debe a que las imágenes en Open Litter Map suelen mostrar basura en cantidades pequeñas o aisladas. El objetivo era incluir una variedad más amplia de imágenes de basura en playas para mejorar la capacidad del modelo de detección y aumentar su precisión.

Glass Images for Waste Segregation: conjunto de datos que incluye principalmente botellas de vidrio en diversos ángulos y tamaños. Se incorporó este conjunto de datos para aumentar

el número de instancias de residuos de vidrio en playas, específicamente botellas de vidrio, el conjunto total de datos. El objetivo era equilibrar las clases durante el entrenamiento del modelo, mejorando así su capacidad para identificar y clasificar estos residuos de manera más precisa.

Los conjuntos de datos (2.), (3.) y (4.), no incluyen archivos con las etiquetas de las coordenadas de las cajas delimitadoras ni las categorías de los residuos detectados en cada imagen. Estos datos fueron etiquetados manualmente por el equipo utilizando la plataforma CVAT (Computer Vision Annotation Tool).

Se pueden visualizar muestras de imágenes de cada conjunto de datos en "Anexo A: Imágenes de Datasets".

DataSet	Nº Imágenes utilizadas	Basura en Playas	Etiquetado	Licencia	Link
(1.) Taco Dataset Yolo	1613	No	Si	CC BY 4.0	www.kaggle.com/datasets/vercelanz09/taco-dataset-yolo-format
(2.) Open Litter Map	1892	Si	No	CC0	https://openlittermap.com/about
(3.) Google Images	151	Si	No	CC	https://www.google.com
(4.) Glass images for waste segregation	501	No	No	Unknown	https://www.kaggle.com/datasets/nandinibagga/glass-images-for-waste-segregation

Tabla 1

4.2. Entrenamiento

Se decidió utilizar YOLO como modelo para la detección y clasificación de residuos en playas debido a sus numerosas ventajas, como su alta velocidad de inferencia y su capacidad para detectar múltiples objetos en tiempo real. En particular, se emplearon los modelos de YOLO versión 8 en sus tamaños n, s y m durante el entrenamiento. Se tomó la decisión de no utilizar versiones de mayor tamaño debido al elevado costo computacional

asociado y la mínima mejora en la métrica mAP. En detalle, el aumento en mAP de YOLOv8s a YOLOv8m es de 5.3 puntos (de 44.9 a 50.2), mientras que el aumento de YOLOv8m a YOLOv8l es de solo 2.7 puntos (de 50.2 a 52.9)

Model	size (pixels)	mAP^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
<u>YOLOv8n</u>	640	37.3	80.4	0.99	3.2	8.7
<u>YOLOv8s</u>	640	44.9	128.4	1.20	11.2	28.6
<u>YOLOv8m</u>	640	50.2	234.7	1.83	25.9	78.9
<u>YOLOv8l</u>	640	52.9	375.2	2.39	43.7	165.2
<u>YOLOv8x</u>	640	53.9	479.1	3.53	68.2	257.8

Tabla 2¹⁰

El enfoque fue basado en entrenar modelos de manera progresiva con un número creciente de imágenes mientras se recolectaban más datos monitoreando los resultados y mejoras de estos modelos hasta alcanzar un rendimiento decente. Dado que, aparte del conjunto de datos (1), los tres conjuntos restantes no están etiquetados y el etiquetado manual de imágenes es muy laborioso, se crearon cuatro conjuntos de datos de entrenamiento. Cada conjunto tiene más imágenes que el anterior, salvo el último, "Conjunto 4", del cual se han eliminado algunas imágenes para balancear la frecuencia de las categorías de residuos.

¹⁰ Esta tabla se encuentra en el manual para el modelo YOLO la cual se encuentra disponible en el source: <https://github.com/ultralytics/ultralytics>

Conjunto de Entrenamiento	Dataset(s)	Nº imagenes	Nº cardboard	Nº plastic	Nº galss	Nº metal	Nº Total
Conjunto 1	(1)	1613	494 (14.52%)	2210 (64.96%)	241 (7.10%)	457 (13.53%)	3402 (100%)
Conjunto 2	(1)+(2)+(3)	2224	780 (12.10%)	3884 (66.37%)	484 (8.27%)	776 (13.26%)	5852 (100%)
Conjunto 3	(1)+(2)+(3) +(4)	4157	1371 (16.56%)	3874 (46.79%)	1603 (19.36%)	1432 (17.29%)	8280 (100%)
Conjunto 4	(1)+(2)+(3) +(4)	3294	1367 (20.18%)	2373 (35.03%)	1602 (23.65%)	1432 (21.15%)	6774 (100%)

Tabla 3

Por ejemplo, el "Conjunto 1" está compuesto únicamente por el dataset (1), que incluye 1,613 imágenes de basura, de las cuales solo una pequeña parte muestra residuos en contexto de playa. En estas 1,613 imágenes, hay un total de 3,402 instancias individuales etiquetadas. La mayoría de estas instancias son plásticos, con 2,210 (64.96%), seguidos por cartón con 14.52%, metal con 13.53% y vidrio con 7.10%.

En el "Conjunto 2", se añadieron 611 nuevas imágenes de los datasets (2) y (3), centradas exclusivamente en residuos en contextos de playa, aumentando el número total de instancias de 3,402 a 5,852. En este conjunto, el plástico sigue siendo la categoría predominante, representando aproximadamente el 66% del total de residuos. La categoría con menos representación sigue siendo el vidrio, con un 8%. Por esta razón, en el "Conjunto 3" se incorporó el dataset (4), que incluye imágenes exclusivamente de botellas de vidrio, elevando la frecuencia relativa de residuos de vidrio al 19%.

Finalmente, el "Conjunto 4" difiere de los anteriores en que, en lugar de añadir más imágenes, se ha utilizado una técnica de submuestreo para reducir la cantidad de la categoría predominante, "plástico". Se eliminaron 863 imágenes, principalmente con residuos de plástico, del "Conjunto 3", reduciendo el total a 3,294 imágenes. Esta reducción tuvo como objetivo equilibrar la frecuencia relativa de las categorías de residuos.

Como resultado, las instancias de plástico se redujeron de 3,874 a 2,373, disminuyendo su frecuencia relativa del 47% al 35%.

Modelo	Conjunto de Entrenamiento	Version de YOLO	Epochs	Patience	Dropout	Augment	Seed
Modelo 1	Conjunto 1	yolov8n	100	15	0.3	True	0
Modelo 2	Conjunto 2	yolov8m	100	10	0.5	True	0
Modelo3	Conjunto 3	yolov8m	100	10	0.5	True	0
Modelo 4	Conjunto 4	yolov8m	200	15	0.3	True	0

Tabla 4

Esta tabla muestra los mejores modelos seleccionados junto con sus respectivas configuraciones y parámetros para cada conjunto de entrenamiento.

La biblioteca Ultralytics permite importar modelos de YOLO y realizar un Fine-Tuning con datos personalizados mediante una sola línea de código utilizando el método “.train()” del modelo YOLO correspondiente. Este método permite personalizar el proceso de entrenamiento mediante la modificación de ciertos parámetros.

En particular, se han modificado manualmente 5 parámetros en la tabla: “epochs”, “patience”, “dropout”, “augment” y “seed”. El resto de los parámetros utilizan los valores por defecto proporcionados por el método “.train()”:

Durante el entrenamiento del modelo, se utilizó un número determinado de *epochs*, que corresponde a la cantidad de veces que el modelo recorre todo el conjunto de datos. Además, se implementó una técnica llamada *patience*, que define cuántas épocas el modelo puede esperar sin mejoras en la métrica de validación antes de detenerse. Para evitar el sobreajuste, se aplicó *dropout*, desactivando un porcentaje de neuronas en cada capa, aunque en este caso el valor por defecto fue 0. Se usaron también técnicas de aumento de datos (*augment*) para generar variaciones en las imágenes de entrenamiento, siguiendo los parámetros predefinidos de YOLO. Finalmente, se estableció una semilla de 0 (*seed*) para garantizar que los resultados sean reproducibles.

5. Resultados:

Durante el entrenamiento del modelo YOLO, se genera automáticamente una carpeta denominada "runs", donde se almacenan todos los resultados del entrenamiento. Esta carpeta incluye un archivo CSV llamado "results.csv", que contiene todas las métricas por cada época de entrenamiento, así como imágenes de gráficos, como la matriz de confusión y las curvas de Precisión y Recall, entre otros. Los archivos de resultados correspondientes al modelo 3 pueden ser consultados en "Anexo B: Gráficos de Evaluación".

En la evaluación de los modelos, se tuvo en cuenta en primera medida el AP(50), una de las métricas más utilizadas en tareas de detección de objetos a través de visión por computadora, cuyo nombre se refiere a la Precisión Promedio (Average Precision), con un 50% de Intersección sobre Unión (IoU), métrica con la cual evaluamos la precisión de los modelos de detección utilizando un único umbral, en el que se considera correcta una predicción si la caja delimitadora predicha tiene al menos, en este caso, un 50% de superposición con la caja real (ground truth). El AP al ser calculado como el área bajo la curva de Precisión-Recall, nos proporciona una medida general de qué tan bien el modelo detecta objetos de una clase o categoría específica a ese nivel de IoU, por lo que un valor de AP(50) más alto indica un mejor rendimiento en la detección, lo que significa que el modelo identifica correctamente los objetos con una superposición suficiente. Por su parte, mAP(50), o Precisión Promedio Media al 50% de IoU, es el promedio del AP(50) calculado para todas las clases del modelo, lo que refleja el rendimiento global en la detección de objetos a ese umbral del 50% en todas las categorías.

Antes de comentar los resultados de los cuatro modelos entrenados, cabe destacar que el Modelo 1 es el único que se ha entrenado con cinco clases (paper, plastic, glass, metal, other), a diferencia de los siguientes modelos, con los que se utilizaron solo cuatro clases (cardboard, plastic, glass, metal), debido al mal rendimiento observado en la detección de la clase "other" por la gran variedad de imágenes diferentes que eran etiquetadas en esta categoría, como se mencionó anteriormente. Además, la nomenclatura para "paper" se cambió a "cardboard" en los modelos posteriores al Modelo 1, para mayor precisión en los términos.

Modelo	Last epoch	mAP (50)	AP(50) paper/cardboard	AP(50) plastic	AP(50) glass	AP(50) metal	AP(50) other
Modelo 1	97	0.164	0.182	0.353	0.077	0.173	0.035
Modelo 2	59	0.261	0.136	0.413	0.212	0.284	X
Modelo3	100	0.506	0.498	0.407	0.619	0.499	X
Modelo 4	74	0.516	0.518	0.280	0.745	0.518	X

Tabla 5

En cuanto a la mAP(50) de los modelos entrenados, ya se observa una mejora significativa en el Modelo 1 tras la incorporación de 611 nuevas imágenes de residuos en playa (de 1,613 a 2,224) y el uso de un modelo más complejo, pasando de YOLOv8n (nano) a YOLOv8m (medium). Esto ha permitido un aumento en la mAP(50) de 0.164 a 0.261.

Por su parte, la mAP(50) del Modelo 3 subió a 0.506, principalmente gracias a la incorporación de 1,933 nuevas imágenes (de 2,224 a 4,157), enfocadas en las categorías minoritarias (cardboard, glass, metal). Desglosando la mAP(50) por clase, se observa que la mejora proviene del aumento en la precisión AP(50) en todas las clases minoritarias, a excepción de la clase predominante, “plastic” que se ha mantenido estable en torno a 0.41, “cardboard” aumento de 0.136 a 0.498, “glass” aumento de 0.212 a 0.619 y “metal” de 0.284 a 0.499.

Finalmente, el Modelo 4 muestra un incremento ligero en la mAP(50), pasando de 0.506 a 0.516. Así mismo, en este modelo se observa una pequeña mejora en todas las clases, excepto, en la clase predominante “plastic” cuya AP(50) disminuyó de 0.407 a 0.28. Esto se debe a la eliminación de 863 imágenes (de 4,157 a 3,294) que contenían mayormente instancias de plástico, reduciendo el número total de instancias de 3,874 a 2,373, la cual se realizó con el objetivo de balancear las clases, en un intento de reducir el margen de error en el modelo.

El Modelo 4 obtuvo el mayor mAP(50) con 0.516, sin embargo, al realizar la comparativa de predicciones en las mismas imágenes de prueba, el Modelo 3 mostró un mejor desempeño general, identificando una mayor cantidad de basura y clasificando las clases

correctas con mayor precisión. Aunque el Modelo 3 fue competitivo en algunas imágenes, el Modelo 2 destacó en escenarios con mayor cantidad de residuos, a pesar de cometer más errores de clasificación.

Se puede visualizar la evaluación de los cuatro modelos en imágenes de prueba en "Anexo C: Evaluación en Imágenes de Prueba".

6. Conclusiones y oportunidades

6.1. Conclusiones

- El objetivo de este trabajo fue desarrollar una herramienta basada en visión artificial que permitiera la identificación y clasificación de residuos sólidos en playas, utilizando un modelo YOLO entrenado con datasets específicos.
- Una vez hechas las pruebas en diversos escenarios y con los resultados analizados, se observó un mejor rendimiento en dos de los modelos desarrollados, dependiendo de si se trataba de videos o imágenes. En particular, el modelo 2 fue seleccionado para la detección en videos, ya que mostró un rendimiento superior al enfrentarse a imágenes con una gran cantidad de residuos. En cambio, para la detección en imágenes estáticas, el modelo 3 fue el más adecuado, logrando un mAP(50) de 0.506.
- Estos resultados muestran la efectividad de utilizar distintos datasets para mejorar la precisión en la identificación de categorías de residuos como cartón, vidrio y metal.
- A pesar del éxito obtenido, el proyecto presenta ciertas limitaciones. La escasez de imágenes en algunos datasets y la heterogeneidad de los tipos de residuos que se pueden encontrar, afectaron la capacidad del modelo para identificar con precisión algunas categorías.

6.2. Utilidad

Finalmente, a través de Streamlit se ha puesto el modelo a disposición en una app con una interfaz sencilla e intuitiva para que cualquier persona lo pueda poner a prueba.¹¹ El

¹¹ El link al web app: <https://beach-litter-detect.streamlit.app/>

funcionamiento de esta se basa en cargar una imagen o un video, dándole la posibilidad al usuario de fijar los parámetros de “Confidence Score Threshold”, “Show image in full screen” y “IoU Threshold”. Una vez fijados estos parámetros y dándole a la opción “Predict” el modelo entra en funcionamiento y procede a identificar en la imagen cargada, los desechos sólidos que se encuentren presentes además de su respectiva categoría, así mismo imprime un índice que indica el número de objetos detectados en cada categoría. Luego muestra las imágenes individuales de los objetos que identificó como desechos con su respectiva etiqueta. Para videos, el funcionamiento es similar, con la diferencia de que este genera un nuevo video en donde se muestran las etiquetas a medida que van pasando los desechos por la imagen y en la parte superior se va mostrando un inventario con el número de objetos que se han ido identificando por categoría. Esto puede llegar a tardar unos minutos en generar el video con la identificación y clasificación de los desechos.

Esta interfaz se ha diseñado con el propósito de poner a prueba, compartir y socializar este modelo, por lo que se puede afirmar que se encuentra en una versión beta. La intención a futuro es flexibilizar la forma en que el usuario pueda interactuar y utilizar el modelo, adaptándolo a diversos formatos y tecnologías dependiendo del escenario que sea usado, ya sea a través de cámaras de vigilancia estáticas, dispositivos móviles o incluso drones. En términos prácticos, esta herramienta busca aumentar la eficiencia en los procesos de identificación y clasificación de residuos en entornos naturales. Si bien en esta primera versión ha sido entrenada para la detección de desechos sólidos en las playas, lo que se busca es ampliar su entrenamiento para que este resulte funcional en otros escenarios naturales como bosques, mares, ríos e incluso en ambientes de ciudad.

Desde el año 2010 la secretaría de salud del Gobierno de España lleva a cabo un programa de seguimiento¹² en distintas playas que rodean toda la costa española con el fin de identificar los diferentes tipos de desechos que quedan en las zonas en diferentes épocas del año, para posteriormente determinar su procedencia y desarrollar políticas públicas al

¹² El programa de seguimiento al que se hace referencia es el “Programa de seguimiento de basuras marinas en playas” de la SECRETARÍA DE ESTADO DE MEDIO AMBIENTE de España, el cual se puede consultar en el informe: https://www.miteco.gob.es/content/dam/miteco/es/costas/temas/proteccion-medio-marino/programadeseguimientodebasurasmarinas_febrero2022_v21_tcm30-419874.pdf

respecto. Hasta la actualidad, este levantamiento de información se ha hecho de forma manual, a cargo de voluntarios y funcionarios que acuden a los puntos seleccionados en el programa de seguimiento para tomar nota de las categorías de desechos que se encuentren en el momento de forma manual. Los datos obtenidos son digitalizados y sometidos a su respectivo proceso de análisis. Este programa es un ejemplo de los escenarios en donde este modelo puede entrar a potenciar los procesos, mejorando la eficiencia en el levantamiento de información, abaratando los costos de la misma y permitiendo que estos seguimientos se puedan hacer en periodos de tiempo mucho más cortos.

6.3. Próximos pasos

- Ampliación del Dataset: Recolectar más imágenes en circunstancias diferentes como cambio en estaciones del año, cambio de condiciones climáticas con el fin de mejorar la capacidad del modelo de generalizar a distintos entornos.
- Ampliación de las categorías y tipos de residuos, esto permite entrar en detalle en la detección de residuos.
- Mejorar la eficiencia del modelo: Particularmente en el caso de los videos es necesario mejorar la velocidad en que el modelo detecta y clasifica los datos, tienen como meta que estos se puedan predecir en tiempo real.
- Integración de nuevas funcionalidades: Para llevar a cabo la expansión del modelo a distintos entornos naturales, se puede incorporar el reconocimiento de objetos flotantes o residuos submarinos, reconocimiento de desechos en orillas, bancos de ríos y lagos para abordar una gama más amplia de contaminación marina.
- Implementar el reconocimiento de residuos sólidos entre la vegetación: con lo que se espera llegar a escenarios como áreas protegidas, bosques y parques naturales, de manera eficiente, lo que podría implementarse en cámaras de vigilancia que capturen imágenes en tiempo real, de esta forma a través del continuo entrenamiento del modelo y la implementación de diferentes tecnologías se puede llevar el modelo a otros entornos y explotar al máximo el uso de este.

- Análisis del impacto ambiental: Realizar estudios que permitan cuantificar el impacto del uso del modelo en la reducción de residuos de manera cualitativa y cuantitativa con el fin de maximizar el valor y la efectividad del mismo.

7. Bibliografía

Banco Mundial (2018) Los desechos: un análisis actualizado del futuro de la gestión de los desechos sólidos, Centro de Enseñanza sobre el Desarrollo, del Banco Mundial, en Tokio. Disponible en: <https://www.bancomundial.org/es/news/immersive-story/2018/09/20/what-a-waste-an-updated-look-into-the-future-of-solid-waste-management>

Kaza, Silpa; Yao, Lisa C.; Bhada-Tata, Perinaz; Van Woerden, Frank. (2018) What a Waste 2.0, A Global Snapshot of Solid Waste Management to 2050. Urban Development, Washington, DC, World Bank. <https://hdl.handle.net/10986/30317>

United Nations Environment Programme (2021). From Pollution to Solution: A global assessment of marine litter and plastic pollution. Nairobi. © 2021 United Nations Environment Programme <https://www.unep.org/interactives/pollution-to-solution/>

SECRETARÍA DE ESTADO DE MEDIO AMBIENTE (2022) PROTOCOLO DE MUESTREO PROGRAMA DE SEGUIMIENTO DE BASURAS MARINAS EN PLAYAS, MINISTERIO PARA LA TRANSICIÓN ECOLÓGICA Y EL RETO DEMOGRÁFICO, disponible en: https://www.miteco.gob.es/content/dam/miteco/es/costas/temas/proteccion-medio-marino/programadeseguimientodebasurasmarinas_febrero2022_v21_tcm30-419874.pdf

8. Anexos

8.1. Anexo A: Imágenes de Datasets

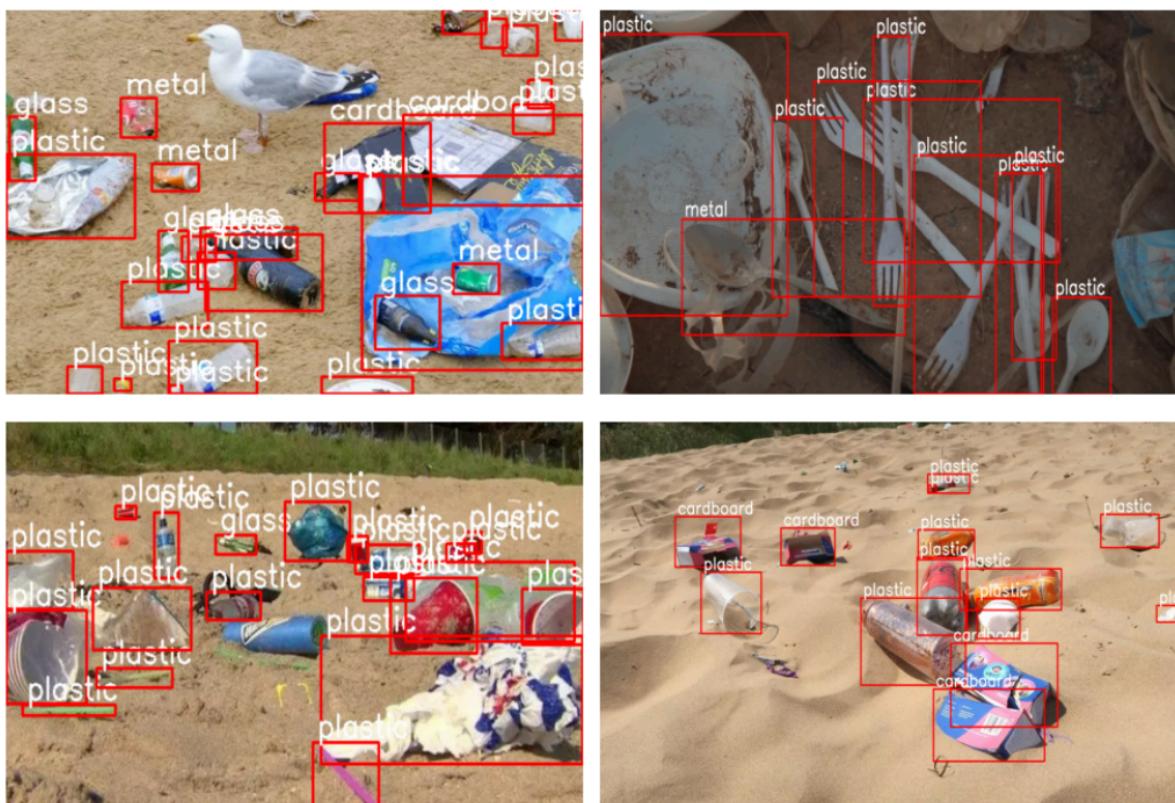
(1) TACO Dataset YOLO



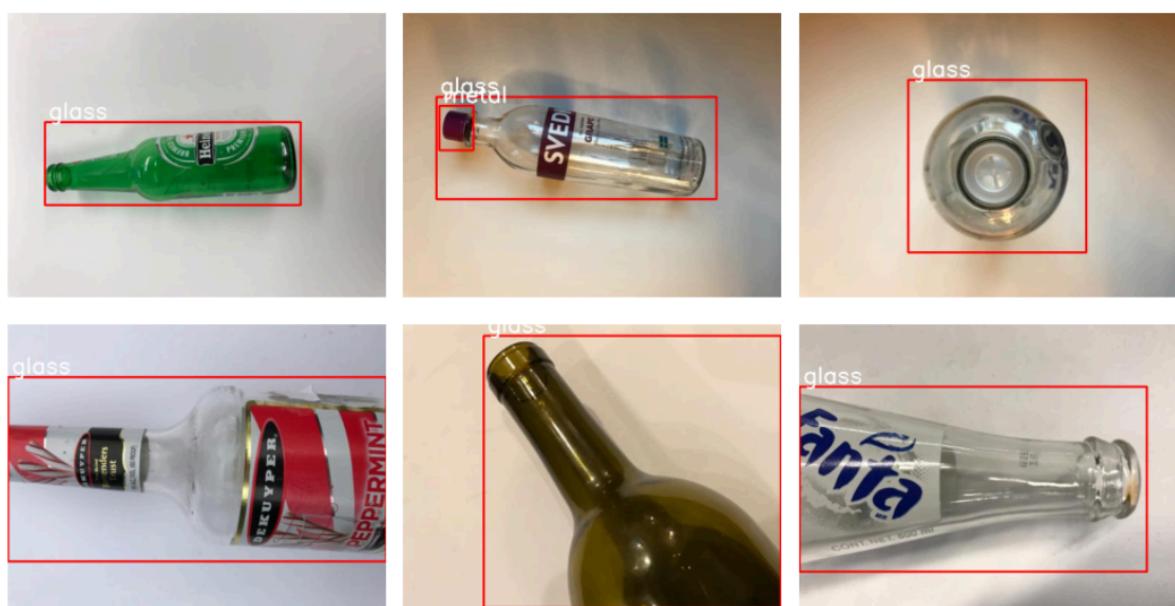
(2) Open Litter Map



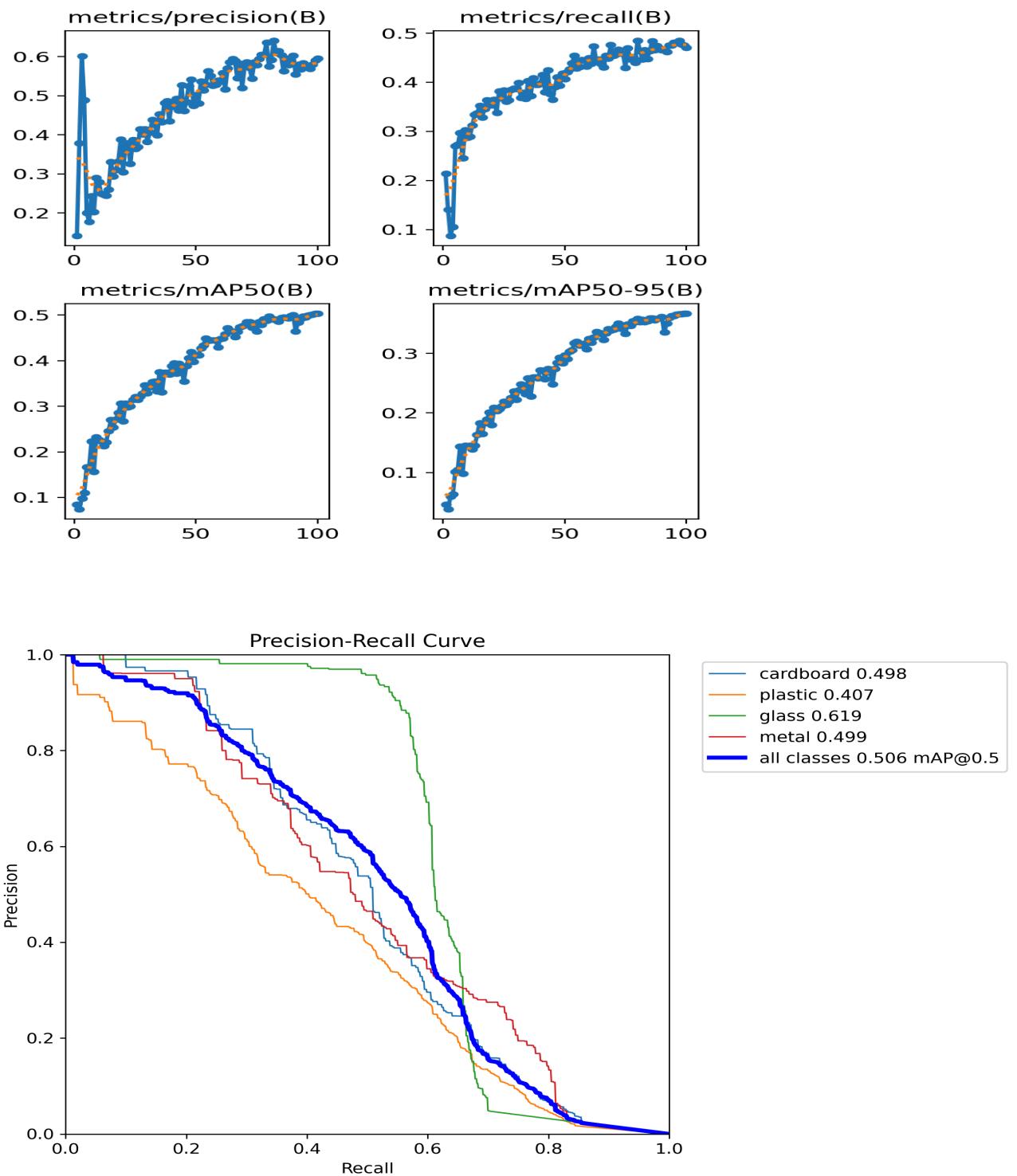
(4) Glass images for waste segregation

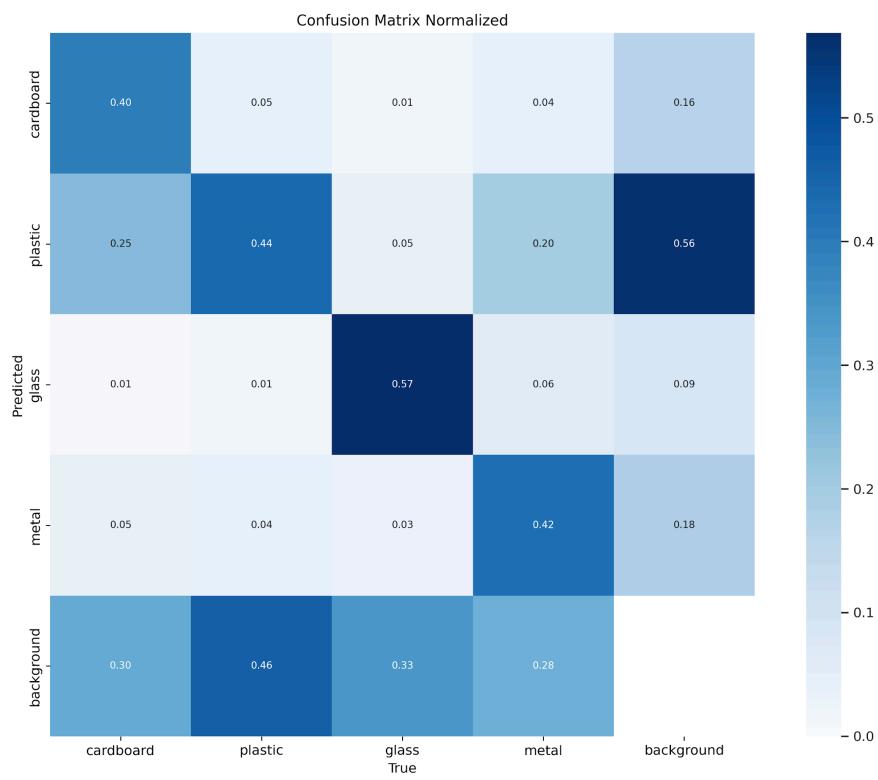
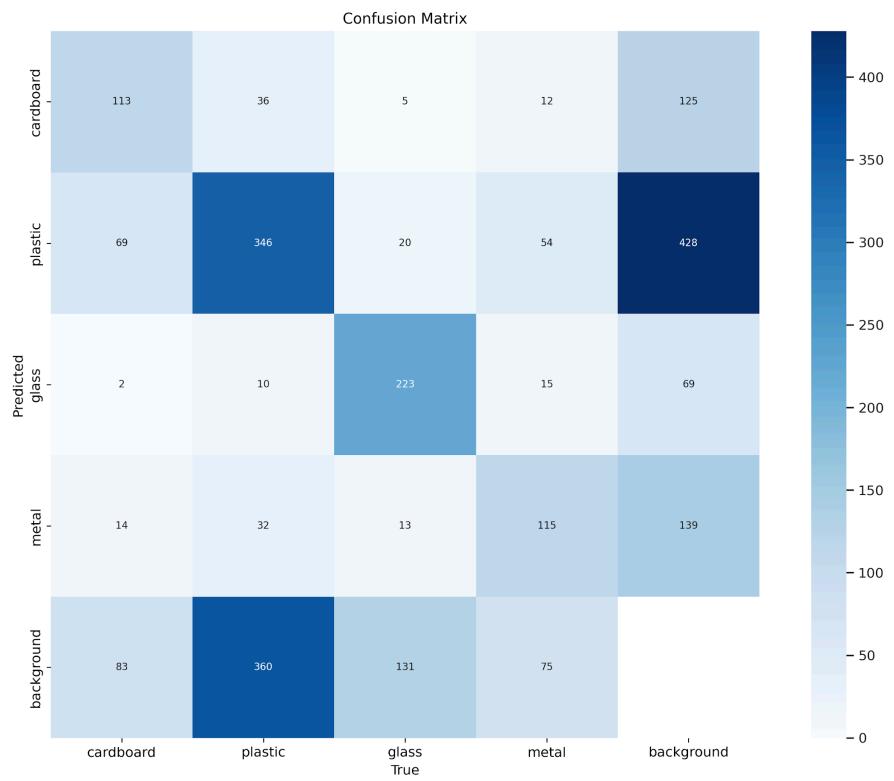


(4) Glass images for waste segregation



8.2. Anexo B: Gráficos de Evaluación





8.3. Anexo C: Evaluación en Imágenes de Prueba



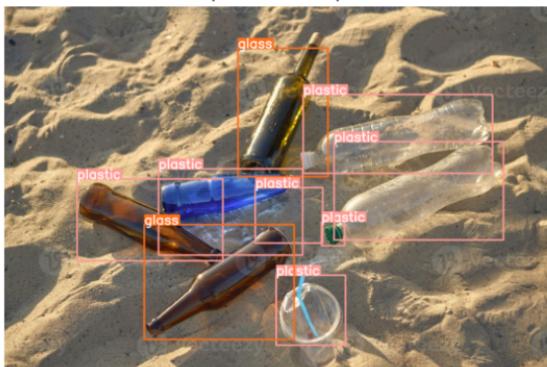
Modelo 1
(7 detecciones)



Modelo 2
(6 detecciones)



Modelo 3
(9 detecciones)



Modelo 4
(3 detecciones)



Modelo 1
(27 detecciones)



Modelo 2
(60 detecciones)



Modelo 3
(110 detecciones)



Modelo 4
(91 detecciones)



Modelo 1
(30 detecciones)



Modelo 2
(94 detecciones)



Modelo 3
(50 detecciones)



Modelo 4
(40 detecciones)



8.4. Anexo D: Código

```
import matplotlib.pyplot as plt
import numpy as np
import os
import pandas as pd
import re
import shutil
import cv2
from PIL import Image
from ultralytics import YOLO
from sklearn.model_selection import train_test_split

#####
# FUNCIONES DE PREPROCESADO DE IMÁGENES
#####

def convert_yolo_labels_to_standard(
    img_tensor: np.ndarray,
    yolo_coordinates: list,
    class_id_is_string=False
):
    """
    Convert YOLO formatted label coordinates to standard bounding box
    coordinates.

    Parameters:
    -----
    img_tensor: np.ndarray
    """

    Parameters:
```

```

    The image as a numpy array.

yolo_coordinates: list

    A list containing YOLO formatted coordinates [class_id,
center_x, center_y, width, height].


Returns:
-----
tuple: A tuple containing (class_id, x_min, y_min, x_max, y_max)
where:
    class_id (int): The class ID of the object.

    x_min (int): The x-coordinate of the top-left corner of the
bounding box.

    y_min (int): The y-coordinate of the top-left corner of the
bounding box.

    x_max (int): The x-coordinate of the bottom-right corner of
the bounding box.

    y_max (int): The y-coordinate of the bottom-right corner of
the bounding box.

"""
image_height, image_width, _ = img_tensor.shape

# DEBUG
# print('class index:', yolo_coordinates[0])

# Parse the YOLO coordinates.

if class_id_is_string is False:
    class_id = int(yolo_coordinates[0])
else:
    class_id = yolo_coordinates[0]

```

```

center_x = float(yolo_coordinates[1])
center_y = float(yolo_coordinates[2])
width    = float(yolo_coordinates[3])
height   = float(yolo_coordinates[4])

# Convert to absolute coordinates.

box_center_x = center_x * image_width
box_center_y = center_y * image_height
box_width     = width      * image_width
box_height    = height     * image_height

x_min = int(box_center_x - (box_width/2))
y_min = int(box_center_y - (box_height/2))
x_max = int(box_center_x + (box_width/2))
y_max = int(box_center_y + (box_height/2))

return (class_id, x_min, y_min, x_max, y_max)

#####
#####

def display_annotated_image(
    img_fname,
    class_list,
    img_dir,
    label_dir
)

```

```

) :

"""

    Displays an image with bounding boxes and class labels annotated
from YOLO format label files.

Parameters:
-----
: str
    The filename of the image (including the extension) to be
annotated and displayed.

: list
    List of class names where each index corresponds to the class
ID used in the YOLO label files.

: str
    Directory containing the image files (.jpg format) to be
annotated.

: str
    Directory containing the YOLO label files (.txt format) that
provide bounding box coordinates
    and class IDs for each object in the image.

"""



```

```

# a txt file might contain more than 1 object coords.

yolo_coords_lists = yolo_coords.splitlines()

for yolo_coords_list in yolo_coords_lists:

    class_id,    x_min,    y_min,    x_max,    y_max    =
convert_yolo_labels_to_standard()

    img_tensor=img_tensor,
    yolo_coordinates=yolo_coords_list.split(' ')
)

# Add bounding box in img.

img_labeled = cv2.rectangle(
    img=img_tensor,
    pt1=(x_min, y_min),
    pt2=(x_max, y_max),
    color=(0, 0, 255), # Note in cv2: BGR (Red)
    thickness=1
)

text = f"{class_list[class_id]} ({class_id})"

# Add class text in img.

text_x = x_min + 5 # Positioning the text slightly to the
right wrt bounding box (0,0).

text_y = y_min - 5 # Positioning the text slightly to the
top.

img_labeled = cv2.putText(
    img = img_labeled,
    text=text,

```

```

        org=(text_x, text_y),
        fontFace=cv2.FONT_HERSHEY_SIMPLEX,
        fontScale=0.4,
        color=(255, 255, 255), # (White)
        thickness=1

    )

print(img_name)
plt.imshow(img_labeled[..., ::-1])
plt.axis('off')
plt.show()

#####
#####

def copy_img_and_label_to_dirs(
    img_name,
    img_dir,
    new_img_dir,
    label_dir,
    new_label_dir,
):
    img_fname = f'{img_name}.jpg'
    shutil.copyfile(
        src=os.path.join(img_dir, img_fname),
        dst=os.path.join(new_img_dir, img_fname)
    )

```

```

label_fname = f'{img_name}.txt'

shutil.copyfile(
    src=os.path.join(label_dir, label_fname),
    dst=os.path.join(new_label_dir, label_fname)
)

#####
#####

def recategorize_labels_txt(
    label_dir,
    old_class_list,
    new_class_list,
    old_to_new_class_dict
):
    """
    Recategorizes YOLO label files by changing the class IDs according
    to a provided mapping from old to new classes.

    Parameters:
    -----
    label_dir: str
        Directory containing the YOLO label files (.txt format) to be
        recategorized.
    old_class_list: list
        List of the original class names corresponding to the class
        IDs in the YOLO label files.
    """

```

```

new_class_list: list

    List of the new class names to which the old class names will
be mapped.

old_to_new_class_dict: dict

    Dictionary that maps old class names (keys) to new class names
(values), allowing the recategorization of labels.

"""

def replace_old_to_new_class_txt(
    yolo_coords,
    old_class_list,
    new_class_list,
    old_to_new_class_dict
):

    old_clase = old_class_list[int(yolo_coords.split(' ')[0])]

    new_clase = old_to_new_class_dict[old_clase]
    new_clase_id = str(new_class_list.index(new_clase))

    new_text = re.sub(
        pattern='^\d+',
        repl=new_clase_id,
        string=yolo_coords
    )

    return new_text

print(len(os.listdir(label_dir)), 'labels txt to recategorize...')

for label_fname in os.listdir(label_dir):
    label_path = os.path.join(label_dir, label_fname)

    with open(label_path, mode='r') as f:

```

```

yolo_coords = f.read()

# a txt file might contain more than 1 object coords.

yolo_coords_lists = yolo_coords.splitlines()

for i, yolo_coord in enumerate(yolo_coords_lists):

    # change class_id in yolo coord line in txt.

    new_yolo_coord = replace_old_to_new_class_txt(
        yolo_coord,
        old_class_list,
        new_class_list,
        old_to_new_class_dict
    )

    if i == 0:

        with open(label_path, mode='w') as file:
            file.write(new_yolo_coord)

    else:

        with open(label_path, mode='a') as file:
            file.write(f'\n{new_yolo_coord}')

print('Done!')


#####
#####
```

```

    class_to_delete: list
):
"""
    Deletes specific class annotations from YOLO label files based on
a given list of classes to remove.

Parameters:
-----
label_dir: str
    Directory containing the YOLO label files (.txt format) to be
processed.

class_list: list
    List of all class names corresponding to the class IDs in the
YOLO label files.

class_to_delete: list
    List of class names to be removed from the label files.

"""

print('Total labels txt:', len(os.listdir(label_dir)))

txts_affected = set()
anno_affected = 0

for label_fname in os.listdir(label_dir):
    label_path = os.path.join(label_dir, label_fname)

    with open(label_path, mode='r') as f:
        yolo_coords = f.read()

        # a txt file might contain more than 1 object coords.

        new_yolo_coords = []
        for i, yolo_coord in enumerate(yolo_coords.splitlines()):
            if yolo_coord in class_to_delete:
                continue
            else:
                new_yolo_coords.append(yolo_coord)

```

```

        class_id = int(yolo_coord.split(' ')[0])

        clase = class_list[class_id]

        if clase in class_to_delete:

            txts_affected.add(label_path)

            anno_affected += 1

        else:

            new_yolo_coords.append(yolo_coord)

    # Generate whole label string and strip empty lines.

    new_yolo_coords_text = '\n'.join(new_yolo_coords).strip()

    # Overwrite label file with new label annotations.

    with open(label_path, mode='w') as file:

        file.write(new_yolo_coords_text)

    print('Total label txts affected:', len(txts_affected))

    print('Total annotations eliminated:', anno_affected)

#####
#####

def count_n_classes_(
    labels_dir,
    class_list
):
    """

```

```
Counts the occurrences of each class in YOLO label files and  
calculates their percentage contribution.
```

Parameters:

```
labels_dir: str
```

Directory containing the YOLO label files (.txt format) to be analyzed.

```
class_list: list
```

List of class names corresponding to the class IDs in the YOLO label files.

Returns:

```
pd.DataFrame
```

A DataFrame containing the count and percentage of occurrences for each class. The DataFrame includes:

- 'count': Total number of instances of each class found across all label files.

- 'perc': Percentage of each class in relation to the total number of class instances.

A row with the label 'total' contains the sum of all class counts and the total percentage (which sums to 100%).

```
"""
```

```
class_count_dict = {clase:0 for clase in class_list}  
total_count = 0  
  
for label_fname in os.listdir(labels_dir):  
    label_path = os.path.join(labels_dir, label_fname)  
    with open(label_path, mode='r') as f:
```

```

yolo_coords = f.read()

# print('\n', label_fname)

# a txt file might contain more than 1 object coords.

yolo_coords_lists = yolo_coords.splitlines()

for yolo_coords_list in yolo_coords_lists:

    clase = class_list[int(yolo_coords_list.split(' ')[0])]

    # print('class_id:', yolo_coords_list.split(' ')[0], '->',
'clase:', clase)

    class_count_dict[clase] += 1

    total_count += 1


df = pd.DataFrame(columns=['count', 'perc'])

for key in class_count_dict:

    count = class_count_dict[key]

    perc = np.round(count / total_count * 100, 2)

    df.loc[key] = [int(count), perc]

df.loc['total'] = [df['count'].sum(), df['perc'].sum()]

return df

#####
#####
```

def get_class_count_per_image_df(

label_dir,

class_list,

exclude_background_imgs=True

```
):

"""
    Generates a DataFrame with the count of each class and their
percentage per image, based on YOLO label files.

Parameters:
-----
label_dir: str
    Directory containing YOLO label files (.txt format) to
analyze.

class_list: list
    List of class names corresponding to the class IDs in the YOLO
label files.

exclude_background_imgs: bool, optional
    Whether to exclude images with no class annotations
(background images) from the output.

    Defaults to True.

Returns:
-----
pd.DataFrame
    A DataFrame where each row corresponds to an image,
containing:
        - 'name': Name of the image (without extension).
        - Counts of each class for that image.
        - 'total': Total number of class annotations in the image.
            - Percentage columns for each class representing the
proportion of that class in relation to the total.

"""


```

```

data = []

for label_fname in os.listdir(label_dir):
    # DEBUG
    # print(label_fname)

    # set up dict with individual img class count info.

    name = '.'.join(label_fname.split('.')[:-1])
    class_count_dict = {'name': name, 'total': 0}
    class_count_dict.update({clase:0 for clase in class_list})

    label_path = os.path.join(label_dir, label_fname)
    with open(label_path, mode='r') as f:
        yolo_coords = f.read()

    # a txt file might contain more than 1 object coords.

    yolo_coords_lists = yolo_coords.splitlines()
    for yolo_coords_list in yolo_coords_lists:
        # DEBUG
        # print(int(yolo_coords_list.split(' ')[0]))

        clase = class_list[int(yolo_coords_list.split(' ')[0])]

        # print(clase)

        # update class count in dict.

        class_count_dict[clase] += 1
        class_count_dict['total'] += 1

    # add individual class count dict to data (list).

    data.append(class_count_dict)

```

```

# Create df.

df = pd.DataFrame(data)

# Create perc prop for each category.

for clase in class_list:

    df[f'{clase}_perc'] = np.round(df[clase] / df['total'] * 100,
2)

# Exclude background imgs (total = 0).

if exclude_background_imgs is True:

    df = df.loc[df.total > 0]

return df

#####
#####

def get_class_count_abs_perc_df(
    class_count_dict
):
    """
        Generates a DataFrame containing the absolute counts and
percentage distribution of class instances.

    Parameters:
    -----
    class_count_dict: dict

```

```
Dictionary where the keys represent class names and the values  
represent the count of instances for each class.
```

```
Returns:
```

```
-----
```

```
pd.DataFrame
```

```
A DataFrame with two columns:
```

- 'count': Absolute count of instances for each class.
- 'perc': Percentage of each class relative to the total count.

```
The DataFrame includes a 'total' row, which sums up the counts  
and percentages for all classes.
```

```
"""
```

```
df = pd.DataFrame(columns=['count', 'perc'])

total_count = sum([v for k, v in class_count_dict.items()])

for key in class_count_dict:
    count = class_count_dict[key]
    perc = np.round(count / total_count * 100, 2)
    df.loc[key] = [int(count), perc]

df.loc['total'] = [df['count'].sum(), df['perc'].sum()]

return df
```

```
#####
#####
```

```
def simulate_undersampling_yolo(
    df_img_class_count,
    class_list,
    target_class,
    n_instances,
    perc_threshold=0.0,
    only_int_imgs=False,
    show_bef_aft=True,
):
    """
    Simulates undersampling of images in a YOLO dataset to reduce the
    number of instances of a target class.

    Parameters:
    -----
    df_img_class_count: pd.DataFrame
        DataFrame containing image-level class counts and percentages,
        where each row corresponds to an image.
    class_list: list
        List of all class names corresponding to the class IDs in the
        YOLO label files.
    target_class: str
        Class name to be undersampled.
    n_instances: int
        Target number of instances for the target class after
        undersampling.
    perc_threshold: float, optional
        Minimum percentage of the target class in an image to be
        considered for undersampling. Defaults to 0.0.
```

```

    only_int_imgs: bool, optional

        If True, only images with numeric names (TACO images) are
        considered. Defaults to False.

    show_bef_aft: bool, optional

        If True, displays the class counts and percentages before and
        after undersampling. Defaults to True.

    Returns:
    ----
    pd.DataFrame

        A subset of the original DataFrame, containing images such
        that the number of instances of the target class
        is reduced to the specified `n_instances`.

    """
    df = df_img_class_count.copy()

    # Exclude background images.
    df = df.loc[df.total > 0]

    # get class count dict from orig df.
    class_count_dict_bef = df.sum()[class_list].to_dict()
    n_imgs_bef = len(df)

    # Df filters:
    ## TACO images (only digit name).

    if only_int_imgs is True:
        taco_imgs_mask = df.name.str.isdigit()

    else:

```

```

taco_imgs_mask = pd.Series([True] * len(df))

df = df.loc[taco_imgs_mask]

## filter by target_class & perc_threshold.

df = df.loc[df[f'{target_class}_perc'] >= perc_threshold]

# shuffle df.

df = df.sample(frac=1).reset_index(drop=True)

# get cumsum col for target class.

df[f'{target_class}_cumsum'] = df[target_class].cumsum()

# get 1st img that adds up to n_instances in target_class (first
row is the smallest).

n_instances_index = df.loc[df[f'{target_class}_cumsum'] >=
n_instances].index[0]

# filter df up to n_instances_index.

df = df.loc[:n_instances_index]

# get class count dict from subset df.

class_count_dict_subset = df.sum()[class_list].to_dict()

n_imgs_subset = len(df)

# get class count dict of orig dict - subset dict.

class_count_dict_aft = {k:class_count_dict_bef[k] - v for k, v in
class_count_dict_subset.items()}

n_imgs_aft = n_imgs_bef - n_imgs_subset

```

```

    if show_bef_aft is True:
                    df_count_bef      =
get_class_count_abs_perc_df(class_count_dict_bef)
                    df_count_sub      =
get_class_count_abs_perc_df(class_count_dict_subset)
                    df_count_aft      =
get_class_count_abs_perc_df(class_count_dict_aft)

# display counts.

print(f"Before ({n_imgs_bef}):")
print(df_count_bef, '\n')
print(f"Subset ({n_imgs_subset}):")
print(df_count_sub, '\n')
print(f"After ({n_imgs_aft}):")
print(df_count_aft, '\n')

return df

#####
#####
```

```

def get_resized_imgs(
    img_dir,
    save_dir,
    dsizes=(700, 700),
    shape_thresh=(2500, 2500),   # (height, width)
    size_thresh_bytes=1_000_000,
    num_tracking=100
)
```

```
) :  
    """  
  
        Resizes images larger than a specified size or resolution and  
        saves them to a directory. Smaller images are copied  
  
        directly. Displays statistics on the resizing process.  
  
  
    Parameters:  
    -----  
  
        img_dir: str  
  
            Directory containing the source images.  
  
        save_dir: str  
  
            Directory where the resized or copied images will be saved.  
  
        dsizes: tuple, optional  
  
            Target dimensions (width, height) to resize images to.  
            Defaults to (700, 700).  
  
        shape_thresh: tuple, optional  
  
            Threshold dimensions (height, width). Images larger than this  
            will be resized. Defaults to (2500, 2500).  
  
        size_thresh_bytes: int, optional  
  
            File size threshold in bytes. Images larger than this size  
            will be resized. Defaults to 1,000,000 bytes (1MB).  
  
        num_tracking: int, optional  
  
            Number of images processed before printing progress. Defaults  
            to 100.  
  
  
    Returns:  
    -----  
  
        None
```

```

    The function saves the resized or copied images to `save_dir`  

and prints a summary of the operation, including  
  

    the total number of resized images, total sizes of the source  

and save directories, and the percentage reduction  
  

    in size after resizing.  
  

"""

src_dir_bytes = 0  

save_dir_bytes = 0  

resize_counter = 0  
  

    print(f'Resizing      images      larger      than  

{size_thresh_bytes/1_000_000}MB  or  width  >  {shape_thresh[0]}  or  

height > {shape_thresh[1]}  to {dsizes}...\n')  
  

for i, img_file in enumerate(os.listdir(img_dir)):  
  

    img_path = os.path.join(img_dir, img_file)  

    img_size = os.path.getsize(img_path)  

    src_dir_bytes += img_size  
  

    img_tensor = cv2.imread(img_path)  

    if img_tensor is None:  

        print(f"Error reading image as tensor: {img_path}")  

        continue  
  

    height, width, _ = img_tensor.shape  
  

    # check file size.  

    if (img_size  >= size_thresh_bytes) or ((height >  

shape_thresh[0]) or (width > shape_thresh[1])):  

        # If equal or larger than 1MB, then resize it.  

        img_resized = cv2.resize(

```

```

        src=img_tensor,
        dsize=dsize,
        interpolation=cv2.INTER_LINEAR
    )

# Save resized img.

img_file_jpg = f"{img_file.split('.')[0]}.jpg"
save_img_path = os.path.join(save_dir, img_file_jpg)
cv2.imwrite(save_img_path, img_resized)

# Increase values of counters.

resize_counter += 1
save_dir_bytes += os.path.getsize(save_img_path)

else:

    # Copy original img from img_dir to save_dir.

    shutil.copyfile(
        src=img_path,
        dst=os.path.join(save_dir, img_file)
    )
    save_dir_bytes += img_size

# Tracking.

if (i + 1) % num_tracking == 0:
    print(f"{i + 1} images processed...")

# Info.

total_images = len(os.listdir(img_dir))

```

```

src_dir_GB = src_dir_bytes / (1024**3)

src_dir_MB = src_dir_bytes / (1024**2)

save_dir_GB = save_dir_bytes / (1024**3)

save_dir_MB = save_dir_bytes / (1024**2)

print(f"""

-Total images resized: {resize_counter} out of {total_images}
({resize_counter/total_images*100:.2f}%).

-Total size of src directory "{img_dir}":
    - {src_dir_GB:.2f} GB ({src_dir_MB:.2f} MG).

-Total size of save directory "{save_dir}":
    - {save_dir_GB:.2f} GB ({save_dir_MB:.2f} MG).

-Total reduction in size after resizing:
    - {(src_dir_MB - save_dir_MB)/src_dir_MB*100:.2f}%!
"""
)

```

#####	FUNCIONES	DE	EVALUACIÓN
<pre> def extract_best_models(runs_dir, save_dir, models_included): """ Extracts the best YOLO model weights ('best.pt') from multiple model training runs and saves them to a specified directory. </pre>	<i></i>	<i></i>	<i></i>

```

Parameters:

-----
runs_dir: str

    Directory containing the YOLO training runs, where each
model's weights are saved.

save_dir: str

    Directory where the extracted 'best.pt' files will be saved.

models_included: list

    List of models that have already been processed and should be
excluded from extraction.

Returns:

-----
None

    The function prints the list of newly included models and
copies their 'best.pt' files to the `save_dir`.

    If no new models are found or any errors occur during the file
extraction, relevant messages are printed.

"""

model_names = [model_name for model_name in os.listdir(runs_dir)
if model_name not in models_included]

if len(model_names) == 0:
    print('No new models added to YOLO_runs!')

if len(model_names) > 0:
    print(f"New models included in YOLO_runs:\n{model_names}")
    print(f"\nExtracting best.pt files to {save_dir}...")
    for model_name in model_names:

```

```

                best_model_path = os.path.join(runs_dir, model_name,
'detect', 'train', 'weights', 'best.pt')

                new_model_path      = os.path.join(save_dir,
f'{model_name}.pt')

                # copy file to new dir with changed name.

                shutil.copy(
                    src=best_model_path,
                    dst=new_model_path
                )

                # verify the file actually exists.

                if not os.path.exists(new_model_path):
                    print('Error extracting best.pt files to:',
new_model_path)

                    print('\nDone!')

```

#####

#####

```

def show_different_args(
    runs_dir,
    model_1,
    model_2
):
    """
        Compares the training arguments of two YOLO models and returns the
        differences.

```

Parameters:

```

runs_dir: str
    Directory containing the YOLO model runs.

model_1: str
    Name of the first model to compare.

model_2: str
    Name of the second model to compare.

Returns:
-----
pd.DataFrame
    A DataFrame containing only the arguments where the two models differ.

"""
# Get parameter names to add to a df as columns.

yaml_path = os.path.join(runs_dir, [model_1, model_2][0],
'detect', 'train', 'args.yaml')

with open(yaml_path, mode='r') as file:
    args_yaml = file.read()

    params = list(map(lambda x: x.split(':')[0],
args_yaml.strip().splitlines()))

df = pd.DataFrame(columns=params)

# Add each model as independent row to the df with args as columns.

for i, model_name in enumerate([model_1, model_2]):
    yaml_path = os.path.join(runs_dir, model_name, 'detect',
'train', 'args.yaml')

```

```

        with open(yaml_path, mode='r') as file:
            args_yaml = file.read()

            paired_args = args_yaml.strip().splitlines()
            args = list(map(lambda x: x.split(': ')[1], paired_args))

            df.loc[model_name] = args

    # return df with only args with differences.

    model_1_row = df.iloc[0]
    model_2_row = df.iloc[1]
    diff_ser = model_1_row != model_2_row

    diff_cols = []
    for i, arg in enumerate(diff_ser):
        col = diff_ser.index[i]
        if (arg is True) & (col not in ['name', 'save_dir']):
            diff_cols.append(col)

    return df[diff_cols]

#####
#####
```

```

def compare_main_metrics_df(
    runs_dir,
    models,
```

```

    int_names=False
):
"""
    Compares the main metrics from the last epoch of training for
multiple YOLO models.

Parameters:
-----
runs_dir: str
    Directory containing the YOLO model runs, where each model's
training results are saved.

models: list
    List of model names to compare.

int_names: bool, optional
    If True, model names in the output DataFrame will be replaced
with sequential integers (e.g., 'model_1', 'model_2').

    Defaults to False, where the original model names are used.

Returns:
-----
pd.DataFrame
    A DataFrame where each row corresponds to a model, containing
the metrics from the last epoch of training.

    The index of the DataFrame is the model name (or integer if
`int_names=True`), and columns are the metric names.

"""
for i, model_name in enumerate(models):
    model_results_path = os.path.join(runs_dir, model_name,
'detect', 'train')

```

```

### Stage 1: show last epoch metrics for each model ###

epoch_metrics = pd.read_csv(os.path.join(model_results_path,
'results.csv'))

epoch_metrics.columns = epoch_metrics.columns.str.strip() # Large blank space before text in cols.

epoch_metrics.rename(columns={'epoch':'last_epoch'},
inplace=True)

if int_names:

    name = 'model_' + str(i + 1)

else:

    name = model_name

epoch_metrics['model'] = name

if i == 0:

    df = pd.DataFrame(epoch_metrics.iloc[-1]).T

else:

    df.loc[i] = epoch_metrics.iloc[-1]

df = df.set_index('model')

return df

#####
#####
```

def compare_metrics_plot(

```

    runs_dir,
    model_1,
    model_2,
    figsize=(35, 35)

):
"""
Generates and displays plots comparing training batch images and
main metrics between two YOLO models.

Parameters:
-----
runs_dir: str
    Directory containing the YOLO model runs, where each model's
    training results are stored.
model_1: str
    Name of the first model to compare.
model_2: str
    Name of the second model to compare.
figsize: tuple, optional
    Size of the figure for the plots. Defaults to (35, 35).

Returns:
-----
None
    The function displays images of training batches and various
    metrics for both models.

    It does not return any values but shows the plots directly.

"""

```

```

        for i, model_name in enumerate([model_1, model_2]):

            model_results_path = os.path.join(runs_dir, model_name,
'detect', 'train')




    ### display 3 training batch images for each model ###

    fig, axes = plt.subplots(nrows=1, ncols=3, figsize=figsize)

        for j, train_img_fname in enumerate(['train_batch0.jpg',
'train_batch1.jpg', 'train_batch2.jpg']):

            train_img_path = os.path.join(model_results_path,
train_img_fname)

            image = Image.open(train_img_path)

            axes[j].imshow(np.array(image))

            axes[j].set_title(f"model_{i + 1}: {train_img_fname}")

            axes[j].axis('off')

            plt.subplots_adjust(wspace=0.05) # Default is 0.2


    ### show main metric pics (loss, conf matrices, PR curve) ###

        for g, img_fname in enumerate(['results.png',
'confusion_matrix.png', 'confusion_matrix_normalized.png',
'PR_curve.png']):

            fig2, axes2 = plt.subplots(nrows=1, ncols=2, figsize=figsize)

            for i, model in enumerate([model_1, model_2]):

                img_path = os.path.join(runs_dir, model, 'detect',
'train', img_fname)

                image = Image.open(img_path)


                axes2[i].imshow(np.array(image))

                axes2[i].set_title(model)

                axes2[i].axis('off')

```

```

# Adjust the spacing between the plots

plt.subplots_adjust(wspace=0.05) # Default is 0.2

#####
#####



def show_pred_images(
    yolo_models_dict,
    pred_img_path,
    mode='detect',
    conf=0.25,
    iou=0.7,
    show_orig_img=False,
    figsize=(10, 10)
):
    """
    Displays prediction results from multiple YOLO models on a given
    image.

    Parameters:
    -----
    yolo_models_dict: dict
        Dictionary where keys are model names and values are YOLO
        model instances.

        Each model instance should have `predict` or `track` methods
        for generating predictions.

    pred_img_path: str
        Path to the image file on which predictions are to be made.
    """

```

```
mode: str, optional

    Mode for prediction. Can be 'detect' for object detection or
'track' for object tracking. Defaults to 'detect'.

conf: float, optional

    Confidence threshold for predictions. Defaults to 0.25.

iou: float, optional

    Intersection over Union (IoU) threshold for non-maximum
suppression. Defaults to 0.7.

show_orig_img: bool, optional

    If True, displays the original image before showing the
predictions. Defaults to False.

figsize: tuple, optional

    Size of the figure for displaying the plots. Defaults to (10,
10).
```

Returns:

None

The function displays the original image (if `show_orig_img` is True) and the predictions from each model

in separate subplots. It does not return any values but shows the plots directly.

"""

```
img_name = pred_img_path.split('/')[-1]
```

```
print(img_name)
```

```
if show_orig_img:

    plt.figure(figsize=figsize)

    plt.title(img_name)
```

```
plt.imshow(Image.open(pred_img_path))

plt.axis('off')

fig, axes = plt.subplots(nrows=len(yolo_models_dict), ncols=1,
figsize=figsize)

for i, (model_name, yolo_model) in enumerate(yolo_models_dict.items()):
    if mode == 'detect':
        pred_img_results = yolo_model.predict(pred_img_path,
verbose=False, iou=iou, conf=conf)[0]
    elif mode == 'track':
        pred_img_results = yolo_model.track(pred_img_path,
verbose=False, iou=iou, conf=conf)[0]
    else:
        raise Exception("mode must be one of the following: ['detect', 'track'].")
n_pred_objects = len(pred_img_results.bboxes)
pred_img_tensor = pred_img_results.plot(conf=False,
labels=True)
axes[i].imshow(pred_img_tensor[..., ::-1])
axes[i].set_title(f'{model_name} ({n_pred_objects})')
axes[i].axis('off')

#####
#####
```

```

def show_individual_detections(
    model_path,
    img_path,
    class_list,
    figsize_detect=(3, 3)
):
    """
    Displays the predictions from a YOLO model on a given image,
    including the annotated image with bounding boxes
    and individual cropped detections.

    Parameters:
    -----
    model_path: str
        Path to the YOLO model weights file.
    img_path: str
        Path to the image file on which predictions are to be made.
    class_list: list
        List of class names corresponding to class IDs used by the
        YOLO model.
    figsize_detect: tuple, optional
        Size of the figure for displaying individual cropped
        detections. Defaults to (3, 3).

    """
    yolo_model = YOLO(model_path, verbose=False)
    pred_results = yolo_model.predict(img_path, iou=0.3)[0]

```

```

        pred_img    =   pred_results.plot(conf=False,    labels=False,
line_width=2)

        plt.imshow(pred_img[..., ::-1])

        img_name = img_path.split('/')[-1][-1]

        plt.title(f" {img_name} ({len(pred_results)})")

        plt.axis('off')

        plt.show()

orig_img = cv2.imread(img_path)

for i, pred_box in enumerate(pred_results.boxes):

    class_id = int(pred_box.cls[0])

    class_name = class_list[class_id]

    conf_score = np.round(float(pred_box.conf[0]), 2)

    detection_xyxy = pred_box.xyxy

    x_min, y_min, x_max, y_max = detection_xyxy[0]

    x_min, y_min, x_max, y_max = int(x_min), int(y_min),
int(x_max), int(y_max)

    cropped_img = orig_img[y_min:y_max, x_min:x_max]

    plt.figure(figsize=figsize_detect)

    plt.imshow(cropped_img[..., ::-1])

    plt.title(f"Detection {i+1} ({class_name} {conf_score})")

    plt.axis('off')

    plt.show()

```

```
#####
#####
```

```
def put_text_in_img_middle_upper(
    img_tensor,
    text,
    font_scale=0.8,
    font_thickness=2,
    color=(255, 255, 255)  # white.
):
```

```
"""
```

```
Adds text to the upper middle of an image tensor.
```

```
Parameters:
```

```
-----
```

```
img_tensor: numpy.ndarray
```

```
        Image tensor (in the form of a NumPy array) where the text will be added.
```

```
text: str
```

```
        The text to be added to the image.
```

```
font_scale: float, optional
```

```
        Scale factor that is multiplied by the font-specific base size. Defaults to 0.8.
```

```
font_thickness: int, optional
```

```
        Thickness of the text strokes. Defaults to 2.
```

```
color: tuple, optional
```

```
        Color of the text in BGR format. Defaults to white (255, 255, 255).
```

```

    Returns:
    -----
    numpy.ndarray

        The modified image tensor with the text added at the upper
        middle position.

    """
    # Load image tensor (np.array).
    image = img_tensor

    # Get image dimensions.
    height, width, _ = image.shape

    # Define the text and font.
    font = cv2.FONT_HERSHEY_SIMPLEX

    # Calculate the size of the text.
    (text_width, text_height), baseline = cv2.getTextSize(text, font,
    font_scale, font_thickness)

    # Calculate X position to center the text.
    x = (width - text_width) // 2

    # Calculate Y position (slightly below the top of the image).
    y = int(height * 0.05) + text_height # Adjust the 0.1 multiplier
    to move the text up or down.

    # Add text to the image.

```

```

        mod_image_tensor = cv2.putText(image, text, (x, y), font,
font_scale, color, font_thickness)

    return mod_image_tensor

#####
#####
```

#####

```

def generate_video_tracking(
    raw_video_dir,
    raw_video_name,
    save_dir,
    model_dir,
    model_name,
    fps_slowdon_factor=1.0,
    conf_threshold=0.25, # default in YOLO.
    iou_threshold=0.5, # default in YOLO.
    show_detect_counts=False,
    text_separator='|',
    class_list=None,
):
    """
    Processes a video by applying object tracking with a YOLO model
    and generates a new video with tracking annotations.

```

Parameters:

`raw_video_dir: str`

Directory where the raw video is located.

```
raw_video_name: str
    Name of the raw video file (without extension).

save_dir: str
    Directory where the processed video will be saved.

model_dir: str
    Directory where the YOLO model weights are located.

model_name: str
    Name of the YOLO model to be used (without extension).

fps_slowdown_factor: float, optional
    Factor to slow down the video. A value of 1.0 keeps the original speed. Values less than 1.0 slow down the video. Defaults to 1.0.

conf_threshold: float, optional
    Confidence threshold for object detection. Defaults to 0.25.

iou_threshold: float, optional
    Intersection over Union (IoU) threshold for object detection. Defaults to 0.5.

show_detect_counts: bool, optional
    Whether to show detection counts on the video frames. If True, `class_list` must be provided. Defaults to False.

text_separator: str, optional
    Separator used in the text showing detection counts. Defaults to '|'.

class_list: list of str, optional
    List of class names corresponding to YOLO model class indices. Required if `show_detect_counts` is True.

Returns:
-----
```

```
None
```

```
The function saves the processed video with annotations to the  
'save_dir` and prints a success message.
```

```
"""
```

```
if (show_detect_counts is True) & (class_list is None):  
    raise Exception('Provide YOLO model class_list to show  
detection counts in frames!')  
  
model_path = os.path.join(model_dir, f'{model_name}.pt')  
raw_video_path = os.path.join(raw_video_dir,  
f'{raw_video_name}.mp4')  
  
video_tracking_name =  
f'{raw_video_name}_{model_name}_fpsFactor{fps_slowdon_factor}_conf{con  
f_threshold}_iou{iou_threshold}_showDetectCounts{show_detect_counts}'  
video_tracking_path = os.path.join(save_dir,  
f'{video_tracking_name}.mp4')  
  
# Initialize videoCapture object.  
cap = cv2.VideoCapture(raw_video_path)  
ret, frame = cap.read()  
  
# Reading 1st frame.  
# ret is True if the frame was successfully read, and False if  
there was an error or the video has ended.  
  
if not ret:  
    print('Error: Could not read the first frame of the video.')  
    cap.release()  
    exit()
```

```

height, width, _ = frame.shape

# Set new FPS (slowing down the video by 0.6x, so new FPS is half
+ little more of the original FPS).

original_fps = int(cap.get(cv2.CAP_PROP_FPS))

new_fps = int(original_fps * fps_slowdown_factor)

# Initialize VideoWriter with exact dimensions as original video.

output = cv2.VideoWriter(
    video_tracking_path, # where the output will be saved.

    cv2.VideoWriter_fourcc(*'MP4V'), # Four Character Code,
commonly used for writing .mp4.

    new_fps, # Get the frame rate per second of the original
video (same speed).

    (width, height) # dimensions of the original video frames.

)

# Load yolo model.

yolo_model = YOLO(model_path)

# Contar detecciones únicas y por clase.

if show_detect_counts is True:

    unique_ids = set()

    detect_class_count_dict = {clase:0 for clase in class_list}

# Process all frames from video file (stream).

while cap.isOpened():

```

```

ret, frame = cap.read()

if not ret:
    break


pred_frame_results = yolo_model.track(frame,
                                       # stream=True,
                                       persist=True,
                                       verbose=False,
                                       conf=conf_threshold,
                                       iou=iou_threshold
                                       )[0]

labeled_frame = pred_frame_results.plot(conf=False)

if show_detect_counts is True:
    for detect_img_box in pred_frame_results.boxes:
        # DEBUG.

        # print(detect_img_box)

        if detect_img_box.is_track:
            detect_id = int(detect_img_box.id)

            if detect_id not in unique_ids:
                detect_class = class_list[int(detect_img_box.cls)]
                detect_class_count_dict[detect_class] += 1
                unique_ids.add(detect_id)

# Definir texto de conteo.

```

```

        total_count = str(len(unique_ids))

        text = 'Total:' + total_count

        tuplas_clase_count = [(clase.capitalize(), str(count)) for
clase, count in detect_class_count_dict.items()]

        for tupla in tuplas_clase_count:

            text += f" {text_separator} {':'.join(tupla)}"

# Añadir texto en imagen anotado.

labeled_frame = put_text_in_img_middle_upper(
    img_tensor=labeled_frame,
    text=text,
    font_scale=0.6,
    font_thickness=2,
    color=(255, 255, 255) # white.

)

# Write processed frame to an output video.

output.write(labeled_frame)

cap.release() # Release the video capture object.
output.release() # Release the video writer object.
cv2.destroyAllWindows() # Close all OpenCV windows.

print(f"{{video_tracking_path}} has been successfully created!")

```