UNCUYO
UNIVERSIDAD NACIONAL DE CUYO

FACULTAD DE INGENIERÍA

# Trabajo Final Inteligencia Artificial I

---

Año 2021: Visión Artifical

Lopez, Valentin - Legajo 12439

# Contents

# Resumen

Se busca desarrollar un sistema de visión artificial que sea capaz de clasificar 4 tipos de piezas diferentes:

1. Clavos
2. Tornillos
3. Arandelas
4. Tuercas

Además, en el caso de que se identifique un **Tornillo** o un **Clavo** se debe calcular la longitud del mismo.

Para la identificación de objetos se hace uso de dos algoritmos de clasificación bastante conocidos: **KNN** y **K-Means**

## KNN: K-Nearest Neighbors

Es un algoritmo de clasificación **supervisado**, lo que quiere decir, que se conoce de antemano las diferentes clases o grupos posibles entre las cuales se debe realizar la clasificación. Aquí el elige el valor del parámetro **K**, el cual es, la cantidad de vecinos mas cercanos que serán tomados en cuenta para determinar a que agrupación pertenece el dato incógnita que se desea clasificar.

## K-Means

Por otro lado, K-Means es un algoritmo del tipo **no supervisado**, lo cual significa que no se conoce previamente el numero de **"clusters"** entre los cuales se debe realizar la clasificación. En este algoritmo el valor de **K** representa el número de **centroides** que se utilizaran para armar los clusters. Dichos clusters serán determinados tomando en cuenta todo el dataset que se tiene más el dato incógnita que se desea clasificar. Al final del proceso el dato incógnita quedara agrupado en uno de dichos clusters completando así la tarea de clasificación.

# Introducción

La visión artificial es un área de estudio que busca dotar a las maquinas con la habilidad de percibir el mundo visualmente. Esto se logra a través de imágenes, las cuales son descompuestas y analizadas en función de los píxeles que las conforman. De esta forma a través de análisis numéricos se logran encontrar patrones, morfologías, contornos, etc que pueden ser analizados para permitir a las maquinas actuar en reacción

La visión artificial es ampliamente utilizada en diversos campos. Algunas de sus aplicaciones se encuentra en: autos autónomos, detección de defectos en lineas de producción, sistemas de identificación de productos, metrología, verificación de montajes, lectura de códigos y caracteres (OCR), Bin picking.

# Especificación del agente

## Tipo de agente

El agente es del tipo **reactivo basado en modelos**. Estos agentes mantienen un estado interno que les permite seguir el rastro de aspectos del mundo que no son observables en el estado actual. La percepción actual se combina con el estado interno antiguo para generar una mejor descripción del estado actual.

## Tabla REAS

| **Agente** | **Medidas de rendimiento** | **Entorno** | **Actuadores** | **Sensores** |
|---|---|---|---|---|
| Sistema de clasificación de imágenes | Correcta clasificación de los objetos | Fondo negro<br><br>Luz controlada | Filtros de imagen<br><br>Algoritmos de clasificación | Cámara |

## Entorno de trabajo

La puesta en escena consta de una caja con fondo negro, con tapa, que recibe luz a través de 2 orificios hechos en la misma. Esto se realiza para controlar la luz que incide sobre el objeto y que no se produzcan sombras que interfieran en la detección. Este entorno de trabajo posee las siguientes características:

- **Totalmente observable**: los sensores (cámara) proporcionan toda la información relevante.
- **Estocástico**: El estado actual no dice nada sobre cual sera el estado siguiente.
- **Episódico**: El siguiente episodio no depende de las acciones previas.
- **Estático**: No se producen cambios en el entorno mientras en agente razona.
- **Discreto**: Existe un número concreto de percepciones y acciones claramente definidos.
- **Agente individual**: No existen otra entidad en el entorno que afecte el comportamiento del agente.

# Diseño del agente

El agente consta de dos sistemas:

- → Una App de celular que sirve para seleccionar el algorimo a utilizar, sacar la foto que desea clasificar y visualizar el resultado junto a todos los filtros aplicados
- → Un servidor en Python donde se aplican los filtros y operaciones correspondientes a la imagen y se ejecutan los algoritmos de KNN y K-Means, según cual haya sido llamado desde la aplicación.

## App de celular

En las siguientes imágenes se puede apreciar todas las etapas que componen la aplicación para obtener la clasificación del objeto:

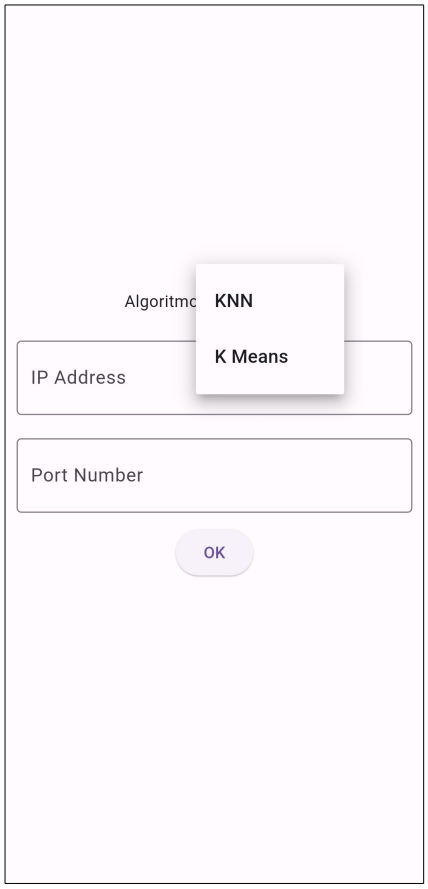



Figure 1: Selección del algoritmo a utilizar

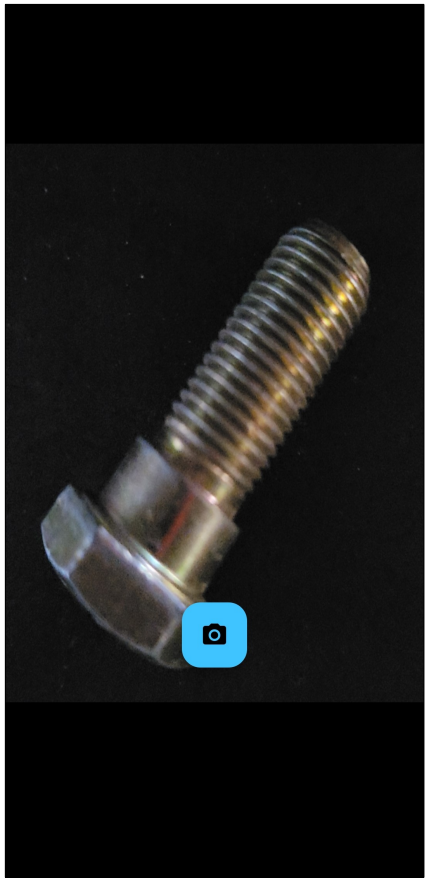


Figure 2: Cámara para tomar la foto


Figure 3: Resultado y filtros utilizados


Figure 4: Plots resultantes

# Servidor Python

El servidor se estructura de la siguiente forma:

```
AI_Classifier_Server
├── images
│   ├── arandelas
│   ├── clavos
│   ├── tornillos
│   └── tuercas
└── src
    ├── main.py
    ├── models
    │   ├── ai_classifier.py
    │   ├── kmeans_classifier.py
    │   └── knn_classifier.py
    ├── routes
    │   ├── kmeans_classifier.py
    │   └── knn_classifier.py
    ├── utils
    │   ├── data_augmentation.py
    │   └── rename_img.py
    └── ai_classifier_logger.py
```

# Código

Los módulos **main.py** y los contenidos dentro de la carpeta **routes** son donde instancia el servidor Python, se crea un **api** y se definen las rutas correspondientes para utilizar los algoritmos de busqueda **KNN** o **K-Means** segun se lo solicite desde la App de celular.

En cada ruta es donde se hace el llamado al algoritmo correspondiente para luego agrupar toda la información resultantes en un fichero .zip y enviarla devuelta a la App para su visualización

A continuación se expone el código de estos 3 scripts:

## Main.py

```
from flask import Flask
from flask_restful import Api

from routes.knn_classifier import KNNClassifierRoute
from routes.kmeans_classifier import KMeansClassifierRoute

app = Flask(__name__)
api = Api(app)
api.add_resource(KNNClassifierRoute, '/knn_classifier')
api.add_resource(KMeansClassifierRoute, '/kmeans_classifier')

@app.route("/")
def hello_world():
    return "<p>Hello, World!</p>"

if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0")
```

## Ruta: kmeans_classifier.py

```
import cv2
import numpy as np
from flask_restful import Resource
from flask import request, send_file
from models.kmeans_classifier import KMeansClassifier

import zipfile

from io import BytesIO
```

```
from PIL import Image

class KMeansClassifierRoute(Resource):
    kmeans_classifier = KMeansClassifier()

    def post(self):
        try:
            image = request.files['image']

            image = np.frombuffer(image.read(), np.uint8)
            image = cv2.imdecode(image, cv2.IMREAD_GRAYSCALE)

            img_vec, centroids, category, object_length, images_dict =
                KMeansClassifierRoute.kmeans_classifier.predict(image)
            # Delete the label image because cannot be serialized
            del images_dict["label_image"]

            # Get the plots
            fig_plots = KMeansClassifierRoute.kmeans_classifier.generate_data_plots
                (img_vec=img_vec, centroids=centroids)
            images_dict = {
                **images_dict,
                **fig_plots
            }

            # Save in memory the category
            categoryTxt = BytesIO()
            categoryTxt.write(f"{category[:-1]}".encode())

            # Save in memory the object length
            object_lengthTxt = BytesIO()
            object_lengthTxt.write(f"{object_length}".encode())

            # Create a BytesIO object to store the ZIP file in memory
            zip_buffer = BytesIO()

            # Create a ZipFile object
            with zipfile.ZipFile(zip_buffer, 'a', zipfile.ZIP_DEFLATED, False) as
                zip_file:

                # Add the category to the ZIP file
                zip_file.writestr(f'category.txt', categoryTxt.getvalue())

                # Add the object length to the ZIP file
                zip_file.writestr(f'object_length.txt', object_lengthTxt.getvalue()
                    )

                for img_name in images_dict:
```

```
                    # Convert NumPy array to PIL Image
                    pil_image = Image.fromarray(images_dict[img_name])

                    # Create an in-memory file-like object
                    image_buffer = BytesIO()
                    pil_image.save(image_buffer, format='JPEG')  # You can adjust
                        the format as needed

                    # Add the in-memory file to the ZIP file
                    zip_file.writestr(f'{img_name}.jpg', image_buffer.getvalue())

            # Seek to the beginning of the buffer
            zip_buffer.seek(0)

            # Return the ZIP file as a response
            return send_file(zip_buffer, download_name="images.zip", as_attachment=
                True)

        except Exception as e:
            print(e)
            return {"error": str(e)}, 500
```

## Ruta: knn_classifier.py

```
import cv2
import numpy as np
from flask_restful import Resource
from flask import request, send_file
from models.knn_classifier import KNNClassifier

import zipfile

from io import BytesIO
from PIL import Image

class KNNClassifierRoute(Resource):
    knn_classifier = KNNClassifier()

    def post(self):
        try:
            image = request.files['image']

            image = np.frombuffer(image.read(), np.uint8)
            image = cv2.imdecode(image, cv2.IMREAD_GRAYSCALE)
```

```
        img_vec, category, object_length, images_dict = KNNClassifierRoute.
            knn_classifier.predict(image, k=3)
        # Delete the label image because cannot be serialized
        del images_dict["label_image"]

        # Get the plots
        fig_plots = KNNClassifierRoute.knn_classifier.generate_data_plots(
            img_vec=img_vec)
        images_dict = {
            **images_dict,
            **fig_plots
        }

        # Save in memory the category
        categoryTxt = BytesIO()
        categoryTxt.write(f"{category[:-1]}".encode())

        # Save in memory the object length
        object_lengthTxt = BytesIO()
        object_lengthTxt.write(f"{object_length}".encode())

        # Create a BytesIO object to store the ZIP file in memory
        zip_buffer = BytesIO()

        # Create a ZipFile object
        with zipfile.ZipFile(zip_buffer, 'a', zipfile.ZIP_DEFLATED, False) as
            zip_file:

            # Add the category to the ZIP file
            zip_file.writestr(f'category.txt', categoryTxt.getvalue())

            # Add the object length to the ZIP file
            zip_file.writestr(f'object_length.txt', object_lengthTxt.getvalue()
                )

            for img_name in images_dict:

                # Convert NumPy array to PIL Image
                pil_image = Image.fromarray(images_dict[img_name])

                # Create an in-memory file-like object
                image_buffer = BytesIO()
                pil_image.save(image_buffer, format='JPEG')  # You can adjust
                    the format as needed

                # Add the in-memory file to the ZIP file
                zip_file.writestr(f'{img_name}.jpg', image_buffer.getvalue())

        # Seek to the beginning of the buffer
```

```
        zip_buffer.seek(0)

        # Return the ZIP file as a response
        return send_file(zip_buffer, download_name="images.zip", as_attachment=
            True)

    except Exception as e:
        print(e)
        return {"error": str(e)}, 500
```

## Algoritmos KNN y K-Means

En la carpeta **models** se encuentran las 3 clases que constituyen los algoritmos de clasificación utilizados.

→ **AIClassifier**

Es una clase abstracta de la cual heredan las clases **KMeansClassifier** y **KNNClassifier**. En ella se encuentran los siguientes métodos:

- **load_images** y **preprocess_image**:

  Son los métodos a través de los cuales se carga la base de conocimiento al sistema. Todas las imágenes son cargadas desde la carpeta **iamges** y luego transformadas a escala de grises. Adicionalmente a esto el método **preprocess_image** se encarga de recortar cada imagen con el fin de guardar en el sistema una base de datos con imágenes de dimensiones homogéneas.

```
# Method to crop and resize the images to always have the same size
def preprocess_image(self, img):
    # crop and resize image
    y_size, x_size = img.shape
    img_cropped = img[
        round(y_size/2)-round(self.img_crop_size/2) : round(y_size/2)+
            round(self.img_crop_size/2),
        round(x_size/2)-round(self.img_crop_size/2) : round(x_size/2)+
            round(self.img_crop_size/2)
    ]
    img_resized = cv2.resize(img_cropped, (self.img_crop_size, self.
        img_crop_size))
    return img_resized
```

```python
def load_images(self, elements):
    # load images
    train_data = []
    train_labels = []

    images_path = os.path.join(AIClassifier.root_folder_path, "images")

    for element in elements:
        imgs = os.listdir(f"{images_path}/{element}")
        self.elements[element] = len(imgs)
        for img in imgs:
            img_new = cv2.imread(
                f"{images_path}/{element}/{img}",
                cv2.IMREAD_GRAYSCALE
            )
            if img_new is not None:
                img_resized = self.preprocess_image(img_new)
                train_data.append(np.array(img_resized))
                train_labels.append(elements.index(element))

            else:
                print(f"Error: It is not possible to read the image {
                    element}/{img}")

    # convert to numpy array
    train_data = np.array(train_data)
    train_labels = np.array(train_labels)

    return train_data, train_labels
```

- **img_to_vec**:

  Es el método encargado de aplicar los filtros, binarizar y posteriormente generar la representación vectorial correspondiente para cada imagen. Para ellos se hace uso de un framework de tratamientos de imágenes llamado **Scikit-Image**. Los filtros y operaciones morfológicas que se llevan a cabo son las siguientes:

  - **Binarización**: conversión de la imagen a blanco y negro.
  - **Cierre** (erosión y luego dilatación): eliminación de huecos internos.
  - **Apertura** (erosión y luego dilatación): eliminación de pequeños ruidos y objetos de fondo.

  Luego se representa cada imagen en con un vector de 9 componentes que contiene:

  - **Excentricidad**: la excentricidad de dicho objeto.
  - **7 Momentos de Hu**: 7 momentos del objeto que son invariables respecto a translaciones, rotaciones y escalamiento de la imagen.

```
# Method to convert the images to a vector representation
def img_to_vec(self, images):
    # Preprocess train images

    # Create an array to store the following properties of the images:
    # - Eccentricity
    # - 7 Hu moments
    img_vec = np.empty([1, 8])

    for img in images:

        # Binarize image
        thresh = threshold_triangle(img)
        image = img > thresh
        image_bw = image.astype(np.uint8) * 255

        # Closing
        image_close = cv2.erode(cv2.dilate(image_bw, self.kernel_close,
            iterations=1), self.kernel_close, iterations=1)
        # Opening
        image_open = cv2.dilate(cv2.erode(image_close, self.kernel_open,
            iterations=1), self.kernel_open, iterations=1)

        # Labeling (identify objects)
        label_image = label(image_open)
        # Get properties of objects
        regions = regionprops(label_image)

        # Get the biggest object and its properties
        area = 0
        for props in regions:
            if props.area > area:
                main_label = props.label
                orientation = props.orientation
                area = props.area
                eccentricity = props.eccentricity
                moments_hu = props.moments_hu

        # Get just the biggest object from the image
        label_image[label_image != main_label] = 0

        new_row = np.array([
            eccentricity,
            *moments_hu
        ])

        img_vec = np.append(img_vec, [new_row], axis=0)
```

```
    return img_vec[1:], orientation, image_bw, image_close, image_open,
        label_image
```

- **calculate_length**

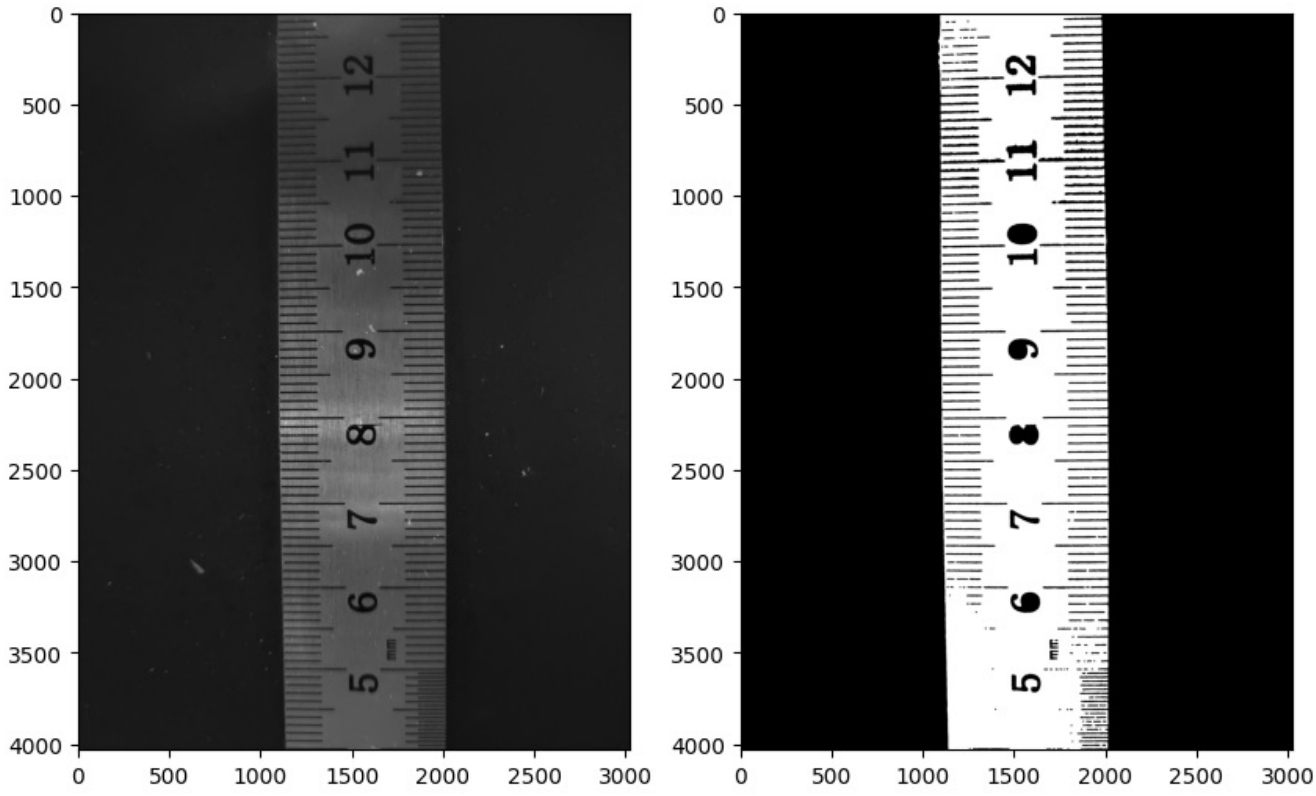
  Finalmente este método se encarga de calcular la longitud del objeto en el caso de haber detectado un **tornillo** o un **clavo**. Esto se logra debido a que todas las fotos son tomadas a la misma distancia vertical.

  El Framework utilizado puede estimar la longitud en pixeles del objeto en la imagen. Para ellos cada imagen previamente rotada de manera que el objeto quede orientado verticalmente y luego se determinan los puntos entremos del mismo. Finalmente su diferencia es la longitud del objeto en pixeles

  Para pasar esa medida a una unidad física de longitud (cm) se utiliza la siguiente relación:

$$\frac{8.8cm}{4032px}$$

  La cual fue obtenida al fotografiar una regla a la misma distancia que los objetos del dataset.




```
# Method to calculate the length of "nails" and "screws"
def calculate_length(self, img, orientation):
    # Rotate the image to put the object vertically
    angle = math.degrees(-orientation)
```

```
    scale = 1.0
    (height, width) = img.shape[:2]
    center = (width / 2, height / 2)
    matrix = cv2.getRotationMatrix2D(center, angle, scale)
    rotated_img = cv2.warpAffine(img, matrix, (width, height))

    # Labeling to get the bbox of the biggest object
    label_image = label(rotated_img)
    regions = regionprops(label_image)
    area = 0
    for props in regions:
        if props.area > area:
            area = props.area
            minr, minc, maxr, maxc = props.bbox
            length = maxr - minr

    length = round(length * self.relation_cm_px, 2) # cm
    return f"{length} cm"
```

→ **KMeansClassifier**

Esta clase hereda de **AIClassifier** y es la que implementa el algoritmo **K-Means**.

- Método **predict**

  Primero se vectoriza la nueva imagen que se quiere y se agrupa junto a las imágenes previamente vectorizadas de la base de conocimiento. Luego se inicializan de manera aleatoria $k = 4$ centroides, de manera que cada uno coincida inicialmente con alguno de los vectores del dataset.

  En este punto, se comienza un proceso iterativo (bucle $while$), donde primero se calcula la distancia de cada punto del dataset a cada uno de los centroides. Luego se determina el centroide más cercano a cada uno de estos puntos y finalmente se actualizan los centroides desplazándolos al valor promedio de los puntos del dataset que tenían ese centroides como el mas cercano. Este proceso iterativo se repite hasta que la variación de los centroides entre 2 iteraciones consecutivas sea nula, particularmente en el código debe ser menor a $centroids\_error$.

  Una vez terminado este proceso y encontrados los centroides, se conoce a que cluster pertenece el dato incógnita pero no se sabe a que categoría pertenece ese cluster. Esta es una de las limitaciones de **K Means**, al ser un algoritmo de aprendizaje de tipo no supervisado. Para poder determinar la categoría a la cual pertenece cada centroide y el dato incógnita se llama al método: **identify_clusters_and_get_prediction** el cual hace uso de **KNN** para determinar esto.

```
def identify_clusters_and_get_prediction(
    self,
    centroids,
    closest_centroid,
    knn_k=6
):
    # KNN algorithm
    knn_k = 6

    # Calculate the distance of each image to each centroid
    distances = [
        [
            self.euclidean_distance(centroid, img)
            for img in self.train_images
        ]
        for centroid in centroids
    ]

    # Get the k-nearest neighbors of each centroid
    neighbors_index = np.argpartition(distances, knn_k, axis=1)[:, :knn_k
        ]
    neighbors = self.train_labels[neighbors_index]

    # Get the most common category of the k-nearest neighbors
    most_commons = np.apply_along_axis(lambda x: np.bincount(x).argmax(),
         axis=1, arr=neighbors)

    # Create an aux array to store the new order full of "inf"
    self.kmeans_labels = np.copy(closest_centroid)
    self.kmeans_labels[:] = self.k
    # Order the centroids by the same order of the categories
    final_centroids = []
    for category in self.categories:
        # Get the index of the category in the "most_commons" array
        centroid_category_index = np.argwhere(most_commons == self.
            categories.index(category))[0]
        # Update the closest centroid with the new centroids's order
        self.kmeans_labels[closest_centroid == centroid_category_index] =
             self.categories.index(category)
        # Reorder the centroids
        final_centroids.append(centroids[centroid_category_index][0])
    # Convert the list to a NumPy array
    final_centroids = np.vstack([final_centroids[0], final_centroids[1],
        final_centroids[2], final_centroids[3]])

    # Get the category of the closest centroid to the new datapoint
    prediction = self.categories[self.kmeans_labels[0]]
```

```
    return final_centroids, prediction
```

```
def predict(
    self,
    img
):
    self.logger.info(f"Predicting")
    # Preprocess and vectorize the image
    img_resized = self.preprocess_image(img)
    img_vec, orientation, image_bw, image_close, image_open, label_image
        = self.img_to_vec([img_resized])
    img_vec = img_vec[0]

    # Build a vector with the images to predict in the 0th position and
        the train images
    imgs_vec = np.vstack([img_vec, self.train_images])

    # KMeans algorithm
    self.logger.info(f"Running algorithm")

    # Initialize the centroids with random images from the train set
    random_args = np.random.randint(0, self.train_images.shape[0], self.k
        )
    centroids = self.train_images[random_args]

    # Repeat the algorithm until the centroids do not change more than
        the "centroids_error"
    old_centroids = np.copy(centroids)
    centroids_error = 0.001
    while (True):

        # Calculate the distance of each image to each centroid
        distances = [
            [
                self.euclidean_distance(img, centroid)
                for centroid in centroids
            ]
            for img in imgs_vec
        ]

        # Assign the closest centroid to each image
        closest_centroids = np.argmin(distances, axis=1)

        # Update the centroids
        old_centroids = np.copy(centroids)
        for i in range(self.k):
            new_centroid = np.mean(imgs_vec[closest_centroids == i], axis
                =0)
            if not np.isnan(new_centroid).any():
```

```
                centroids[i] = new_centroid

        # Break the loop if the centroids do not change more than the "
            centroids_error"
        if (abs(np.sum(old_centroids - centroids)) < centroids_error):
            break

    # Indentify the cluster of each centroid and
    # get the closest centroid to the image to predict that is in the 0th
        position
    final_centroids, prediction = self.
        identify_clusters_and_get_prediction(
        centroids=centroids,
        closest_centroid=closest_centroids
    )

    # Calculate the length of the object if it is a "nail" or a "screw"
    if prediction in ["clavos", "tornillos"]:
        self.logger.warning(f"Calculating length")
        object_length = (
            self.calculate_length(image_open, orientation)
        )
    else:
        object_length = None

    self.logger.info(f"Predicting done")

    return (
        img_vec,
        final_centroids,
        prediction,
        object_length,
        {
            "Escala de grises": img_resized,
            "Binarizacion": image_bw,
            "Cierre": image_close,
            "Apertura": image_open,
            "label_image": label_image
        })
```

→ **KNNClassifier**

Al igual que **KMeansClassifier**, esta clase hereda de **AIClassifier** y es la que implementa el algoritmo **KNN**.

- Método **predict**

  En KNN, al igual que en K-Means, lo primero que se hace es vectorizar la nueva imagen que se quiere clasificar. La diferencia es que al ser un algoritmo de tipo

supervisado, se conoce de antemano que es cada punto presente en la base de conocimiento. Por lo que, teniendo el vector de la nueva imagen, se procede a calcular la distancia del mismo a cada vector presente en el dataset. Luego se ordenan estas distancias de menor a mayor y se eligen las $k$ primeras.

El nuevo dato sera clasificado dentro del grupo al que pertenezcan la mayor cantidad de los $k$ vecinos que se eligieron.

```
def predict(
    self,
    img,
    k : int = 3
):

    self.logger.info(f"Predicting")

    # Preprocess and vectorize the image
    self.logger.info(f"Preprocessing images")
    img_resized = self.preprocess_image(img)
    img_vec, orientation, image_bw, image_close, image_open, label_image
       = self.img_to_vec([img_resized])
    img_vec = img_vec[0]

    self.logger.info(f"Running algorithm")

    # Calculate the distance of each image to the new datapoint
    distances = [
        self.euclidean_distance(img_vec, train_img)
        for train_img in self.train_images
    ]

    # Get the k-nearest neighbors
    neighbors_index = np.argsort(distances)[:k]
    neighbors = self.train_labels[neighbors_index]

    # Get the most common category of the k-nearest neighbors
    most_common = np.bincount(neighbors).argmax()
    prediction = self.categories[most_common]

    # Calculate the length of the object if it is a "nail" or a "screw"
    objects_length = []
    if prediction in ["clavos", "tornillos"]:
        self.logger.warning(f"Calculating length")
        objects_length = (
            self.calculate_length(image_open, orientation)
        )
    else:
```

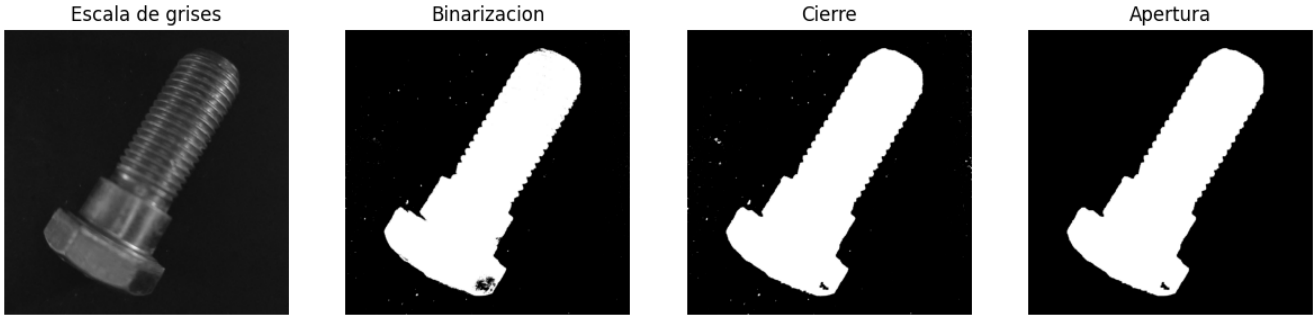
```
        objects_length = None

    self.logger.info(f"Predicting done")

    return (
        img_vec,
        prediction,
        objects_length,
        {
            "Escala de grises": img_resized,
            "Binarizacion": image_bw,
            "Cierre": image_close,
            "Apertura": image_open,
            "label_image": label_image
        })
```

Ambos clases **KNNClassifier** y **KMeansClassifier** poseen un método llamado: $generate\_data\_plots()$ que se encarga de generar los plots que luego serán enviados para si visualización en la App

# Ejemplo de aplicación

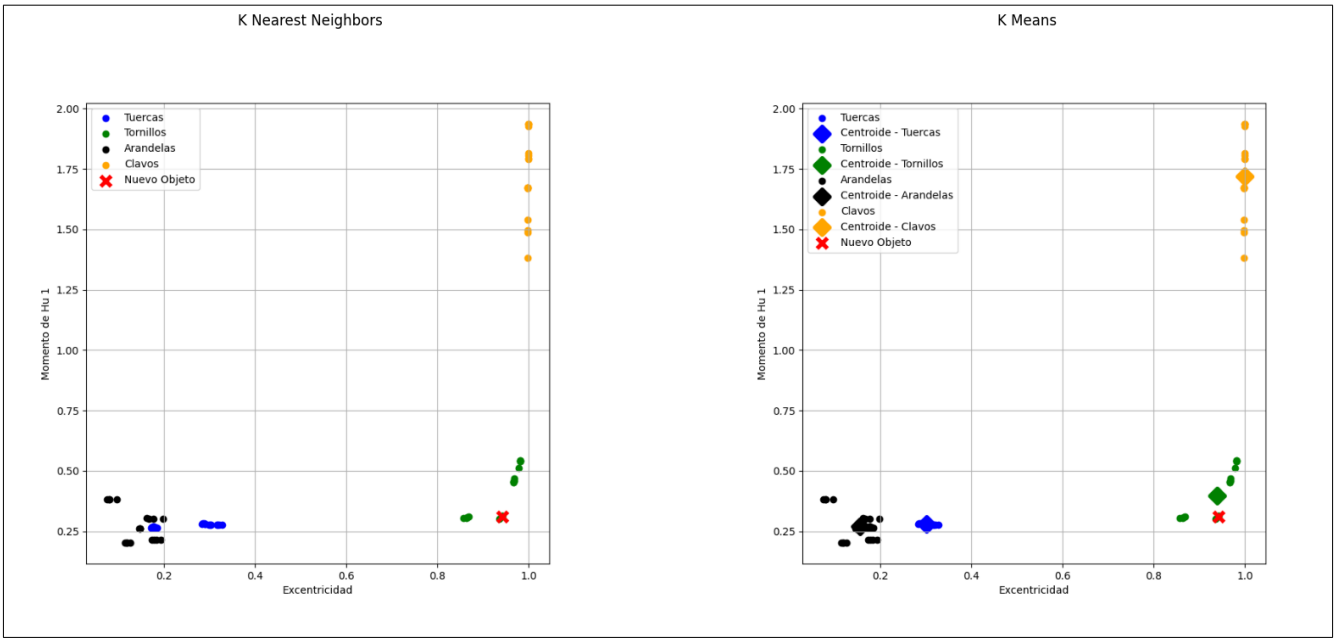
Para ejemplificar el uso de estos dos algoritmos se mostrara el proceso y resultados para la identificación de un **tornillo**

Primero se aplican los filtros y operaciones morfológicas a la imagen para eliminar el ruido de fondo y obtener aislar el tonillo.




A continuación se calculan la **excentricidad** y los **7 momentos de Hu** sobre esta ultima imagen simplificada que se obtuvo.

Una vez vectorizada la imagen se ejecutan ambos algoritmos para encontrar la clasificación. Con el fin de poder visualizar los resultados, se exponen a continuación el gráfico del **primer momento de Hu** vs la **excentricidad**.



La respuesta de cada algorimo fue:

- KNN: Tornillo
- K-Means: Tornillo

Y por tratarse de un clavo:

- Longitud: 4.31 cm

En ambos gráficos se puede apreciar como el objeto incógnita (X roja) se encuentra próximo al cluster identificado como ”Tornillos” (puntos verdes). Por lo que es esperable que ambos algoritmos lo identifiquen como parte del mismo.

Por otro lado, visualizando el gráfico de la derecha se pueden ver, además, los centroides encontrados como resultado del algoritmo de **K-Means**.

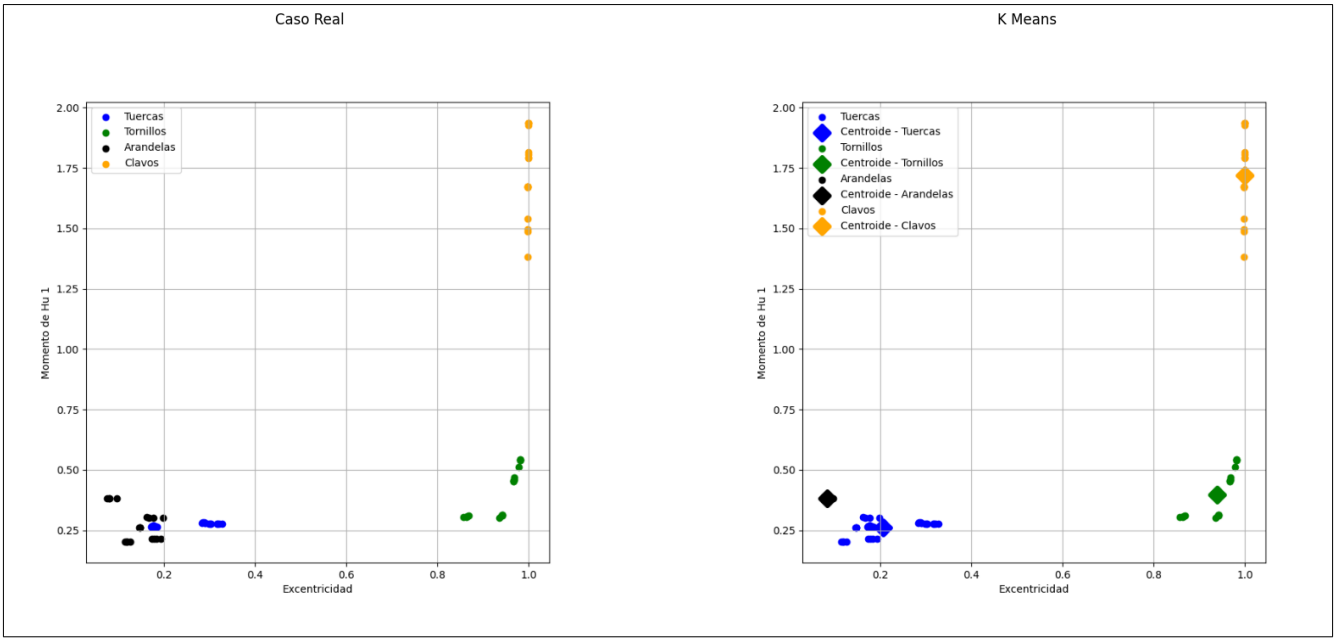
Un aspecto importante a destacar, es la diferencia en la clasificación que ocurre respecto al caso ”real”. El gráfico de la izquierda esta creado considerando las categorías reales de cada punto, por lo que podemos decir que representa las agrupaciones ”reales”. Por otro lado, las agrupaciones en el gráfico de la derecha (K-Means), al depender de los centroides encontrados, presenta diferencias respecto al caso real. Se puede ver como alrededor del centroide de las ”Arandelas” (rombo negro) hay dos puntos que fueron considerados como tales pero que originalmente eran ”Tuercas”.

# Resultados

Lo expuesto al final de la sección anterior deja en evidencia una de las mayores diferencias, y a veces, limitaciones de **K-Means**. Al tratarse de un algoritmo de aprendizaje de tipo no supervisado, ocurre que cuando existen objetos cuyas representaciones se encuentran muy próximas entre si, este algoritmo las clasifica como del mismo grupo, aunque no lo sean.

Este problema se vuelve más evidente entre las ”Arandelas” y ”Tuercas”. cuyos clusters están muy próximos debido a su similitud en forma y tamaño. En estos casos, se encontró que K-Means es mucho más propenso a confundir esto dos grupos que **KNN**, especialmente con objetos que queden representados entre ambas agrupaciones.

En el siguiente gráfico se puede ver un caso en el que, debido a la inicialización aleatoria de los centroides en K-Means, el cluster correspondiente a las ”Arandelas” quedó reducido a solo unos puntos que se encontraban mas aislados que el resto.



# Conclusiones

El algoritmo KNN, para este caso, resulto ser mucho más preciso que el algoritmo KMeans. Esto es debido al problema explicado anteriormente de las tuercas y arandelas en la clasificación de KMeans. Para el caso de KNN, al estar los datos etiquetados previamente, se obtiene una exactitud mucho mayor a la hora de clasificar una nueva imagen.

Además todas las mediciones están fuertemente ligadas a las condiciones lumínicas del entorno. Imágenes con mucho ruido, no llegan a ser filtradas correctamente y por lo tanto esto conduce a clasificaciones erróneas. Potenciales soluciones a este problema serian realizar un estudio de bordes y contornos con la intención de aislar el objeto de ruidos mas intensos o utilizar otros métodos de clasificación de imágenes que sean mas adaptativos, como la utilización de redes neuronales convoluciones. En todo caso este problema de iluminación siempre dependerá del entorno en el que se haga trabajar al agente. En entornos donde se tenga iluminación controlada, estos algoritmos tendrían un comportamiento mas que aceptable.

# Bibliografía y/o referencias

1. Material de la cátedra - cursado 2021
2. Documentación de Scikit-Image: https://scikit-image.org/docs/stable/
3. Momentos de imagen: https://es.wikipedia.org/wiki/Momentos_de_imagen#Momentos_de_Hu:_invariantes_de_rot