Trabajo Final Inteligencia Artificial I-2020: Visión Artificial

Guarise Renzo
Legajo: 12262
Universidad Nacional de Cuyo - Facultad de Ingeniería
Inteligencia Artificial I
Ingeniería Mecatrónica

16 de noviembre de 2021

Resumen

En el presente proyecto se desarrollará un programa de visión artificial para el reconocimiento y clasificación de frutas (bananas, naranjas, limones y tomates). En primer lugar, se procesarán las distintas imágenes de frutas de una base de datos. De este procesamiento, mediante **opencv**, obtendremos de cada imagen distintos parámetros como color, contornos, áreas, perímetros, momentos de áreas, etc. Con la información obtenida de todos los parámetros del conjunto de entrenamiento se programaran 2 algoritmos de clasificación **K-means** y **K-nn** con el fin de poder determinar cual es el mejor para esta tarea. Obteniendo muy buenos resultados, siendo la tasa de aciertos del **95 %** para el algoritmo **k-nn** y de **91 %** para el algoritmo **k-means**.

Pudiendo concluirse que el algoritmo **k-nn** funciona mejor para datos dispersos o en donde no hay límites bien marcados a diferencia del algoritmo **k-means** el cual disminuye su performance para este tipo de datos.

1. Introducción

El problema a resolver consiste en el diseño de un sistema de reconocimiento de frutas por visión artificial con la intención de agilizar el proceso de cobro en las cajas de un supermercado. Para ello se desarrolló un agente prototipo que pueda reconocer bananas, naranjas, limones y tomates a partir de fotografías (fig.1)

Figura 1: Frutas a clasificar: banana, naranjas, limones y tomates.

Partiendo de esta base se creó una base de datos con fotos tomadas para cada una de estas frutas, pero antes de entrar en detalles sobre el desarrollo del programa primero explicaremos un poco qué es la visión artificial.
La visión artificial es una disciplina científica que incluye métodos para adquirir, procesar, analizar y comprender las imágenes del mundo real con el fin de producir información numérica o simbólica para que puedan ser tratados por un ordenador. Tal y como los humanos usamos nuestros ojos y cerebros para comprender el mundo que nos rodea, la visión artificial trata de producir el mismo efecto para que los ordenadores puedan percibir y comprender una imagen o secuencia de imágenes y actuar según convenga en una determinada situación. Esta comprensión se consigue gracias a distintos campos como la geometría, la estadística, la física y otras disciplinas. Para este procesamiento de imágenes se utilizó una biblioteca de código abierto llamada OpenCV (Open Source Computer Vision Library) con la que se pudo hacer el tratamiento de las imágenes (fig. 2).

El proceso de la visión artificial consiste en procesar las imágenes aplicando distintos filtros para obtener diferentes versiones de una misma foto y con esas imágenes poder calcular parámetros numéricos que representen cada una de ellas. Estos parámetros luego son puestos en vectores para poder interpretarlos y analizarlos con otros algoritmos de IA.

Las etapas de la visión artificial son las siguientes:

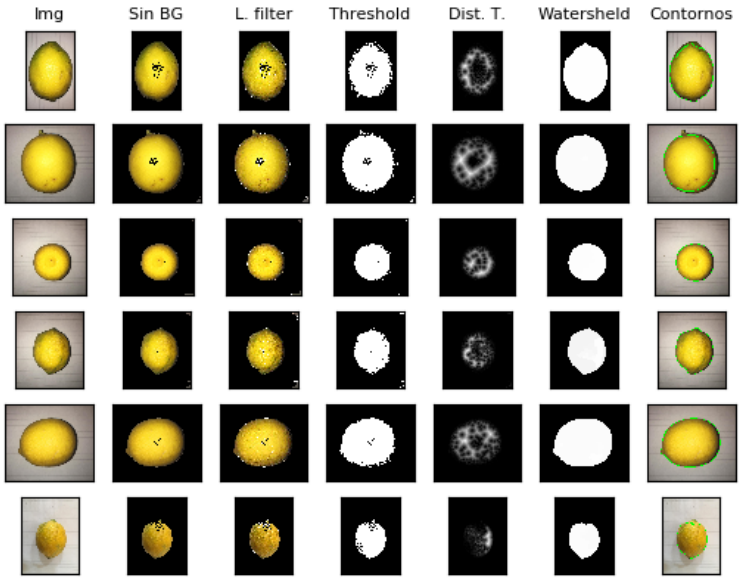


Figura 2: Procesamiento de las imágenes de limones.

- **Adquisición y digitalización**: Es el proceso de capturar una imagen y pasarla a algún formato digital, se utiliza una cámara para capturar la escena y después se envía a una unidad donde pueda ser procesada.
- **Procesamiento previo:** La cámara normalmente captura ruido, por lo que debe de haber un pre-procesamiento para eliminar dicho ruido de la imagen mediante una gran variedad de filtros. En ocasiones también deben hacérsele transformaciones geométricas como recortes o rotaciones, esto también se hace en esta etapa.
- **Obtención de características:** En esta etapa haremos resaltar las características de la imagen que son de nuestro interés. Las operaciones que comúnmente se realizan aquí son: detección de esquinas, colores, flujo óptico y formas, realce de bordes, entre otros. Aquí es donde se debe utilizar un buen software para todo este tipo de operaciones.
- **Clasificación:** Acá se decide a que categoría pertenece o es más probable que pertenezca una cierta observación (o percepción) según su vector de características.
- **Interpretación:** La clasificación obtenida anteriormente se organiza en una estructura adecuada para mostrar los resultados.

Ahora que tenemos una idea de lo que se trata la visión artificial explicaremos un poco como se representa una imagen de manera digital.
Una imagen digital está compuesta de un número finito de elementos y cada uno tiene una localidad y un valor particular. A estos elementos se les llama puntos elementales de la imagen o píxeles, siendo este último el término comúnmente utilizado para denotar la unidad mínima de medida de una imagen digital. En la Figura 3 se muestra una representación de una imagen con 256 niveles de intensidad. En ella, cada uno de los píxeles está representado por un número entero que es interpretado como el nivel de intensidad luminosa en la escala de grises. Ampliando la imagen en una zona cualquiera, se pueden apreciar estos valores, que se muestran en forma de matriz en la misma figura, correspondiéndose cada elemento de la matriz Nij con las coordenadas en el plano x=i, y=j.

El fundamento para describir una imagen digital en color es el mismo que el expuesto anteriormente, con la salvedad de que cada elemento o píxel es descrito y codificado de otra forma, según el espacio de color que se esté utilizando. Así por ejemplo, para un espacio de color RGB (generalmente el más usado para representar imágenes), se representa cada píxel como un color creado a partir de ciertas cantidades de los colores rojo, verde y azul. Esta representación se puede interpretar como una matriz de tres niveles de intensidad, donde cada nivel corresponde a la intensidad de color de las componentes rojo, verde y azul, como se muestra en la Figura 4.

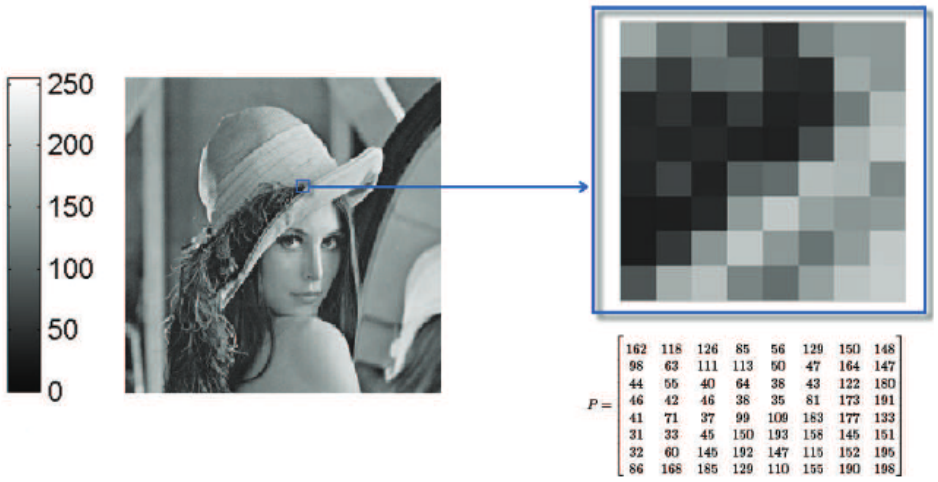


Figura 3: Imagen con 256 niveles de intensidad y representación numérica de un fragmento 8x8.

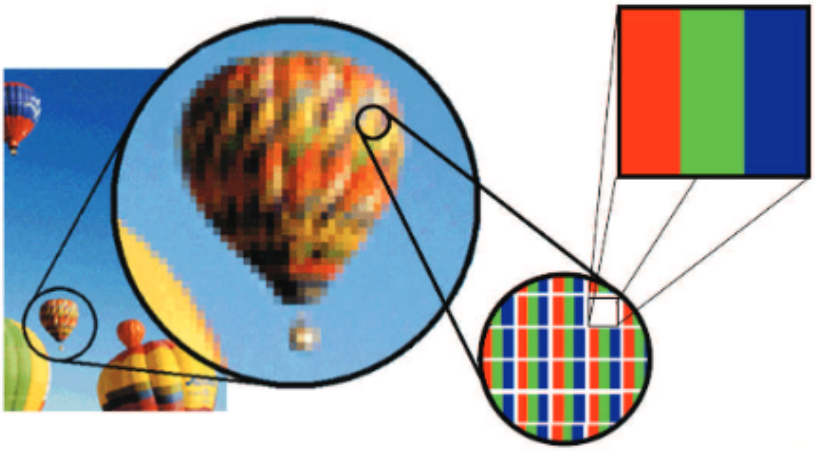


Figura 4: Componentes primarias en los píxeles de una imagen en color.

La visión artificial explicada anteriormente es solo una parte del problema y se limita al correcto tratamiento de las imágenes para la obtención de buenos parámetros característicos. La otra parte del problema es poder interpretar los parámetros de una imagen desconocida y comparándolo con los parámetros obtenidos, poder decir a que grupo de frutas pertenece. Para esta segunda parte se utiliza el mismo procesamiento de visión artificial y algoritmos de clasificación que comparan los nuevos datos con los datos del conjunto de entrenamiento para así poder dar un resultado. El resultado que se espera es que dada una imagen se compare con la base de datos y los algoritmos puedan decir con un cierto porcentaje de certeza que fruta es.

2. Especificación del agente

2.1. Especificación del entorno de trabajo

En primer lugar vamos a identificar el entorno en donde va a funcionar nuestro agente. Este se va a encontrar en una caja de supermercado, cuyo elemento de entrada va a ser una cámara apuntando a la cinta transportadora y queremos poder reconocer cuando pase alguna fruta de 4 tipos diferentes, para luego, poder contarla dentro de los productos que va a comprar la persona. Teniendo en cuenta lo dicho anteriormente especificaremos nuestra tabla **REAS** (Rendimiento, Entorno, Actuadores, Sensores) siendo esto factores claves para el diseño correcto de nuestro agente:

- **Rendimiento**: define el criterio de éxito del agente. Obviamente, alguno de estos objetivos entran en conflicto por lo que habrá que llegar a acuerdos.

- **Entorno**: el conocimiento del medio, o lo que se sabe del entorno que lo rodea. Cuanto más restringido esté el entorno, más fácil será el problema del diseño.
- **Actuadores**: elementos que permiten llevar a cabo acciones al agente.
- **Sensores**: elementos que permiten conocer el estado del entorno en cada instante.

Tipo de agente	Rendimiento	Entorno	Actuadores	Sensores
Sistema de Reconocimiento de Frutas	-Rápido -Seguro -Eficaz -Precisión	-Caja de supermercado -Entrada y salida de productos -Cajero/a -Cliente	- Cinta -Flash para mejorar la iluminación	Cámara

Tabla 1: Descripción REAS del entorno de trabajo del sistema de reconocimiento de frutas.

2.2. Propiedades del entorno de trabajo

Una vez especificado el entorno en el que trabajará nuestro agente identificaremos un pequeño número de dimensiones o propiedades en las que categorizar estos entornos. Estas dimensiones determinan, hasta cierto punto, el diseño más adecuado para el agente y la utilización de cada una de las familias principales de técnicas en la implementación del agente. Estas dimensiones son:

- **Totalmente o parcialmente observable:** Es un entorno totalmente observable si los sensores proporcionan toda la información relevante.
- **Determinista o estocástico:** Es un entorno determinista si el estado siguiente se obtiene a partir del actual y de las acciones del agente.
- **Episódico o secuencial:** Cada episodio consiste en la percepción del agente y la realización de una única acción posterior. No depende de las acciones previas.
- **Estático o dinámico:** Un entorno dinámico puede sufrir cambios mientras el agente esta razonando.
- **Discreto o continuo:** En un entorno discreto existe un número concreto de percepciones y acciones claramente definidas.
- **Agente individual o multiagente:** Una entidad del entorno se considera agente si su comportamiento se describe mejor por la maximización de una medida de rendimiento cuyo valor depende de otro agente.

En la tabla 2 se muestran las propiedades del entorno de trabajo.

Entorno de trabajo	Observable	Determinista	Episódico	Estático	Discreto	Agente
Sistema de Reconocimiento de Frutas	Totalmente	Si	Si	Si	Si	Individual

Tabla 2: Propiedades del entorno de trabajo.

2.3. Tipo de agente

Sabiendo esto, programaremos un **agente que aprende**. Este se puede dividir en cuatro componentes conceptuales, tal y como se muestra en la Figura 5 . En nuestro caso, el **elemento de aprendizaje** será nuestra base de datos y los algoritmos **K-means** (**aprendizaje no supervisado**) y **K-nn** (**aprendizaje supervisado**), el **elemento de actuación** será aquel que realizará una acción en función de la salida del **elemento de aprendizaje**, **crítica** no tendremos y **generador de problemas** tendremos en nuestro agente basado en el algoritmo **K-means** ya que se ha mejorado y se ha combinado con un algoritmo de búsqueda local, el temple simulado más específicamente, para buscar una semilla que me lleve a la mejor clasificación posible. La distinción

más importante entre el elemento de aprendizaje y el elemento de actuación es que el primero está responsabilizado de hacer mejoras y el segundo se responsabiliza de la selección de acciones externas. El elemento de actuación es lo que recibe estímulos y determina las acciones a realizar. El elemento de aprendizaje se realimenta con las críticas sobre la actuación del agente y determina cómo se debe modificar el elemento de actuación para proporcionar mejores resultados en el futuro. La crítica indica al elemento de aprendizaje qué tal lo está haciendo el agente con respecto a un nivel de actuación fijo. La crítica es necesaria porque las percepciones por sí mismas no prevén una indicación del éxito del agente. El último componente del agente con capacidad de aprendizaje es el generador de problemas. Es responsable de sugerir acciones que lo guiarán hacia experiencias nuevas e informativas.

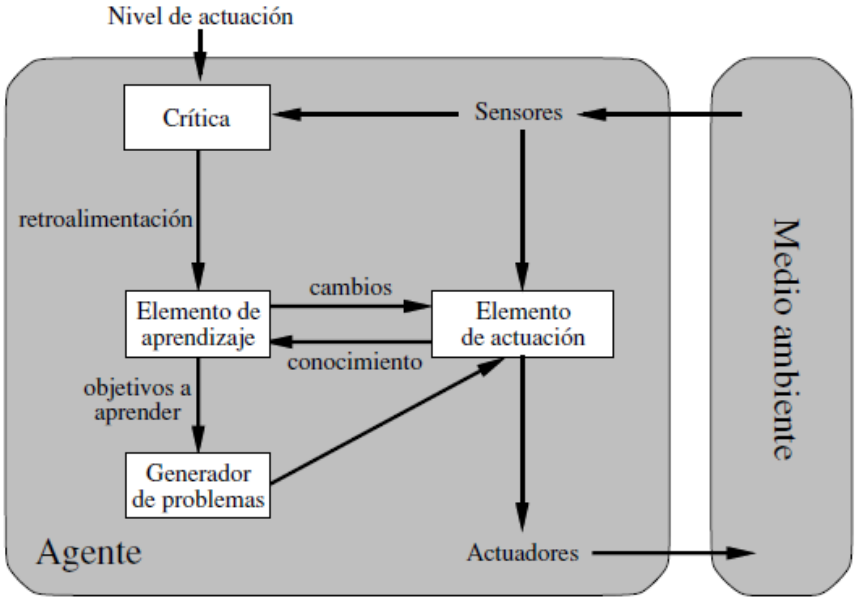


Figura 5: Modelo general para agentes que aprenden.

3. Diseño del agente

3.1. Etapas del agente

Ahora explicaremos cada una de las etapas de nuestro agente y como fue su desarrollo. Estas consisten en:

- **Conformar la base de datos:** Consiste en hacer una búsqueda de imágenes representativas para el conjunto de entrenamiento.

  En primer lugar se buscaron algunas imagenes en internet y luego se tomaron fotos propias, se buscó obtener asi un buen numero de ejemplos con fotos variadas con diferentes fondos y frutas con distintas características. En la figura 6 se aprecian todas las imagenes en la base de datos de las bananas, como se puede observar hay fotos tanto de internet como de propia autoria de distintas formas y fondos se busca asi tener una base de datos abarcativa.

- **Procesamiento de las imágenes:** Una vez obtenida nuestra base de datos procedimos al acondicionamiento de estas. Esto consistió en aplicarle una serie de filtros a nuestras imágenes. A continuación explicaremos el procedimiento guiándonos con la figura 7:

  1. En primer lugar se aplicó un Filtro Laplaciano (**L. filter**). El cual permite enfatizar los bordes de la imagen. La ecuación de dicho filtro es la siguiente:

  $$g(x,y) = f(x,y) + c \cdot \bigtriangledown^2 f(x,y) \tag{1}$$

  Donde: $g(x,y)$ es la imagen mejorada y $f(x,y)$ es la imagen original.

  2. Se procede a eliminar el fondo de la imagen volviendolo negro (**Sin BG**).

Figura 6: Base de datos bananas.

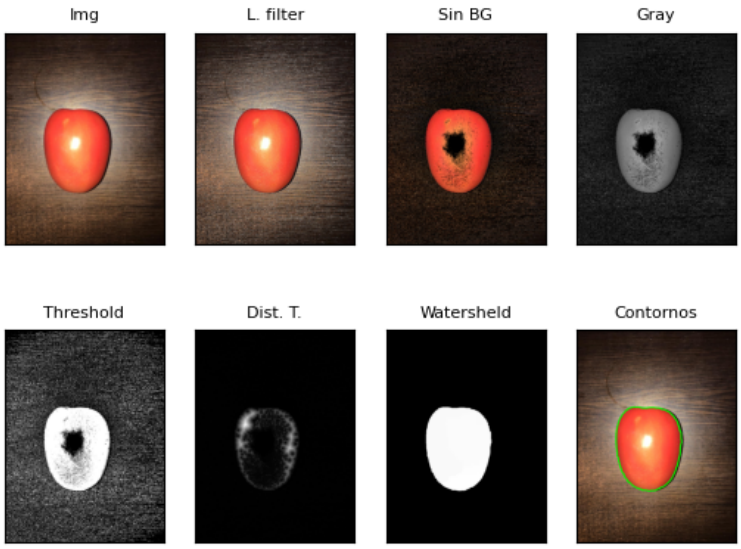


Figura 7: Procesamiento de las imágenes.

3. La imagen se convierte a escala de grises (**Gray**).
4. Se aplica el método **Otsu's Binarization** que a diferencia del global thresholding, en el cual utilizamos un valor elegido arbitrario como umbral, el método de Otsu evita tener que elegir un valor y lo determina automáticamente, por ejemplo si consideramos una imagen con solo dos valores de color distintos (imagen bimodal), donde el histograma solo constaría de dos picos. Un buen umbral estaría en el medio de esos dos valores. De manera similar, el método de Otsu determina un valor de umbral global óptimo a partir del histograma de la imagen.(**Threshold**)

5. Aplicamos el método **(**Distance Transform), el cual nos permite obtener los picos de la imagen, a partir de cuales podemos realizar una segmentación de la imagen basada en marcadores utilizando el watershed algorithm.(**Dist. T**)
6. Por último realizamos el **watershed algorithm**, el cual nos permite obtener los contornos de la fruta. (**Watershed**)

- **Vectorización:** Consiste en obtener parámetros o propiedades características de cada imagen procesada, guardarla en una base de datos, filtrado (de los más importantes) y normalización.

  Una vez acondicionada las imágenes se pudieron marcar los contornos y obtener el área de la fruta, su perímetro y distintas figuras que encerraran la fruta como un rectángulo, rectángulo de mínima área (es un rectángulo rotado), círculo de mínima inclusión y una elipse, como se aprecia en la figura 8.

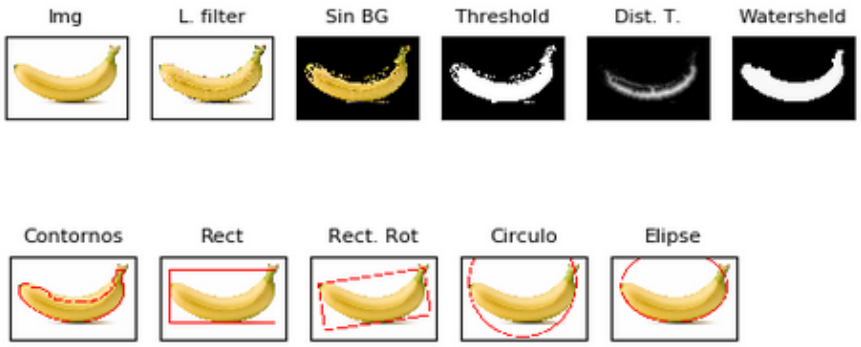
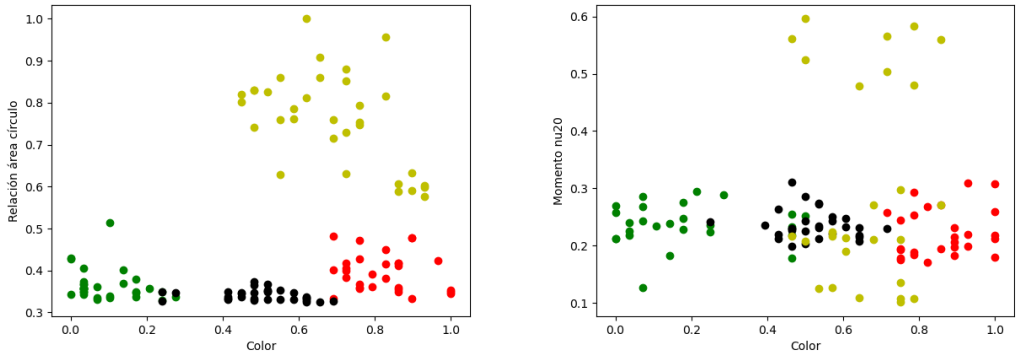


Figura 8: Características extraidas de una fruta.

El desafío principal de esta etapa es poder procesar las imágenes y obtener buenos parámetros que representen de la mejor forma posible las 4 clases de frutas y que puedan ser diferenciadas fácilmente. Por ende, procedimos a filtrar los datos que obtuvimos de la imagen los cuales fueron cinco:

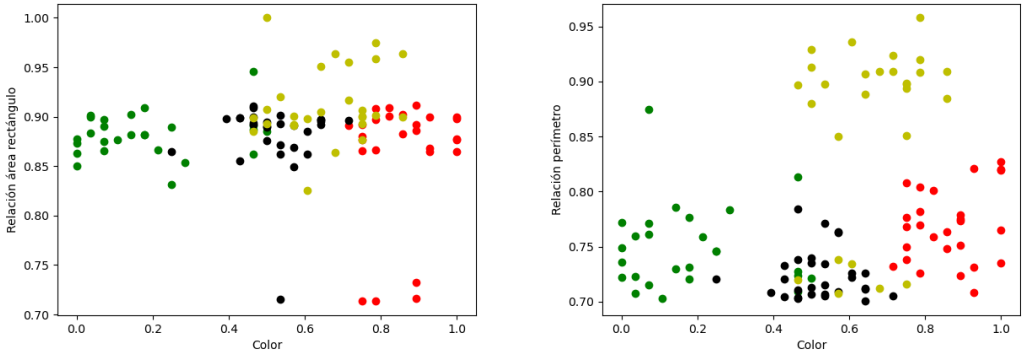
1. **Color**: para lo cual se convirtió a HSV para separar más fácil en una escala las gamas de colores. Se toma en cuenta solo el canal H.
2. **Relación área rectángulo rotado - área fruta** : Se calcula el cociente entre el área de la fruta y el área del rectángulo mínimo que encierra la fruta.
3. **Relación área círculo - área fruta**: mismo que el anterior, pero respecto del área de un círculo que encierra a la fruta.
4. **Relación perímetro círculo - perímetro fruta (Redondez)**: Se calcula el cociente entre el perímetro de la fruta y el perímetro del círculo que encierra a la fruta.
5. **Momentos**: (equivalente a momento de inercia, pero 2D) que fueron sacados en diferentes direcciones y están todos normalizados tanto en módulo como respecto del centroide de la fruta.

El color fue un parámetro clave para diferenciar las frutas, el único inconveniente es que tenemos dos frutas con el mismo color, limones y bananas, por lo que necesitamos al menos un parámetro más para poder realizar una correcta clasificación. Para poder saber cuales son los mejores parámetros se graficaron todos los parámetros antes mencionados contra el color como se aprecia en la figura 9. Como podemos apreciar los dos parámetros que mejor representan a los distintos grupos son el **Color** y la **Relación área círculo - área fruta**.



(a) Relación área círculo

(b) Momento nu20



(c) Relación área rectángulo

(d) Relación perímetro

Figura 9: Evaluación de los distintos parámetros.

- **Procesamiento de los datos de prueba:** Se realiza el mismo procesamiento de imagen con una foto, en principio, desconocida.
- **Vectorización de los elementos de prueba:** Vectorizar y normalizar los parámetros de la imagen de prueba.
- **Clasificación:** Se clasifica la imagen nueva utilizando los algoritmos **K-means** y **Knn**.
  - **K-means:** Recibe los parámetros y arma K grupos que tengan características parecidas. Luego de hacer este agrupamiento en K grupos se fija qué grupo está más cerca de los parámetros de la imagen dada y la clasifica en el grupo más cercano.

    Dado que el algoritmo **K-means** presenta el problema de que converge a óptimos locales, y su solución depende del estado inicial del algoritmo se realizó una mejora en este, se implementó el algoritmo de **Temple simulado** para buscar el mejor estado inicial al cual el algoritmo converge al óptimo global. En la figura 10 se observa la evolución de la energía del **Temple Simulado**.

    Así se obtiene como resultado la clasificación que se observa en la figura 11, siendo la base de datos la que se observa en la figura 12. Vemos que la clasificación realizada es muy satisfactoria.

    A continuación explicaré como funciona el algoritmo **K-means** con **Temple Simulado**:

    1. Se da una clasificación aleatoria inicial o estado inicial.
    2. Se calculan los centroides y la clasificación en función del algoritmo **K-means**.
    3. Se calcula la energía de la solución en función de la distancia de cada punto de la base de datos a su centroide.
    4. Se calcula un estado vecino.
    5. Se calcula la energía de este estado.
    6. Se compara la energía del nuevo estado con la del estado actual. Si es menor se toma el nuevo estado, si es mayor se puede tomar o no según una probabilidad en función de la temperatura.

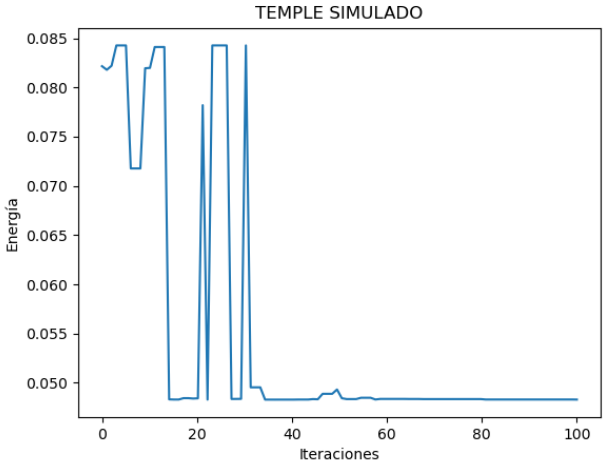


Figura 10: Temple simulado.

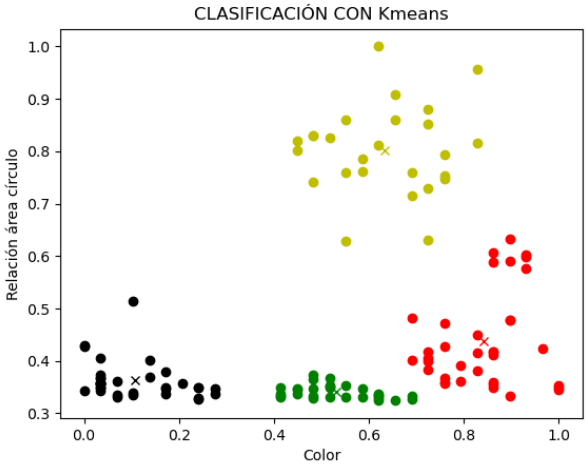


Figura 11: Clasificación con **K-means**.

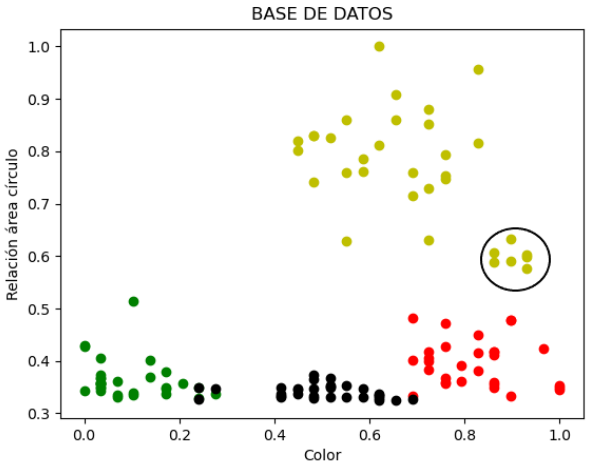


Figura 12: Base de datos.

7. Se repite desde 4 hasta que el algoritmo converja o se realicen un número de iteraciones establecido.

De esta manera se le busca dar cierta estocacidad al algoritmo **Kmeans**, ya que sin esta, este tiende a estancarse en mínimos locales por su comportamiento semejante al algoritmo **Hill Climbing**

- **Knn:** El algoritmo Knn trabaja con la misma base de datos pero parte de los parámetros de la imagen desconocida y busca los vecinos mas cercanos. Por lo tanto el algoritmo **Knn** es un algoritmo supervisado, a diferencia del **K-means** que es no supervisado. En la figura 13 se puede apreciar como funciona este algoritmo. Del punto dado por los parámetros de la imagen desconocida (X negra) buscamos k vecino y si esos vecinos son en su mayoría limones, por ejemplo, el algoritmo clasifica la imagen como un limón.

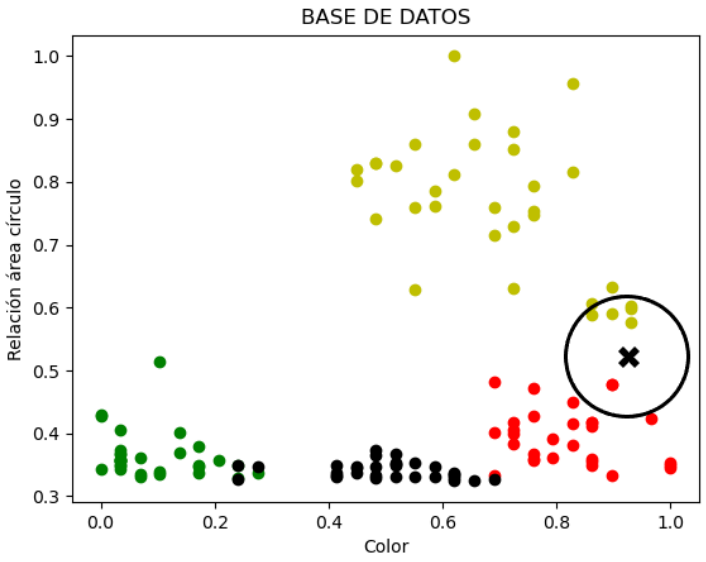


Figura 13: Algoritmo **Knn**.

- **Conclusión:** Luego de hacer la clasificación el algoritmo entrega los resultados.

4. Planificación del Diseño del Agente

Las tareas para diseñar el agente son:

- Buscar un buen conjunto de imágenes representativas de cada grupo.
- Buscar qué filtros son fundamentales para obtener los parámetros que se necesitan.
- Hacer el programa de procesamiento de imágenes.
- Mostrar los resultados del procesamiento para supervisarlos.
- Probar todas las imágenes y ajustar parámetros para obtener el mejor procesamiento posible.
- Obtener los parámetros finales y guardarlos en archivos, para formar una Base de Datos. Hay que vectorizar la información y normalizar los vectores.
- Diseñar el algoritmo **K-means** y programarlo, pasarle el conjunto de entrenamiento (base de datos) y verificar resultados con el conjunto de testeo. Ajustar parámetros para mejorar resultados.
- Diseñar el algoritmo **K-nn** y programarlo, pasarle el conjunto de entrenamiento (base de datos) y verificar resultados con el conjunto de testeo. Ajustar parámetros para mejorar resultados.

A continuación se presenta la tabla de actividades y tiempos:

N° Actividad	Descripción	Duración (Días)
1	Búsqueda de imágenes de conjunto de entrenamiento	1
2	Búsqueda de filtros principales y ajuste parámetros.	5
3	Programar procesamiento de imágenes.	5
4	Mostrar los resultados y filtrar los más representativos, luego vectorizar y normalizar los parámetros obtenidos	2
5	Programar algoritmo **K-means**.	1
6	Programar algoritmo **K-nn**.	1
7	Realizar pruebas y ajustar parámetros.	1

Tabla 3: Actividades y tiempos.

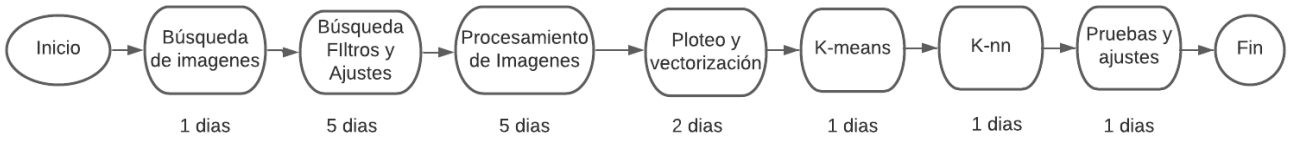


Figura 14: Actividades en el orden que se realizaron y su duración.

Si representamos estas actividades en una forma que nos permita encontrar el camino crítico, el diagrama queda como el que se muestra en la figura 15.

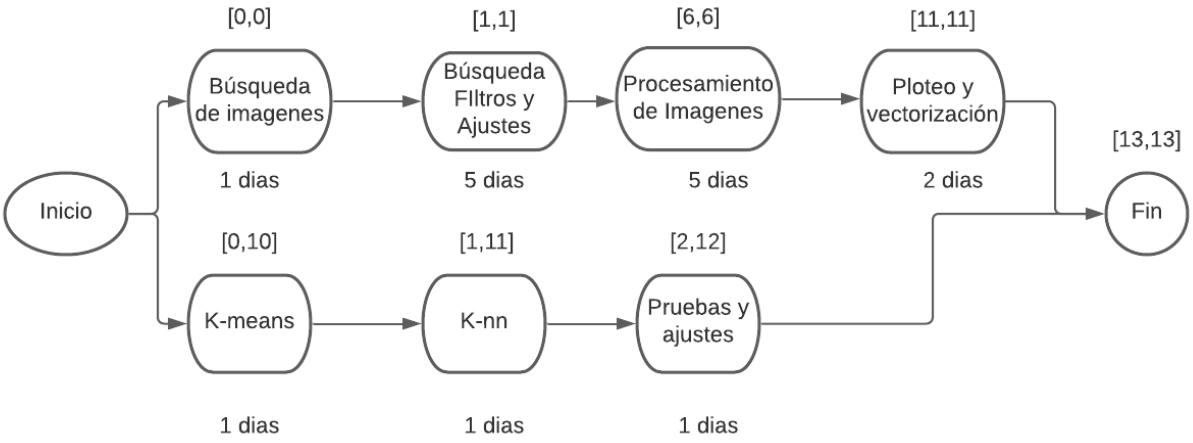


Figura 15: Actividades que se pueden desarrollar en paralelo o que no dependen de las otras en un principio.

Así el camino crítico y los intervalos de relajación quedan como en la figura 16.

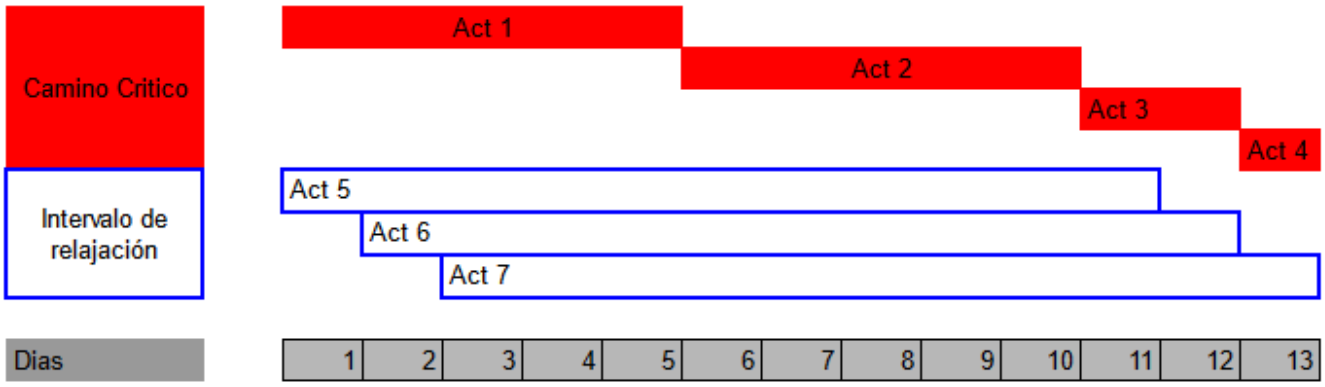


Figura 16: Actividades que representan el camino crítico y los intervalos de relajación.

5. Código

A continuación se explicará el código desarrollado. En la figura 17 se puede observar un diagrama de clase del código el cual intenta hacer más fácil su explicación.

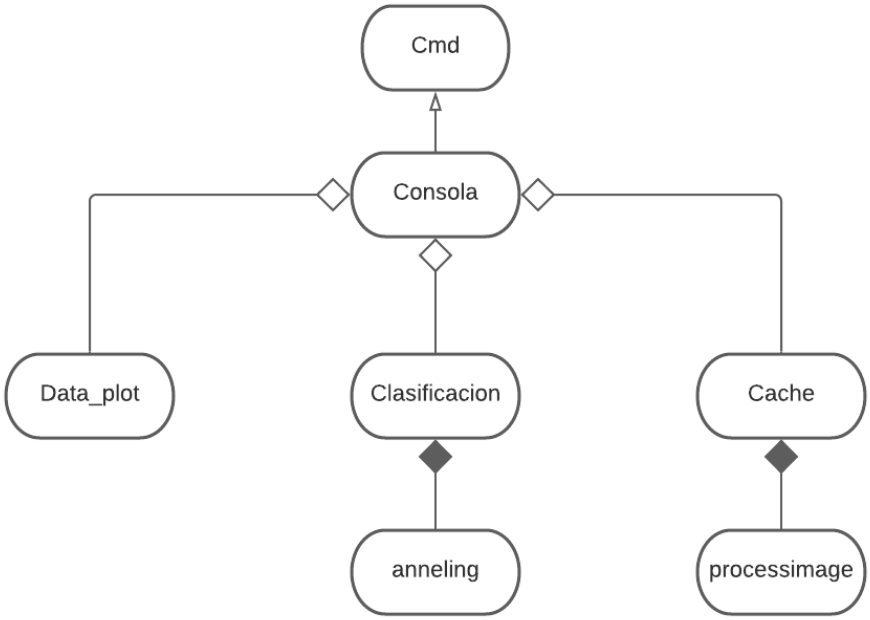


Figura 17: Diagrama de clases del código.

A continuación se explicará cada clase:

- **Cmd**: Es una clase de la librería **cmd** que proporciona un marco simple para escribir intérpretes de comandos orientados a líneas.
- **Consola**: Clase que hereda de **cmd** que se creó con el fin de crear un intérprete de comando para facilitar el uso de la aplicación. A continuación se muestra el código:

```
class Consola(Cmd):

    panelesCreados = 0
    doc_header = 'Comandos documentados'

    def __init__(self):···

    def do_Prueba(self,args):···

    def do_leerbd(self,args):···

    def do_plotbd(self,args):···

    def do_plotdatos(self,args):···

    def do_foto(self,args):···

    def do_camera(self,args):···

    def default(self, args):···

    def precmd(self, args):···

    def do_exit(self,args):···
```

Figura 18: Métodos de la clase **Consola**.

```
def __init__(self):

    Cmd.__init__(self)
    path=path_dir()
    self.c=Cache(path)
    T=ley_enfriamiento(1000,100,1.3)
    temple=anneling(T)
    self.cls=clasificacion(temple)
    self.dd=data_plot()
    try:
        self.cls.data_accond(self.c.data)
    except:
        self.c.leer_imagenes()
        self.cls.data_accond(self.c.data)

def do_Prueba(self,args):

    """Realiza la clasificación de los datos de prueba con los algoritmos K-means y Knn"""

    path=path_dir()
    path=path+rf'\prueba'
    c2=Cache(path)
    c2.leer_imagenes()
    self.cls.class_pruebas(c2.data,c2.tabla)

def do_leerbd(self,args):

    """Lee la base de datos"""

    self.c.leer_imagenes()
    self.cls.data_accond(self.c.data)
```

Figura 19: Detalles de la clase **Consola**. Métodos **Prueba** y **leerbd**

```
def do_plotbd(self,args):

    """Plotea la base de datos"""

    self.c.leer_imagenes()
    self.cls.data_accond(self.c.data)
    self.dd.plotimage(self.c.tabla,5)

def do_plotdatos(self,args):

    """Plotea parámetros caraterísticos de la base de datos"""

    self.dd.plotdata(self.c.data,self.cls)

def do_foto(self,args):

    """Clasifica una foto dada con los algoritmos K-means y Knn"""

    self.cls.class_foto(args)

def do_camera(self,args):

    """"Clasificación de frutas mediante la camera"""
    self.cls.kmeans_init(plot=False)
    cap = cv.VideoCapture(1)
    if not cap.isOpened:
        print('--(!)Error opening video capture')
        exit(0)
    while True:
```

Figura 20: Detalles de la clase **Consola**. Métodos **plotbd**, **plotdatos**, **foto** y **camera**.

```
        ret, frame = cap.read()
        fr=frame.copy()
        d,b,x,y,w,h=self.cls.class_foto(frame,camera=True)
        text_=f'K-means:{d}; Knn:{b}'
        cv.rectangle(fr,(x,y),(x+w,y+h),(0,255,0),2)
        cv.putText(fr,text_,(x,y),0,0.5,(0,255,0),2)
        cv.imshow('Clasificador de frutas', fr)

        if frame is None:
            print('--(!) No captured frame -- Break!')
            break
        if cv.waitKey(10) == 27:
            break

def default(self, args):

    self.bool=0
    print("Error. El comando \'" + args + "\' no esta disponible")

def precmd(self, args):

    print("-------------------------------------------------------")
    return(args)

def do_exit(self,args):

    """"Terminar consola"""

    raise SystemExit
```

Figura 21: Detalles de la clase **Consola**.

- **data_plot**: Esta clase busca plotear los distintos datos que se manejan en el programa, como se ve en la figura 17 esta es agregada por la clase **consola**. A continuación se detalla el código de esta.

```
class data_plot():

    def __init__(self):···

    def matplotlib_multi_pic1(self,s):···

    def plt_basedatos(self,s):···

    def plotdata(self,data,cls):···

    def plotimage(self,tabla,n):···
```

Figura 22: Métodos de la clase **data_plot**.

```
def __init__(self):
    pass

def matplotlib_multi_pic1(self,s):

    """Función para plotear el tratamiento de las imagenes"""

    titles=['Img','L. filter','Sin BG','Gray','Threshold','Dist. T.','Watersheld','Contornos']
    aa=0
    for i in s:
        bb=0
        for j in i:
            title=titles[bb]
            plt.subplot(len(s),len(i),aa+1)
            j=cv.cvtColor(j,cv.COLOR_BGR2RGB)
            plt.imshow(j)
            if aa<len(i):
                plt.title(title,fontsize=8)
            plt.xticks([])
            plt.yticks([])
            aa+=1
            bb+=1
    plt.show()
```

Figura 23: Detalles de la clase **data_plot**. Método **matplotlib_multi_pic1**.

```
def plotdata(self,data,cls):

    """Plotea parámetros característicos de los datos"""

    color=['or','go','ko','yo','ob']
    for i in range(len(cls.x_T)):
        plt.plot(cls.x_T[i],cls.y_T[i],color[cls.cluster_T[i]])

    plt.xlabel("Color")
    plt.ylabel("Relación área círculo")
    plt.show()

    z_T=np.array(data['nu02'])
    z_T=z_T/max(z_T)
    for i in range(len(cls.x_T)):
        plt.plot(cls.x_T[i],z_T[i],color[cls.cluster_T[i]])

    plt.xlabel("Color")
    plt.ylabel("Momento nu20")
    plt.show()

def plotimage(self,tabla,n):

    """Ploteamos el procesamiento de las imagenens"""

    for i in tabla['imagen'].keys():
        for j in range(round(len(tabla['imagen'][i])/n)):
            self.matplotlib_multi_pic1(tabla['imagen'][i][j*n:(j+1)*n])
```

Figura 24: Detalles de la clase **data_plot**. Métodos **plotdata** y **plotimage**.

- **clasificacion**: Esta clase fue creada para el acondicionamiento de datos y la clasificación de datos mediante los algoritmos **K-means** y **Knn**. Esta es agregada por la clases **consola** y está compuesta por la clase **anneling**.

```
class clasificacion():

    def __init__(self,temple):···

    def data_accond(self,data):···

    def kmeans_init(self,plot=False):···

    def class_pruebas(self,data,tabla):···

    def class_foto(self,name,camera=False):···
```

Figura 25: Métodos de la clase **clasificacion**.

```python
def __init__(self,temple):

    self.kn=4
    self.temple=temple
    self.dd=data_plot()
    self.pr=processimage()

def data_accond(self,data):

    """Acondicionamiento de los datos almacenados en Cache"""

    #x_T & y_T: Lista con la info de todas las frutas
    #cluster_T: Lista con la info de que fruta representa cada dato
    self.x_T=[]
    self.y_T=[]
    self.cluster_T=[]
    ii=0
    for j in data['data_x'].keys():
        self.x_T.extend(data['data_x'][j])
        self.cluster_T.extend([ii for i in range(len(data['data_x'][j]))])
        self.y_T.extend(data['data_y'][j])
        ii=ii+1

    self.x_T=np.array(self.x_T)
    self.y_T=np.array(self.y_T)
    self.x_Tmax=max(self.x_T)
    self.y_Tmax=max(self.y_T)

    #Normalizamos los datos
    self.x_T=self.x_T/self.x_Tmax
    self.y_T=self.y_T/self.y_Tmax
```

Figura 26: Detalles de la clase **clasificacion**. Método **data_accond**.

```python
def kmeans_init(self,plot=False):

    """Realiza la clusterización de la base de datos mediante el algoritmo Kmeans"""

    #Cluster inicial
    self.cluster=rnd.choices(range(4),k=len(self.x_T))
    self.cluster,list_energy=self.temple.search(self.cluster,self.x_T,self.y_T,self.kn)
    #Aplicamos kmeans
    self.cluster,self.x_cluster,self.y_cluster=kmeans(self.x_T,self.y_T,self.cluster,kn=self.kn)

    if plot:
        #Ploteamos las imagenes vectorizadas
        print('BASE DE DATOS')
        self.color=['or','go','ko','yo','ob']
        for i in range(len(self.x_T)):
            plt.plot(self.x_T[i],self.y_T[i],self.color[self.cluster_T[i]])
        plt.xlabel('Color')
        plt.ylabel('Relación área círculo')
        plt.title('BASE DE DATOS')
        plt.show()
        print("APLICACIÓN DE TEMPLE SIMULADO")
        plt.plot(np.linspace(0,len(list_energy),len(list_energy)),list_energy)
        plt.xlabel('Iteraciones')
        plt.ylabel('Energía')
        plt.title('TEMPLE SIMULADO')
        plt.show()
```

Figura 27: Detalles de la clase **clasificacion**. Método **kmeans_init**.

```python
def class_pruebas(self,data,tabla):

    """Clasificamos imagenes de prueba mediante kmeans y knn"""

    self.kmeans_init(plot=True)
    for i in data['data_x'].keys():
        s1_list=[]
        s2_list=[]
        b1_list=[]
        b2_list=[]
        for j,l in zip(data['data_x'][i],data['data_y'][i]):
            s1,b1=kmeans_class([j,l],self.x_cluster,self.y_cluster,self.cluster,self.cluster_T,self.x_Tmax,self.y_Tmax)
            s2,b2=knn(5,[j,l],self.x_T,self.y_T,self.cluster_T,self.x_Tmax,self.y_Tmax)
            s1_list.append(s1)
            s2_list.append(s2)
            b1_list.append(b1)
            b2_list.append(b2)

        sep = '|{}|{}|{}|{}|{}|\n'.format('-'*18,'-'*28,  '-'*10,  '-'*18,  '-'*10)
        rep=i+'\n'
        rep=rep + str('{0}|    Prueba nº     |           Kmeans          |    %    |       Knn       |    %    |\n{0}'.format(sep))
        for n,s11,b11,s22,b22  in zip(range(len(s1_list)),s1_list,b1_list,s2_list,b2_list):
            rep=str(rep)+'| {0:^15d}  | {1:^26s} | {2:^8s} | {3:^16s} | {4:^8s} | \n{5}'.format(n,s11,b11,s22,b22,sep)
        print( str(rep))

    self.dd.plotimage(tabla,1)
```

Figura 28: Detalles de la clase **clasificacion**. Método **class_pruebas**.

```python
def class_foto(self,name,camera=False):

    """Clasificamos una imagen de prueba mediante kmeans y knn"""

    if camera:
        self.pr.process(name,foto=False)
    else:
        self.kmeans_init(plot=False)
        path, _ = os.path.split(os.path.abspath(__file__))
        self.pr.process(path+rf'\{name}')
    [x,y,a,a2,a3]=self.pr.data()
    s1,b1=kmeans_class([x,y],self.x_cluster,self.y_cluster,self.cluster,self.cluster_T,self.x_Tmax,self.y_Tmax)
    s2,b2=knn(5,[x,y],self.x_T,self.y_T,self.cluster_T,self.x_Tmax,self.y_Tmax)

    if camera:
        return s1+b1,s2+b2,self.pr.x,self.pr.y,self.pr.w,self.pr.h
    else:
        print('Kmeans:  ',s1,b1)
        print('Knn:  ',s2,b2)
```

Figura 29: Detalles de la clase **clasificacion**. Método **class_foto**.

- **Cache**: Esta clase esta compuesta por la clase **processimage** mediante la cual procesa las imagenes y las vectoriza, y realiza el almacenamiento de estos datos en un archivo **.json**.

```python
class Cache():

    def __init__(self,path,archivo=True,foto=False):···

    def crear(self,name,imagen=False):···

    def leer_imagenes(self):···
```

Figura 30: Métodos de la clase **Cache**.

```python
class Cache():

    def __init__(self,path,archivo=True,foto=False):

        self.path=path
        self.archivo=archivo
        self.Foto=foto
        try:
            if archivo:
                tf = open(self.path+"/cache.json", "r")
                self.data = json.load(tf)
                self.tabla={}
                tf.close()
            else:
                self.tabla={}
                self.data = {}

        except :
            self.tabla={}
            self.data = {}

        self.frutas=['limon','tomate','naranja','banana']
```

Figura 31: Detalles de la clase **Cache**.

```python
def crear(self,name,imagen=False):

    """Procesa las imagenes y crea un archivo .json para guardar los parámetros"""

    pr=processimage()
    contenido = os.listdir(self.path+rf'\{name}')
    data_fruta_x_dict={}
    data_fruta_y_dict={}
    img_dict={}
    img=[]
    data_fruta_x=[]
    data_fruta_y=[]
    rel_nu=[]
    rel_ret=[]
    rel_per=[]

    if len(self.tabla) != 0 and imagen:

        img_dict=self.tabla['imagen']

    if len(self.data) != 0:

        data_fruta_x_dict=self.data['data_x']
        data_fruta_y_dict=self.data['data_y']
        rel_nu=self.data['nu02']
        rel_ret=self.data['ret']
        rel_per=self.data['per']
```

Figura 32: Detalles de la clase **Cache**. Método **crear**.

```
if self.Foto:

    pr.process(self.path)
    x,y,f,g,h=pr.data()

    if imagen:
        img.append(pr.imgT)

    data_fruta_x.append(x)
    data_fruta_y.append(y)
    rel_nu.append(h)
    rel_per.append(f)
    rel_ret.append(g)

for i in contenido:
    pr.process(self.path+rf'\{name}\{i}')
    x,y,f,g,h=pr.data()

    if imagen:
        img.append(pr.imgT)

    data_fruta_x.append(x)
    data_fruta_y.append(y)
    rel_nu.append(h)
    rel_per.append(f)
    rel_ret.append(g)

if imagen:
    img_dict.update({name:img})

data_fruta_x_dict.update({name:data_fruta_x})
data_fruta_y_dict.update({name:data_fruta_y})
```

Figura 33: Detalles de la clase **Cache**. Método **crear**.

```
    if imagen:
        self.tabla.update({'imagen':img_dict})

    self.data.update({'data_x':data_fruta_x_dict})
    self.data.update({'data_y':data_fruta_y_dict})
    self.data['nu02']=rel_nu
    self.data['ret']=rel_ret
    self.data['per']=rel_per

    if self.archivo:
        tf = open(self.path+"/cache.json", "w")
        json.dump(self.data,tf)
        tf.close()

def leer_imagenes(self):

    """Leamos la imagenes de la base de datos"""

    for i in self.frutas:
        self.crear(i,imagen= True)
```

Figura 34: Detalles de la clase **Cache**. Método **leer_imagenes**.

- **processimage**: Esta clase fue creada para la aplicación de filtros, procesamiento y vectorización de imagenes.

```
class processimage():

    def __init__(self):···

    def background(self,thr):···

    def filter(self):···

    def process(self,name,foto=True):···

    def data(self):···
```

Figura 35: Métodos de la clase **processimage**.

```
class processimage():

    def __init__(self):
        pass

    def background(self,thr):

        """Borramos en el fondo de la foto"""

        hist1 = cv.calcHist([self.src_copy2], [2], None, [256], [0, 256])
        brillo=np.argmax(hist1)

        if brillo>100:
            self.imgResult[np.all(self.imgResult>=thr, axis=2)] = 0
            self.imgT.append(self.imgResult.copy())
        else:
            self.imgResult[np.all(self.imgResult<=thr, axis=2)] = 0
            self.imgT.append(self.imgResult.copy())

    def filter(self):

        """Aplicamos un filtro laplaciono"""

        kernel = np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]], dtype=np.float32)
        imgLaplacian = cv.filter2D(self.src, cv.CV_32F, kernel)
        sharp = np.float32(self.src)
        self.imgResult = sharp - imgLaplacian
        #Convertimos devuelta a escala de grises
        self.imgResult = np.clip(self.imgResult, 0, 255)
        self.imgResult = self.imgResult.astype('uint8')
        self.imgT.append(self.imgResult.copy())
        imgLaplacian = np.clip(imgLaplacian, 0, 255)
        imgLaplacian = np.uint8(imgLaplacian)
```

Figura 36: Detalles de la clase **processimage**. Métodos **background** y **filter**.

```python
def process(self,name,foto=True):

    """ Realiza el procesamiento de las imagenes y su vecotrización """

    self.imgT=[]
    if foto:
        self.src = cv.imread(name)
        self.src_copy = cv.imread(name)
        self.src_copy2 = cv.imread(name)
    else:
        self.src=name
        self.src_copy = name
        self.src_copy2 =name

    self.imgT.append(self.src.copy())
    src_copy2=cv.cvtColor(self.src_copy2,cv.COLOR_BGR2HSV)
    hist = cv.calcHist([src_copy2], [1], None, [256], [0, 256])
    saturacion=np.argmax(hist)
    hist = cv.calcHist([src_copy2], [2], None, [256], [0, 256])
    brillo=np.argmax(hist)
    thr=130
    if saturacion<=5 and brillo<=230:

        thr=120

    if saturacion<=5 and brillo>=230:

        thr=140

    if 5>saturacion and saturacion<=15:

        thr=80
```

Figura 37: Detalles de la clase **processimage**. Método **process**.

```python
self.filter()
self.background(thr)


#Convertimos a escala de grises
bw = cv.cvtColor(self.imgResult, cv.COLOR_BGR2GRAY)
self.imgT.append(bw.copy())
#Aplicamos el threshold
_, bw = cv.threshold(bw, 0, 255, cv.THRESH_BINARY + cv.THRESH_OTSU)
self.imgT.append(bw.copy())
#Aplicando el distance Transform, podemos obtener los picos de la imagen
# por lo que a partir de ahí podemos realizar una segmentación de la imagen basada en marcadores utilizando el watershed algorithm.
dist = cv.distanceTransform(bw, cv.DIST_L2, 3)
# Normalizamos la imagen en el rango = {0.0, 1.0}
# asi podemos visualizarla y aplicarle el threshold.
cv.normalize(dist, dist, 0, 1.0, cv.NORM_MINMAX)
self.imgT.append(dist.copy())
_, dist = cv.threshold(dist, 0.4, 1.0, cv.THRESH_BINARY)
# Dilatamos un poco la imagen transformada
kernel1 = np.ones((3,3), dtype=np.uint8)
dist = cv.dilate(dist, kernel1)
dist_8u = dist.astype('uint8')
# Buscamos los marcadores
contours,_ = cv.findContours(dist_8u, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_SIMPLE)
# Creamos una imgane marcada para el watershed algoritmo
markers = np.zeros(dist.shape, dtype=np.int32)
# Rellenamos los marcadores
```

Figura 38: Detalles de la clase **processimage**. Método **process**.

```
for i in range(len(contours)):
    cv.drawContours(markers, contours, i, (i+1), -1)
# Dibujamos el fondo de los marcadores
cv.circle(markers, (5,5), 3, (255,255,255), -1)
#Aplicamos el watersheld algorithm
self.imgResult[bw==0] = 0
cv.watershed(self.imgResult, markers)
self.mark = markers.astype('uint8')
self.mark = cv.bitwise_not(self.mark)
kernel = np.ones((5,5), dtype=np.uint8)
#Filtramos el ruido.
self.mark=cv.morphologyEx(self.mark,cv.MORPH_CLOSE,kernel,iterations = 3)
self.mark=cv.morphologyEx(self.mark,cv.MORPH_OPEN,kernel,iterations = 3)
self.imgT.append(self.mark.copy())
contour1,_ = cv.findContours(self.mark,1,2)
self.cnt=0

if len(contour1)>0:
    a=0
    for i in range(len(contour1)):
        if len(contour1[i])>=len(contour1[a]):
            a=i
    self.cnt=contour1[a]
    self.cnt= cv.convexHull(self.cnt)
    cv.drawContours(self.src_copy,contour1, -1, (0,255,0), 5)
    self.imgT.append(self.src_copy.copy())
    self.x,self.y,self.w,self.h = cv.boundingRect(self.cnt)
```

Figura 39: Detalles de la clase **processimage**. Método **process**.

```
def data(self):

    s=cv.bitwise_and(self.src_copy,self.src_copy,mask = self.mark)
    s=cv.cvtColor(s,cv.COLOR_BGR2HSV)
    hist = cv.calcHist([s], [0], None, [256], [1, 256])
    color=np.argmax(hist)
    if color > 50:
        color=1
    area = cv.contourArea(self.cnt)
    perimetro = cv.arcLength(self.cnt,True)
    (x,y),radius = cv.minEnclosingCircle(self.cnt)
    rel_cir=math.pi*(radius**2)/area
    rel_per=math.pi*2*radius/perimetro
    rect = cv.minAreaRect(self.cnt)
    rel_ret=rect[1][0]*rect[1][1]/area
    M = cv.moments(self.cnt)
    return float(color), float(rel_cir),float(rel_per),float(rel_ret),float(M['nu20'])
```

Figura 40: Detalles de la clase **processimage**. Método **data**.

- **anneling**: Esta clase realiza el temple simulado.

```
class anneling():

    def __init__(self,_T):···

    def next_state(self):···

    def energy(self,state):···

    def probability(self,delta_energy,Temp):···

    def search(self,cluster_in,x_T,y_T,kn):···
```

Figura 41: Métodos de la clase **anneling**.

```python
class anneling():

    def __init__(self,_T):

        self.T=_T
        self.t=len(_T)
        self.it=0

    def next_state(self):

        """ Genera un estado vecino"""

        aux=self.actual_state.copy()
        index=random.sample(range(self.d2),2)
        aux[index[0]],aux[index[1]] = self.actual_state[index[1]],self.actual_state[index[0]]

        return aux

    def energy(self,state):

        """Devuelve la energia del conjunto de ordenes o el camino"""

        dT=kmeans(self.x_T,self.y_T,state,kn=self.kn,anelling=True)
        energy=dT.sum()

        return energy

    def probability(self,delta_energy,Temp):

        """ Devuelve una probabilidad en función de la temperatura"""

        return math.exp(delta_energy/Temp)
```

Figura 42: Detalles de la clase **anneling**. Métodos **next_state**, **energy** y **probability**

```python
def search(self,cluster_in,x_T,y_T,kn):

    it=0
    self.actual_state=cluster_in
    self.x_T=x_T
    self.y_T=y_T
    self.kn=kn
    self.list_energy=[]
    self.next_stateaux=[]
    self.d2=len(cluster_in)
    self.E1=self.energy(self.actual_state.copy())

    for i in range(self.t):

        it+=1

        if abs(self.T[i])==0 or it==self.it :

            return self.actual_state,self.list_energy


        self.next_stateaux=self.next_state()
        E2=self.energy(self.next_stateaux.copy())
        deltaenergy=self.E1-E2
```

Figura 43: Detalles de la clase **anneling**. Método **search**

```
if deltaenergy>0:

    it=0
    self.actual_state=self.next_stateaux
    self.E1=E2

else:

    prob=self.probability(deltaenergy,self.T[i])
    choise=random.choices([0,1],[1-prob,prob])

    if (choise[0]==1):

        self.actual_state=self.next_stateaux
        self.E1=E2

self.list_energy.append(self.E1)
```

Figura 44: Detalles de la clase **anneling**.

- **Funciones**

  - **kmeans**: Implementa el algoritmo **k-means**. Recibe un **cluster** inicial aleatorio y los datos de la base de datos **x_T** y **y_T**. El algoritmo devuelve la clasificación hecha y las coordenadas de los centroides, en caso de que este se implemente dentro del Temple simulado se devuelde un parámetro que es una medida de la energía.

```
def kmeans(x_T,y_T,cluster,kn=4,anelling=False):

    #cluster: es la clasificación inicial de los datos de manera aleatoria.
    #Definimos inicialmente los centroides en cero
    x_i=0
    y_i=0

    dT=np.zeros(kn)
    it=True
    ii=0
    while(it):

        x_cluster=np.zeros(kn)
        y_cluster=np.zeros(kn)

        #Calculamos la distancia a cada centroide, y clasificamos la información en función de la menor distancia
        for i in range(len(x_T)):

            if ii>0:

                d=np.power((x_T[i]-x_i),2)+np.power((y_T[i]-y_i),2)
                a=d.argmin()
                cluster.append(a)
                dT[a]+=d[a]

            x_cluster[cluster[i]]+=x_T[i]
            y_cluster[cluster[i]]+=y_T[i]

        #Calculamos los centroides.
        for i in range(kn):

            x_cluster[i]/=(cluster.count(i)+1e-5)
            y_cluster[i]/=(cluster.count(i)+1e-5)
            dT[i]/=(cluster.count(i)+1e-5)
```

Figura 45: Detalles de la función **kmeans**.

```
#Verificamos si el algoritmo converge
if np.all(abs(x_cluster-x_i)<0.001) and np.all(abs(y_cluster-y_i)<0.001):


    if anelling:
        return dT
    else:
        return cluster,x_cluster,y_cluster

else:

    x_i=x_cluster
    y_i=y_cluster
    ii+=1
    cluster=[]
```

Figura 46: Detalles de la función **kmeans**.

- **kmeans_class**: Clasifica un dato de entrada en funcion de los centroides calculados con el algoritmo k-means.

```
def kmeans_class(data,x_cluster,y_cluster,cluster,cluster_T,x_Tmax,y_Tmax):

    """Clasificamos un dato de entrada en funcion de los centroides calculados con el algoritmo k-means"""

    frutas=['limon','tomate','naranja','banana']

    #Normalizamos los datos
    data[0]=data[0]/x_Tmax
    data[1]=data[1]/y_Tmax
    data=np.array(data)

    #Calculamos la distancia del dato con respecto a los centroides y nos quedamos con el menor.
    d=np.power((data[0]-x_cluster),2)+np.power((data[1]-y_cluster),2)
    a=d.argmin()

    dict1={'0':0,'1':0,'2':0,'3':0}
    #Calculamos la presición del algoritmo.
    for i in range(len(cluster)):

        if a==cluster[i]:

            dict1[f'{cluster_T[i]}']+=1
    T2=np.array(list(dict1.values()))
    T=np.sum(T2)

    a=T2.argmax()
    aux=dict1[f'{a}']
    s=frutas[a]
    b= f'{round(aux/T*100)}%'
    return s,b
```

Figura 47: Detalles de la función **kmeans_class**.

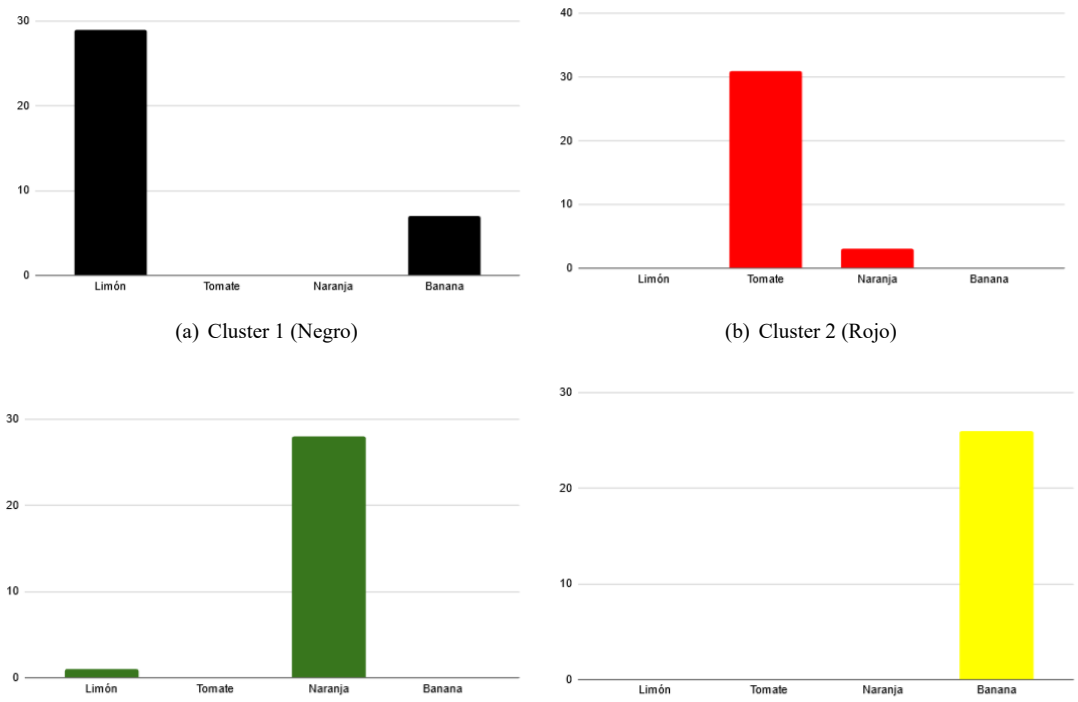
- **knn**: Implementa el algoritmo **knn**. Recibe el número de vecinos **k**(es importante que este número siempre sea impar para que no nos de un resultado imparcial), la **data** a clasificar, un **cluster** inicial aleatorio y los datos de la base de datos **x_T** y **y_T**. Devuelve la clasificación realizada.

```python
def knn(k,data,x_T,y_T,cluster_T,x_Tmax,y_Tmax):

    #Aplicamos el algoritmo K nearest neighbours.
    frutas=['limon','tomate','naranja','banana']

    #Normalizamos los datos
    data[0]=data[0]/x_Tmax
    data[1]=data[1]/y_Tmax
    data=np.array(data)

    #Calculamos la distancia de nuestro dato de entrada con todos los vecinos
    d=np.power((data[0]-x_T),2)+np.power((data[1]-y_T),2)
    aux=list(d.copy())

    #Ordenamos de menor a mayor
    d.sort()


    dict1={'0':0,'1':0,'2':0,'3':0}
    #Seleccionamos los k vecinos mas cercanos
    for i in range(k):
        a=aux.index(d[i])
        dict1[f'{cluster_T[a]}']+=1

    T2=np.array(list(dict1.values()))
    T=np.sum(T2)
    #Clasificamos en función de los vecinos

    a=T2.argmax()
    aux=dict1[f'{a}']
    s=frutas[a]
    b= f'{round(aux/T*100)}%'
    return s,b
```

Figura 48: Detalles de la función **knn**.

6. Ejemplos de aplicación

A continuación se mostrará un ejemplo para cada caso. Pero en primer lugar se muestra en la figura 49 como estan conformados los cluster mostrados en la figura 11 realizados por el algoritmo **K-means**.



(a) Cluster 1 (Negro)

(b) Cluster 2 (Rojo)

(c) Cluster 3 (Verde)

(d) Cluster 4 (Amarillo)

Figura 49: Conformación de los clusters.

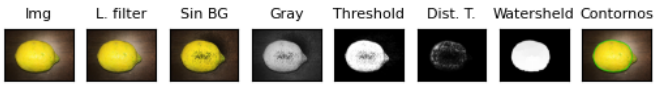
Cluster 1		Cluster 2		Cluster 3		Cluster 4	
Limón	29	Limón	0	Limón	1	Limón	0
Tomate	0	Tomate	31	Tomate	0	Tomate	0
Naranja	0	Naranja	3	Naranja	28	Naranja	0
Banana	7	Banana	0	Banana	0	Banana	26

Figura 50: Detalle de conformación de los clusters.

6.1. Limón

Se nuestra el comportamiento del algoritmo para la imagen mostrada en el figura 51 (a), como vemos en la fig 51(b) la obtención del contorno es correcta y ambos algoritmo clasifican la fruta correctamente dando como resultado: **K-means=81 %** de certeza y **Knn=100 %** de certeza.

(a) Limón



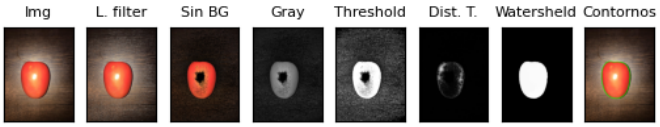
(b) Procesamiento limón

Figura 51: Ejemplo limón.

6.2. Tomate

Para probar el algoritmo para esta fruta se dio la imagen mostrada en la figura 52 (a), como vemos el algoritmo puede detectar en donde esta la fruta correctamente (figura 52 (b)) y clasificarla correctamente dando: **K-means=91 %** de certeza y **Knn=100 %** de certeza.

(a) Tomate.

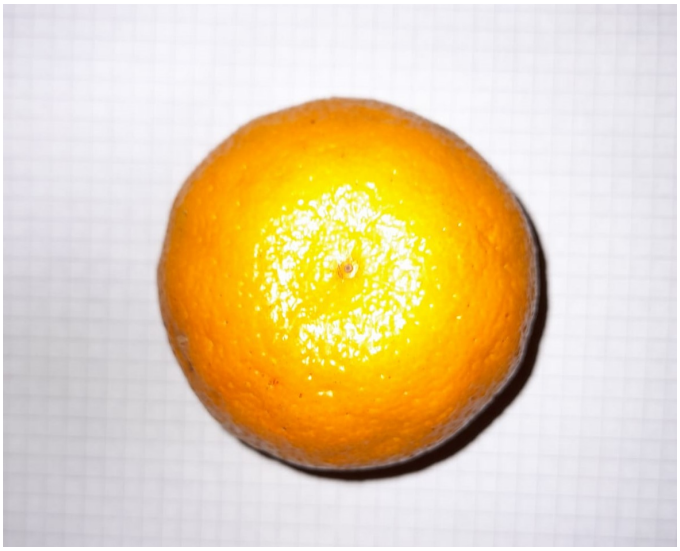


(b) Procesamiento tomate.

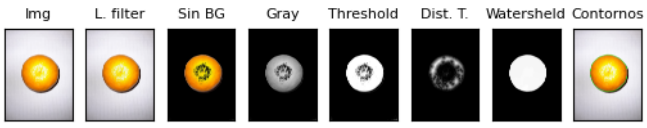
Figura 52: Ejemplo tomate.

6.3. Naranja

Se le dio al algoritmo la imagen mostrado en la figura 53(a) y se pudo clasificar correctamente dando como resultado: **K-means: 97 %** y **knn: 100 %**.



(a) Naranja



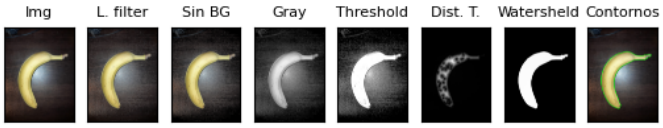
(b) Procesamiento naranja

Figura 53: Ejemplo naranja.

6.4. Banana

Por último, se realizó el mismo procedimiento que en los casos anteriores con la banana de la figura 54, dando como resultado: **K-means: 81 % limón** y **Knn: 100 % banana**. En nuestra base de datos tenemos imagenes muy similares a la de este ejemplo (figura 55, bananas muy curvas por lo que tienen mayor redondez) , la desventaja de estas es que se encuentran cerca de los puntos que representan a los limones como se ve en la figura 12 (Puntos amarillos encerrados por el circulo negro), por ende el algoritmo **K-means** la clasifica como un limón, ya que al cluster que entra se encuentra conformado en su mayoría por limones. Por otro lado, el algoritmo **Knn** la clasifica correctamente como una banana, ya que los k vecinos más cercanos son bananas, esto muestra la potencialidad de este algoritmo para clasificar datos dentro de grupos de datos donde no se muestra una separación marcada. Se buscó con este ejemplo mostrar la ventaja del algoritmo **Knn** con respecto al algoritmo **K-means**.

(a) Banana



(b) Procesamiento banana

Figura 54: Ejemplo banana

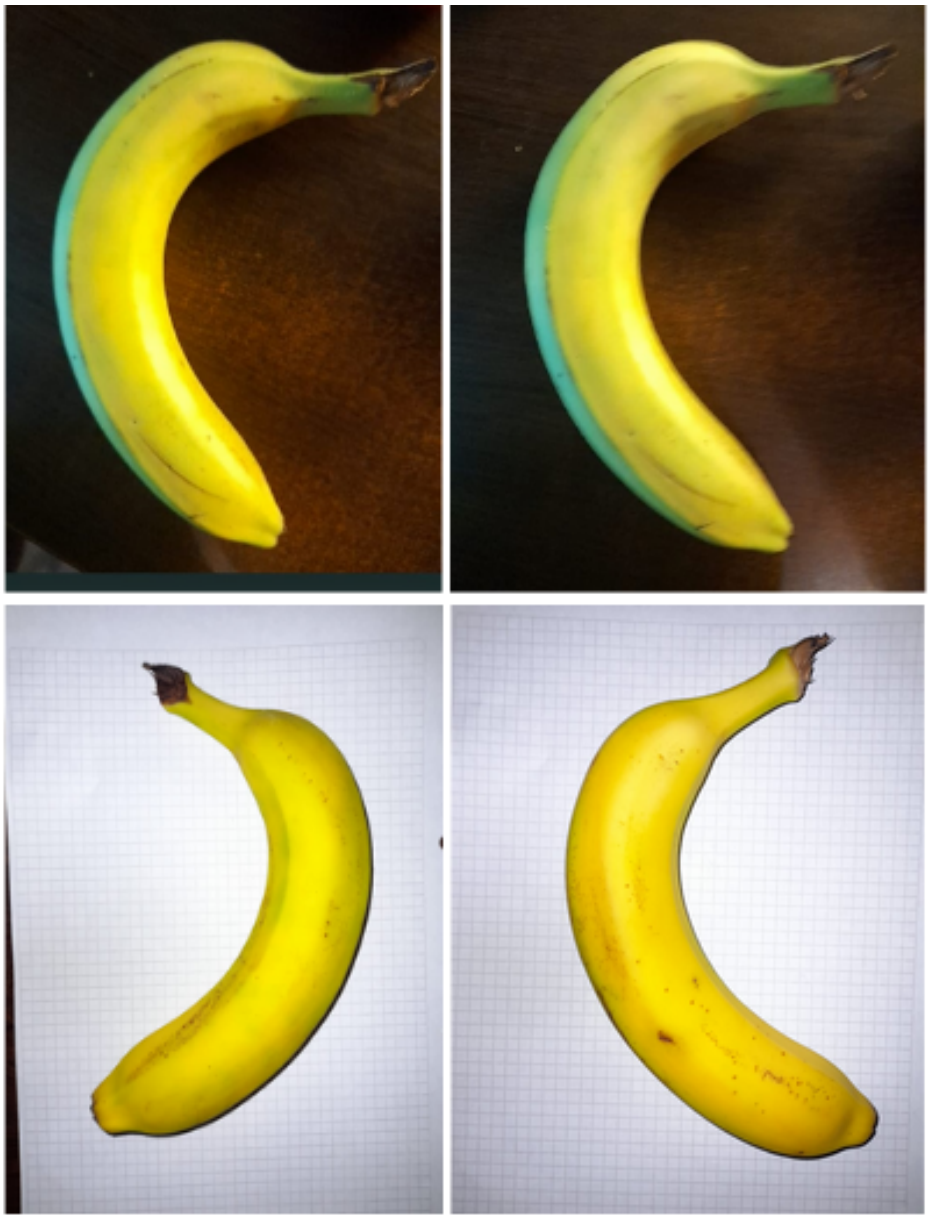


Figura 55: Bananas que caen dentro del grupo de limones.

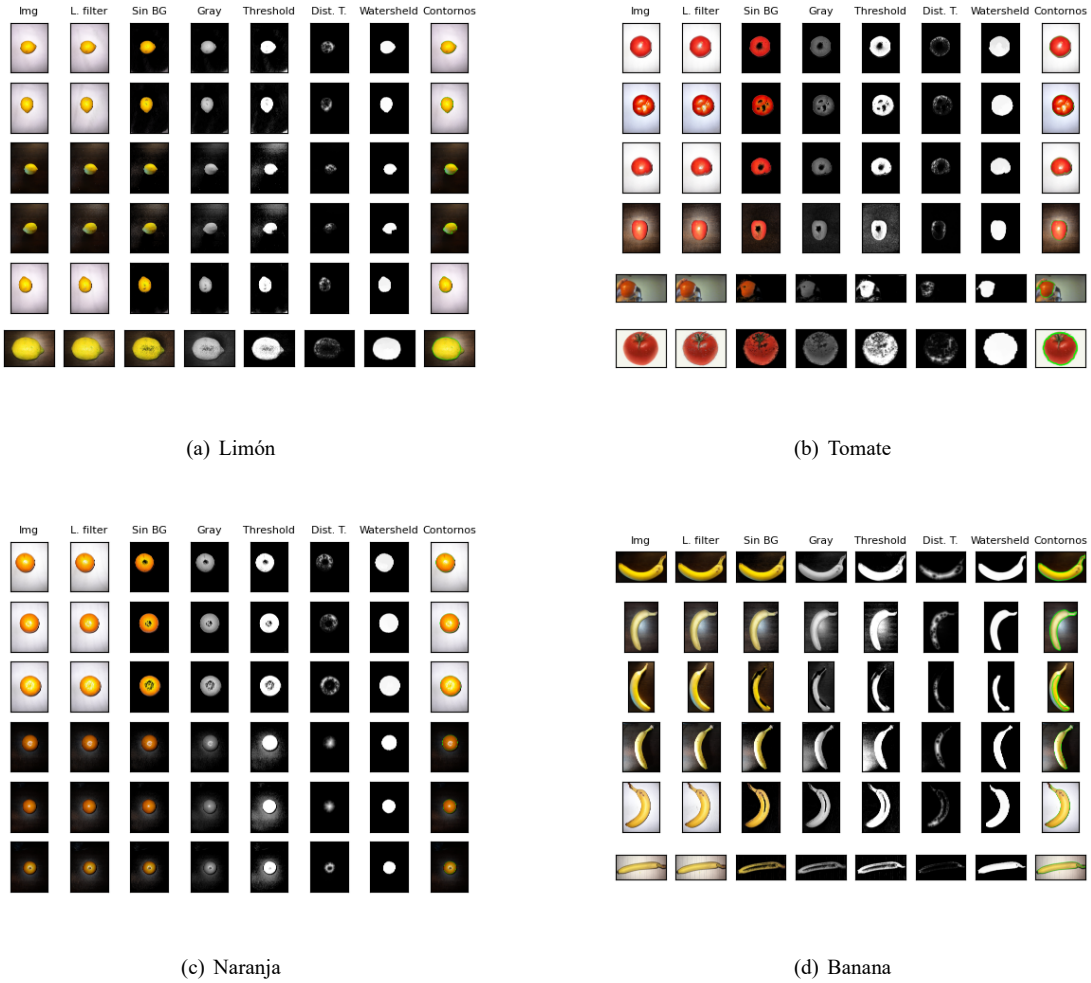
7. Resultados

Los datos utilizados en el conjunto de entrenamiento fueron:

- Imágenes en cualquir formato.
- La fruta debe aparecer sola y con buena iluminación.
- El fondo debe ser lo más neutro posible, preferentemente blanco o negro.
- Las dimensiones no afectan porque todas las imágenes son redimensionadas al mismo tamaño.
- Para las bananas se utilizaron 33 imágenes de entrenamiento. Todas con bananas amarillas o maduras, pero ninguna verde (plátanos).
- Para los tomates se utilizaron 31 imágenes de entrenamiento. Distintos tipos, tomates redondos y tomates perita.

- Para los limones se utilizaron 30 imágenes de entrenamiento. Limones amarillos ninguno verde (limas), en lo posible que se vieran de costado, algunos con hojas.
- Para las naranjas se utilizaron 31 imágenes de entrenamiento. Naranjas bien maduras (anaranjadas) y bien redondas. Algunas con hojas.

Se porcedió a testear los algoritmos con 6 imagenes de prueba de cada fruta (figura 56), tratando de que sean imágenes lo más variadas posibles y siguiendo los mismos requisitos anteriores, ninguna imagen de prueba estaba incluida en el conjunto de entrenamiento.



(a) Limón

(b) Tomate

(c) Naranja

(d) Banana

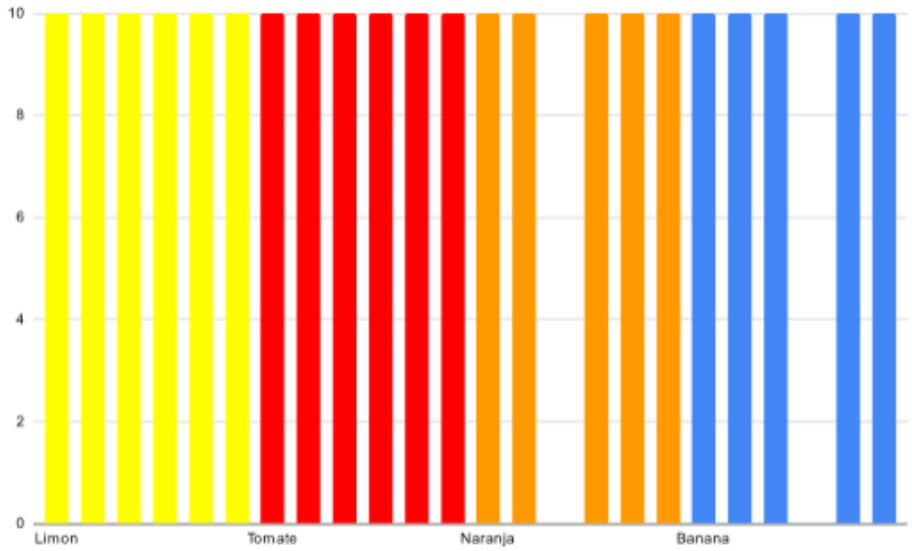
Figura 56: Procesamiento del conjunto de prueba

A continuación se presentan las estadísticas de cada algoritmo:

- **K-means**: Como vemos en la figura 57 el algoritmo presenta resultado satisfactorios, con una **exactitud** de 91 %, es decir, que la gran mayoría de las veces el algoritmo acierta. Para probar el algoritmo se corrió este diez veces para el mismo conjunto de prueba y para cada acierto se sumó 1, asi para las fotos que se acertó siempre se tiene 10 y para las que no se acertó nunca 0. Como vemos nuestro algoritmo tuvo problemas para reconocer la tercera foto del conjunto de pruebra de naranjas, dado que debido a la mala iluminación de esta foto el algoritmo la clasificó como un limón por su color amarillento y en la cuarta foto del conjunto de las bananas, en donde el algoritmo clasificó a esta como un limón, debido a esta banana es más curva y aumentó su redondez.

	Limon	Tomate	Naranaja	Banana	
Foto 1	10	10	10	10	
Foto 2	10	10	10	10	
Foto 3	10	10	0	10	
Foto 4	10	10	10	0	
Foto 5	10	10	10	10	
Foto 6	10	10	10	10	
Promedio fruta	100	100	83,33333333	83,33333333	Exatitud
Promedio Total	91,66666667				

(a) Tabla



(b) Gráfica

Figura 57: Estadísticas **K-means**, 4 clusters.

- **Knn**: Se realizó el mismo procemiento que con **K-means** obteniendo como resultado lo mostrado en la figura 58 y 59. Como vemos la **exactitud** de este algoritmo es del 95 %.

	Limon	Tomate	Naranaja	Banana	
Foto 1	10	10	10	10	
Foto 2	10	10	10	10	
Foto 3	10	10	0	10	
Foto 4	10	10	10	10	
Foto 5	10	10	10	10	
Foto 6	10	10	10	10	
Promedio fruta	100	100	83,33333333	100	Exatitud
Promedio Total	95,83333333				

Figura 58: Estadísticas **Knn**. Tabla

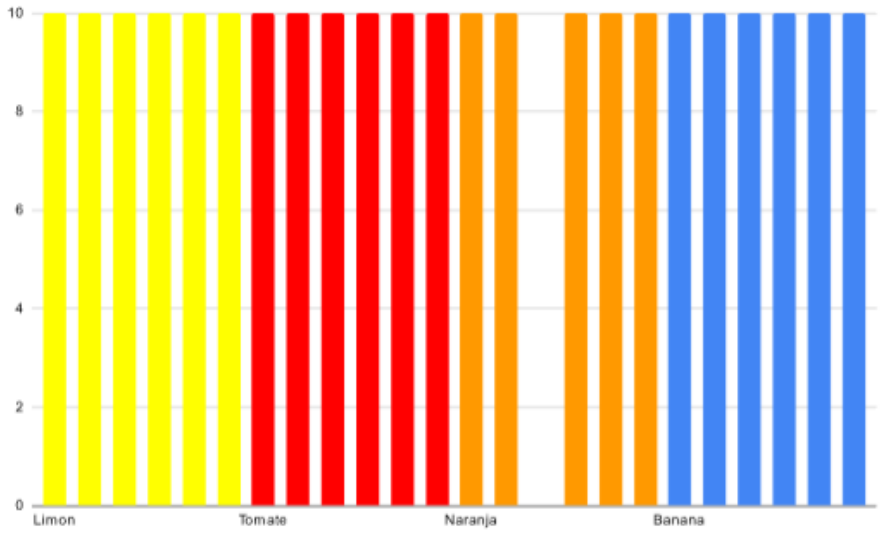
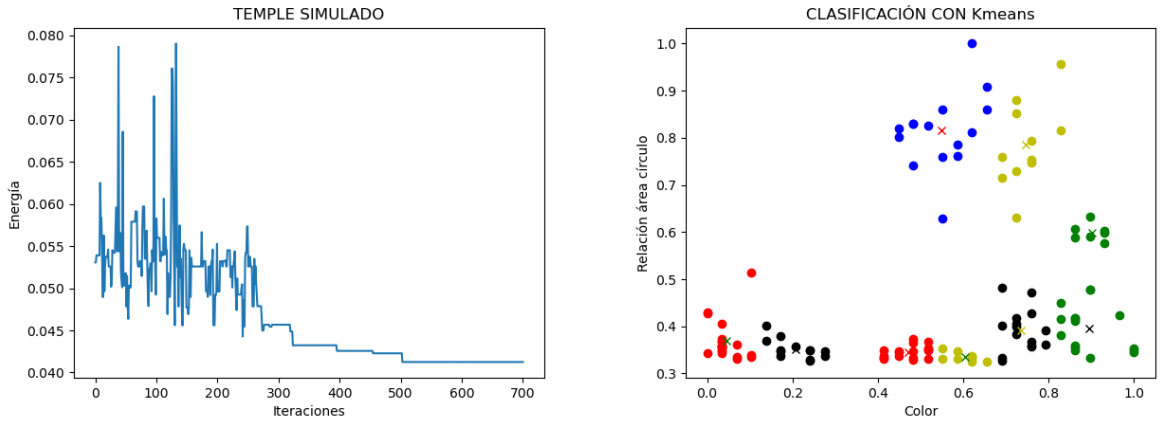


Figura 59: Estadísticas **Knn**. Gráfica

Como vemos ambos algoritmos tienen una **precisión** del 100 %, es decir, siempre nos dan el mismo resultado. Esto se ve notoriamente ya que a los algoritmos se los corrió 10 veces y todos los intentos se obtuvo los mismos resultados.

Por último, no conformes con la exactitud de nuestro algoritmo **K-means** se procedió a aumentar el número de cluster a 9, en la figura 60 se ve la clasificación, intentando mejorar este parámetro, dando como resultado lo plasmado en la figura 61 y 62. Como vemos el algoritmo mejora su exactitud a 94 % pero por otro lado disminuye su precisión.



(a) Temple

(b) Clasificación

Figura 60: **K-means** con 9 clusters.

	Limon	Tomate	Naranaja	Banana	
Foto 1	10	10	10	10	
Foto 2	10	10	10	10	
Foto 3	10	10	0	10	
Foto 4	10	10	10	6	
Foto 5	10	10	10	10	
Foto 6	10	10	10	10	
Promedio fruta	100	100	83,33333333	93,33333333	Exatitud
Promedio Total	94,16666667				

Figura 61: Estadísticas **K-means**, 9 clusters. Tabla.

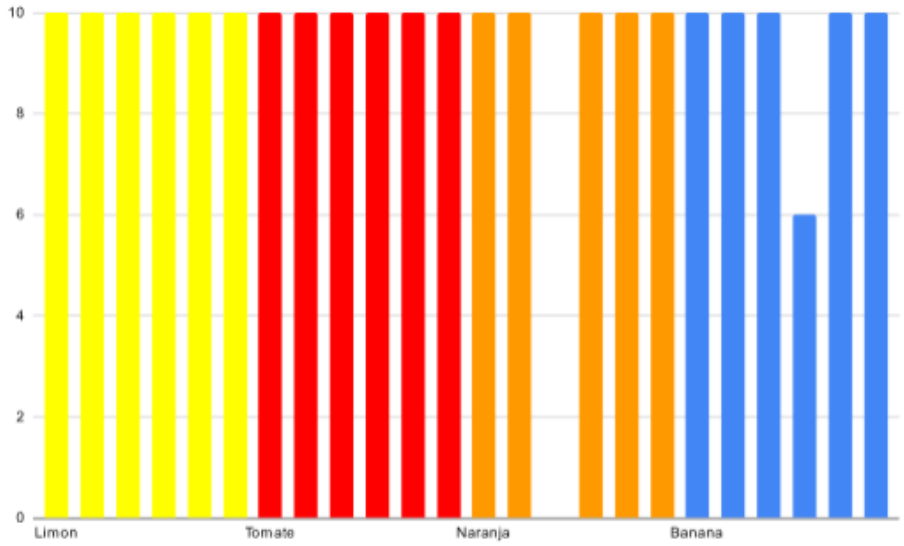


Figura 62: Estadísticas **K-means**, 9 clusters. Gráfica.

8. Conclusiones

Como conclusiones podemos sacar que el algoritmo k-nn funcionó mejor y prácticamente no tuvo desaciertos. En cambio, el k-means en algunos casos puso puntos dentro de categorías que por características eran similares, pero que no eran de las mismas frutas. La eficiencia para k-nn fue de 95 % y del 91 % para K-means.

Las ventajas de este tipo de sistemas de visión artificial es que en general mejora a medida que el conjunto de datos aumenta y además es bastante sencillo de programar e implementar y es muy económico ya que requiere solamente de una cámara y no mucho costo computacional. El costo computacional no es grande porque una vez analizadas las imágenes todo se reduce a operaciones con vectores.

Las desventajas son que no son tan precisos ni confiables como otros tipos de algoritmos de inteligencia artificial, hay algoritmos que resuelven mejor las tareas. Además, que si en nuestros algoritmos ahora ponemos imágenes de varias frutas juntas no podrá reconocer ninguna, si tiene fallas en sus resultados tampoco nos vamos a enterar.

Algunas posibles mejoras a implementar son mejorar la detección de color, con diferentes cálculos de las capas RGB (no solo usar HSV). También ver si es posible aplicar algún filtro que nos de un parámetro referido a la textura de la imagen ya que las naranjas y limones, por ejemplo, son mas porosos que los tomates que son más lisos, entonces si por algún método de iluminación se pudiera detectar esta propiedad se podría mejorar bastante el algoritmo. Por el momento solo se trabaja con parámetros de color y parámetros de forma.

9. Referencias

1. Apuntes de cátedra

2. https://opencv.org/
3. https://opencv.org/opencv-free-course/
4. https://www.analyticsvidhya.com/blog/2019/03/opencv-functions-computer-vision-python/
5. https://stackoverflow.com/
6. https://homepages.inf.ed.ac.uk/rbf/CVDICT/cvdict.htm
7. https://homepages.inf.ed.ac.uk/rbf/HIPR2/hipr_top.htm