UNIVERSIDAD DE PIURA

FACULTAD DE INGENIERÍA

Detección de macronutrientes y enfermedades en campos de cultivo de banano orgánico con Machine Learning

Trabajo de Investigación para optar el Grado de
Bachiller en Ingeniería Mecánico - Eléctrica

Evelia Román Alvarado
María de Fátima Ruiz García

Asesor:
Dr. Ing. William Ipanaqué Alama

Piura, febrero de 2021

UNIVERSITAS STUDIORUM PIURENSIS

Resumen

La detección de deficiencia de macronutrientes y enfermedades en el banano orgánico se ha convertido en un factor determinante para ser de este un excelente producto de exportación. En el presente trabajo se realiza una comparación entre tres algoritmos de clasificación híbridos y redes neuronales convolucionales, es decir, la combinación de estos algoritmos de clasificación (*Support Vector Classifier, K-Nearest Neighbors y Random Forest*) y filtros convolucionales de una arquitectura de 11 capas y *VGG16*, con el objetivo de que a partir de los modelos propuestos se logre reducir las pérdidas producto de la deficiencia de macronutrientes y aparición de enfermedades en plantaciones de banano orgánico, además de disminuir el tiempo del proceso actual de detección de estos.

Se han utilizado dos tipos de *data set* de imágenes de hojas de banano orgánico, una para detección de deficiencia de macronutrientes otra para enfermedades. El primer *data set* de deficiencia de macronutrientes consta de imágenes de 3 macronutrientes: nitrógeno, fósforo, potasio e imágenes de hojas sanas. Este data set fue obtenido por un grupo de ingenieros de la Universidad Inmaculada Concepción en la región de Davao del norte, en Filipinas. En el caso del *data set* de enfermedades consta de dos tipos de enfermedades: sigatoka negra, marchitez bacteriana e hojas sanas. Fue obtenida en un repositorio de *GitHub*. Para los modelos de redes neuronales convoluciones se decidió trabajar con dos tipos de arquitecturas: una de 11 capas de filtros convoluciones y la *VGG16.*

Estos algoritmos y redes neuronales convolucionales fueron evaluados a partir de su *accuracy, f1 score y confusion matrix.* A partir de estos indicadores se llegó a la conclusión que los algoritmos de clasificación híbridos son superiores en *accuracy* que el de las redes neuronales convoluciones. Al comparar solo los algoritmos de clasificación híbridos se descubrió que *Random Forest* con filtros convoluciones de *VGG16* obtuvo los mejores resultados en cuanto a *accuracy* y *f1 score* para ambos data set.

UNIVERSITAS STUDIORUM PIURENSIS

Tabla de contenido

Lista de figuras

UNIVERSITAS STUDIORUM PIURENSIS

Introducción

La región de Piura se caracteriza por contar con las características de suelo y clima idóneas para la siembra de distintos cultivos de alta demanda comercial en diversos mercados como Asia y Europa. Uno de los cultivos que han tomado más importancia en la última década es el banano orgánico, dada la constante mejora del proceso productivo se ha visto necesario la implementación de nuevas técnicas de vanguardia que permita al cultivo llegar a cumplir con los estándares internacionales de calidad.

Siendo así, una de las problemáticas a abordar por el sector agrícola el manejo eficiente, control efectivo de los nutrientes y enfermedades para que a partir de ellos se puedan reducir los costos referidos al uso de fertilizantes y al tratamiento de estas respectivamente ya que muchas veces la mala aplicación y selección de estos puede causar grandes daños al cultivo y dañar la cadena de producción del banano orgánico teniendo como consecuencia final un impacto negativo en el sector económico-agrícola.

El presente trabajo buscará el diseño e implementación de un modelo de diagnóstico de nutrientes y enfermedades en suelos de cultivo de banano orgánico empleando herramientas de aprendizaje automático como redes neuronales convoluciones y algoritmos de clasificación híbridos evaluando ventajas y desventajas de cada uno.

Las redes neuronales convolucionales (CNN) (Q. Zhang 2018) se caracterizan por su alta capacidad de clasificar imágenes como estudios anteriores lo comprueban. Por otra parte, los algoritmos de clasificación como: *Support Vector Classifier, K-Nearest Neighbors y Random Forest* han alcanzado gran popularidad para clasificar imágenes y toma de decisiones mucho más rápidas que las CNN, resaltando entre las anteriores *Random Forest* cuyo desempeño en tareas similares a las propuestas en este trabajo han tenido muy buenos resultados.

Por último, agradecemos a nuestros padres que han sido y serán un pilar fundamental en nuestra formación académica y a nuestros asesores que sin su guía la elaboración de este trabajo de investigación no hubiese sido posible.

UNIVERSITAS STUDIORUM PIURENSIS

Capítulo 1
Marco teórico y estado del arte

1.1 Caracterización del suelo del banano orgánico

El suelo se puede definir como una sustancia heterogénea, se puede también resaltar que la conformación y composición de esta puede variar según del lugar donde se analice. Hay otros factores de los cuales el suelo depende, algunos de estos pueden ser: clima, vegetación, etc. Cabe resaltar que las características del suelo van a resultar muy importantes al momento de decidir que el cultivo a sembrar, siendo en este caso, el banano orgánico. (Rumiche 2015)

Para el cultivo del banano orgánico, se necesitan ciertos requerimientos del suelo, dentro de ellos se encuentra el pH cuyo valor óptimo está entre 6.5-7, esto quiere decir que el suelo se encuentra en un medio ni tan ácido ni tan básico. Además, es muy importante que los suelos sean abundantes de materia orgánica para poder asegurar el buen crecimiento del cultivo (Vegas 2013) Uno de los métodos más utilizados para mantener la riqueza del suelo es mediante el abonado por medio de las mismas hojas o tallos de la planta. Sin embargo, a pesar de ser un método muy común puede traer como consecuencia la infestación del suelo de plagas que hayan podido quedar en las hojas del banano y así reducir las probabilidades del buen desarrollo de la planta.

1.1.1 Requerimientos del del suelo de banano orgánico

Como se ha mencionado antes hay ciertos factores que son determinantes para el buen cultivo del banano orgánico. Lo recomendable es que se busque un suelo plano ya que es donde mejor se da el crecimiento de esta planta, pero se puede considerar cierto porcentaje de inclinación en el terreno sin que el cultivo se vea afectado. Además, para toda buena siembra es también importante que el suelo de cultivo se encuentre localizado en una zona con un buen nivel freático, lo recomendable es que sólo sea de 1.20 m. (Vegas 2013)

Es importante resaltar que cuando el suelo ya ha sido utilizado para la siembra de otros cultivos, solo necesitaría retirar la maleza y hojas secas del anterior cultivo para empezar a cultivar la nueva planta. También se debe considerar un buen sistema de drenaje y de riego donde los surcos no excedan el 2% de pendiente permitida. (Vegas 2013)

1.1.2 Banano – Tipo de clima

- **Altitud.** Para el cultivo de banano, las zonas comprendidas entre 0 y 30 m.s.n.m (zonas costeras) son las adecuadas para el desarrollo del cultivo Son preferibles las llanuras húmedas próximas al mar.
- **Requerimientos de agua.** Dada su naturaleza herbácea y la gran superficie de la hoja, el requerimiento de agua del banano es alto. El 85-88% de la masa del fruto se constituye por agua. La pluviosidad necesaria varía de 120 a 150 mm de precipitaciones mensuales o 44 mm semanales. En caso de no ser provista por lluvia, esta tiene que ser provista por riegos de carácter regular y de manera constante.
  La carencia de agua en cualquier momento puede causar la reducción en el número y tamaño de los frutos y en el rendimiento final de la cosecha, además de desecación de hoja, marchitez de las vainas y la rotura del pseudotallo.
- **Temperatura.** Necesita una temperatura media entre 26 y 27°C para el crecimiento óptimo. La variación de la temperatura afecta el ciclo vegetativo del cultivo, pues se prolonga en cuanto menor sea la temperatura. Si la temperatura llega a estar por debajo de los 16°, la actividad vegetativa de la planta se reduce, paralizando la salida de hojas. Finalmente, bajo los 12°C, la fructificación se detiene.
- **Luminosidad.** El banano se puede cultivar en zonas con iluminación variada. Aunque una reducción de la iluminación no interrumpe el ciclo vegetativo, lo alarga considerablemente, aumentando el tiempo entre 8 a 14 meses. Por esa razón, se prefieren zonas de sol y despejadas. Estas condiciones se cumplen en la latitud 30 a 31º norte o sur y de los 1 a los 2 m de altitud.
- **Vientos.** Los efectos del viento van desde transpiración anormal debido a reapertura de estomas en la planta hasta la laceración de lámina foliar. Esto provoca pérdidas de rendimiento de hasta 20%. No se recomienda exponer el cultivo de banano a velocidades de viento mayores de 20 km/hora. (Infoagro Systems S.L s. f.; Torres 2012)

1.2 Variables de interés

La producción agrícola de alimentos actualmente está influenciada por distintos aspectos tales como, la administración del agua y el abastecimiento de los nutrientes necesarios con los cuales, se podrá asegurar una buena producción futura. Cada uno de estos nutrientes tendrán una función determinante en el desarrollo de las plantas y, por ende, se deberá guardar conocimiento de la relación existente en su exceso y/o deficiencia de estos en los cultivos. (Gutiérrez Castorena, Gutiérrez Castorena, y Ortiz Solorio 2015)

1.2.1 Macronutrientes:

- **Nitrógeno.** Es un elemento necesario para la síntesis de la clorofila y, por ende, es muy necesario para el proceso de fotosíntesis de la planta, ya que, sin este nutriente, la planta no podrá absorber la luz solar para el desarrollo del proceso

fotosintético, privándola así, de la capacidad para absorber demás nutrientes. Las plantas que presentan bajas concentraciones de Nitrógeno, aproximadamente 10 g kg-1 presentan una coloración verde claro, la misma que se logra percibir en las hojas viejas, mientras que las hojas nuevas permanecen con un color verde por un periodo de tiempo más prolongado. Por otro lado, valores excesivos de Nitrógeno o por encima de 20 a 50 g kg-1.

- **Fósforo.** Es un elemento que sirve para el desarrollo de las plantas ya que promoverá el desarrollo de las raíces, así como también, será un agente importante en la transferencia de características hereditarias.
  Una de las principales señales de carencia de fosforo en la planta es la apariencia de las hojas, las cuales aparecerán torcidas y con un aspecto de color purpura, mientras que para una deficiencia severa de fosforo, aparecerán áreas muertas en las hojas, en el tallo e incluso en el fruto. Por otro lado, una carencia de fosforo en el fruto, retardará el proceso de maduración de los cultivos.
- **Potasio.** Es uno de los elementos más importantes en la nutrición de la planta, así como también en el proceso de fotosíntesis de esta; por otro lado, la carencia de este nutriente implicara una reducción en el crecimiento y producción de la planta, debido a la reducción en la acumulación de carbohidratos propiciada por la falta de potasio.
  Una de las principales características que se generan por la deficiencia de potasio, es la necrosis de los márgenes en las hojas, así como también, se genera un proceso de crecimiento o desarrollo lento de la planta.

1.2.2 Micronutrientes:

- **Boro.** La carencia de Boro en plantas genera la presencia de paredes menos resistentes con respecto a plantas con buenos contenidos de boro. Las características generadas por la carencia de boro, será distinto de acuerdo con la especie vegetal en cuestión, así pues, para concentraciones, menores de 15 mg kg-1, se logra evidenciar ciertas características tales como, reducción del crecimiento y deformaciones, así como también, la presencia de hojas quebradizas y raíces cortas.
- **Cloro.** Es un elemento móvil dentro de la planta, es por ello la existencia de cierta dificultad para poder discriminar con facilidad la carencia de Cloro en la planta; sin embargo, existen ciertos aspectos tales como, la presencia de hojas y raíces de tamaño limitado que ayudaran a percibir la ausencia de dicho micronutriente. Por otro lado, cuando se habla de exceso de este elemento, las características saltan más a la luz y son más comunes y graves, esto último debido a la aparición de síntomas de toxicidad generados por el exceso de cloro en la planta, lo cual se puede percibir por una reducción en el ancho de las hojas, las mismas que tienden a rizarse.
- **Hierro.** El hierro es un agente importante en la producción de clorofila de la planta, por ende, su ausencia estará marcada por la pérdida de color verdoso en ella, con lo

cual, existe la posibilidad de tener una pigmentación totalmente amarilla en la planta, solo para casos de deficiencia fuerte de este micronutriente. Otra característica importante es la aparición de tallos y ramas finas y curvadas con ciertas limitaciones en su tamaño.

- **Zinc.** El zinc es un elemento que pertenece a la familia de micronutrientes más influyentes en el desarrollo de las plantas; su presencia está influenciada por pH bajos lo cual significará que será un suelo ácido. Entre las principales funciones del zinc con micronutriente esencial tenemos, la actuación como agente enzimático y esencial para la regulación de la estructura proteica de la planta.
  Como bien se dijo, la presencia de zinc estará delimitado por la presencia de pH en los suelos, por lo cual, se podrá afirmar que para suelos ácidos o con pH altos corresponderán valores altos de zinc. (DECHEN y NACHTIGALL 2007)

1.3 Enfermedades del banano orgánico

- **Sigatoka negra o raya negra de la hoja causada por Mycosphaerella fijiensis.** Esta enfermedad es conocida como una de las más dañinas para los cultivos, la cual, dependerá en su mayoría del patrón pluviométrico de las áreas de cultivo. Por otro lado, para su tratamiento se han desarrollado sistemas de aviso en función de las variables climáticas, así como también, se ha intensificado el uso de fungicidas y medidas de saneamiento.
- **Marchitez bacteriana por Ralstonia solanacearum raza 2**. Esta enfermedad también llamada “Moko del plátano” tiene un centro de origen ubicado en la Amazonia, la cual presenta un amplio rango de ataque que difiere con respecto a los hospederos a los cuales ataca (especialmente al banano y heliconias), así como también, con la distribución geográfica, patogenicidad, relaciones epidemiológicas y propiedades fisiológicas.

1.4 Lenguajes de programación

1.4.1 Python

Como su título lo indica, Python es un lenguaje de programación que en los últimos años ha logrado instaurarse como uno de los más populares dentro de los desarrolladores. Este lenguaje fue creado alrededor de 1991 como una iniciativa de heredero del lenguaje ABC. Ha logrado superar a otros lenguajes de programación como MATLAB y el lenguaje R ya que estos están más enfocados en operaciones lineales y análisis de data interactiva respectivamente. Se puede decir que lleva la delantera en las habilidades y capacidades de programación ya que posee distintas herramientas que MATLAB y R no pueden. (Kumar y Panda 2019)

Según lo mencionado antes, Python tiene herramientas conocidas como librerías. Este lenguaje de programación está enfocado en 3 aspectos importantes para la resolución de

problemas, los cuales son: programación orientada a objetos, programación imperativa y programación funcional.

Cabe resaltar que Python tiene bastantes similitudes al lenguaje C ya que este ha influenciado mucho a los componentes que Python tiene, por ejemplo, librerías iterativas, definición de clase, funciones, etc. (Kumar y Panda 2019)

Se ha posicionado como uno de los lenguajes más usados ya que cualquier programador puede adaptarse, además, tiene la capacidad de que los códigos creados por distintos programadores alrededor del mundo puedan ser usados de nuevo con la intención de crear un repositorio de información y a la vez una comunidad conformada por estos programadores donde tienen la posibilidad de compartir ideas y de ayudarse en proyectos mutuos.

1.5 Machine Learning

Machine Learning o aprendizaje automático es una rama de la Inteligencia Artificial, en donde la computadora incrementa su conocimiento para cumplir una tarea; a través de algoritmos tiene la capacidad de identificar patrones en datos masivos para hacer predicciones; permitiéndoles ser autónomas, sin necesidad de ser programados.

Se asume un enfoque de agente autónomo; es decir, para indicar un individuo, un programa, un artefacto, robot, etc. que esté bajo observación cuando realiza una tarea cognitiva identificable, como aprender y realizar una misma tarea de varias maneras, si es posible, y dependiendo de las circunstancias. Capaz de identificar el curso más apropiado para la resolución de la tarea, además de modificar sus decisiones cuando las condiciones así lo requieran.

La selección junto la adaptación caracteriza el proceso de aprendizaje automático; el sistema selecciona las características más relevantes de un objeto y las compara con otras conocidas. (Moreno y Armengol, 1994)

La clasificación de los métodos de aprendizaje según su estrategia y las ayudas que recibe un sistema de aprendizaje es:

- **Aprendizaje supervisado.** Los ejemplos proporcionados como entrada son necesarios para cumplir las metas del aprendizaje, otorgando ejemplos y especificando de qué concepto son. Los algoritmos con este aprendizaje previo asocian los datos y le permite tomar decisiones o hacer predicciones. Un ejemplo es el detector de spam en los correos electrónicos, que ha aprendido a reconocer patrones del historial de correos.
- **Aprendizaje no supervisado.** Los algoritmos no cuentan con un conocimiento previo, por lo que están diseñados para desarrollar nuevos conocimientos mediante el descubrimiento de regularidades en los datos (*data-driven*).

- **Aprendizaje mediante refuerzos.** Este aprendizaje se encuentra entre los dos anteriores y a medio camino. Su objetivo es que un algoritmo aprenda a partir de la propia experiencia, capaz de tomar la mejor decisión ante diferentes situaciones, utilizando un proceso de prueba y error.

1.5.1 Deep Learning

Deep Learning o aprendizaje profundo es una forma de aprendizaje automático que permite a las computadoras aprender de la experiencia, recopilando los conocimientos a través de esta y comprendiendo el mundo en términos de una jerarquía de conceptos; permite a la computadora aprender conceptos complicados construyéndolos a partir de otros más simples; este conjunto de niveles son parte de lo llamado Red Neuronal Artificial (RNA).

1.5.1.1 Redes neuronales convolucionales. En los últimos años, la inteligencia artificial se ha desarrollado tanto que ha logrado grandes avances, por ejemplo, una herramienta conocida como las redes neuronales convolucionales que se consideran una arquitectura de *Deep Learning* (Q. Zhang 2018), las cuales han logrado sobresalir frente a las ya conocidas redes neuronales artificiales. Se han desarrollado de tal manera que han logrado resolver problemas de tal complejidad que antes no se pensaba que se podrían resolverse mediante estas herramientas (Albawi, Mohammed, y Al-Zawi 2017). Una década atrás todavía se pensaba que la inteligencia artificial no podía superar al ser humano ya que su error era mayor al error humano, pero ahora eso ha cambiado siendo el error de este tipo de redes de solo 3%.

Una de las características más importantes de las redes neuronales convolucionales es que su metodología de trabajo para el procesamiento y detección de imágenes, es decir, no toma en consideración donde se encuentre el objeto en la imagen sino en reconocer la imagen cual sea su posición. (Albawi et al. 2017)

Otro aspecto importante de las redes neuronales convolucionales es que está conformada por capas donde cada una de ellas conforma lo que se conoce como una capa de convolución también se les pude considerar como filtros, en ellos cuales se mantiene una especie de jerarquía, por ejemplo, la primera capa siempre se encarga de detectar los bordes y conforme avancen estos filtros las características que se detectarán en las imágenes serán cada vez más importantes. (Teuwen y Moriakov 2019)

Este tipo de arquitectura no discrimina cual sea el origen de la data, además una de las cosas que la caracterizan es que tiene la posibilidad de extraer ciertas características de la imagen con el objetivo de obtener mejores resultados, algo que otras arquitecturas no tienen.

Además, se puede rescatar que una de las características más importantes es la invariación de traducción pues los filtros son independientes de la locación.

De esta manera las funciones se pueden ser más eficientes y la cantidad de parámetros se pueden reducir notablemente eso hará que como consecuencia la rapidez de la red neuronal convolucional sea mayor.

El tamaño de la data ya puede pasar a un segundo plano porque hay métodos y herramientas en las cuales se puede optimizar la cantidad de datos y obtener el mismo resultado si la data fuera extensa.

Las redes neuronales convolucionales están arregladas de tal manera que se distinguen las siguientes dimensiones. (Teuwen y Moriakov 2019)

- Canales
- Profundidad
- Altura
- Número de filtros

Los bloques más comunes que se utilizan para componer la estructura de una red neuronal convolucional son: la capa convolucional, la capa de *pooling* y las capas interconectadas. Entre sus tareas se encuentran la reducción de las dimensiones y la clasificación de estas capas. Se usan una serie de filtros que con el entrenamiento de la red neuronal convolucional logrará que la eficiencia de estos filtros mejore y que puedan llevar a cabo su tarea como las que se ha mencionado anteriormente como la de detección de bordes u orientación. (Teuwen y Moriakov 2019)

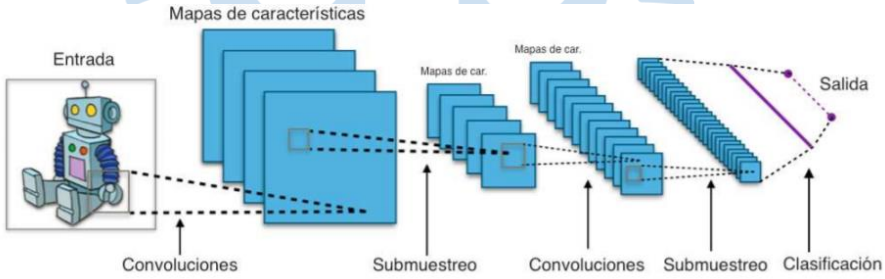

Figura 1. Estructura base de una red neuronal convolucional.

Fuente: Redes Convolucionales en R, Jaime Durán Suárez, 2017.

1.5.1.2 Estructura de red neuronal convolucional

- **Capa de entrada.** La capa de entrada se puede definir como la primera capa, es decir, la puerta principal de la red neuronal convolucional. Se usa como modo de entrada imágenes que pueden estar en distintos formatos, pero depende lo que se quiera trabajar para poder realizar la estandarización pertinente. (Durán Suárez et al. 2017)

Otra manera de verlo es que, al ingresar la imagen a la red, está la ve como una matriz que tiene valores de pixeles. Lo que la computadora vea despenderá del tamaño de la imagen y como se ha explicado antes estará comandada por dimensiones de altura, ancho y profundidad, estos se determinan como valores del canal RGB que pueden ir dentro del rango de 0-255 pixeles lo cual se puede entender como una escala de grises. (Q. Zhang 2018)

Figura 2. Ejemplo de imagen de entrada a la red neuronal convolucional.

Fuente: Redes neuronales convolucionales en R, Jaime Durán Suárez, 2017.

- **Capa convolucional.** En esta capa se tiene como principal objetivo reducir la cantidad de parámetros que conforma la imagen, para que el análisis de la imagen se centre en las características más importantes de esta. Lo anteriormente comentado se basa en un principio matemático el cual se basa en las constantes operaciones de suma y multiplicación entre la imagen y un filtro o *kernel.* (Durán Suárez et al. 2017)
  Otro aspecto que es importante resaltar es que también busca que el desarrollo de esta red no sea muy pesado y que se pueda realizar desde cualquier computadora, por lo que el análisis de las imágenes se lleva a cabo sin analizar todos los elementos que la componen en la capa oculta sino, lo que se realiza es designar cierta cantidad de pixeles de la imagen a las neuronas de la capa oculta, de esta manera se reducen las conexiones y la carga ya no sea tan pesada. (Durán Suárez et al. 2017) (Q. Zhang 2018)

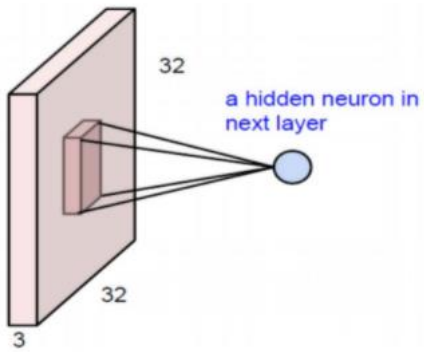

Figura 3. Ejemplo de Capa Convolucional.

Fuente: Understanding of a convolutional neural network, Saad Albawi, Tareq Abed Mohammed, 2017.

- ***Stride.*** Se considera como una herramienta de reducción de parámetros para disminuir la carga computacional. Básicamente hace referencia a un filtro o *kernel* y se caracteriza por evitar el sobre posicionamiento, esto dependerá del tamaño del filtro aunque en mucho de los casos estos tienen una dimensión de 3x3 (Albawi et al. 2017). Un factor importante es el paso con el cual el filtro pasará sobre la matriz, es aquí donde entra el *stride* pues si este tiene un valor de uno, el paso será de uno y si el stride tiene un valor a dos, el paso será de dos y así sucesivamente.

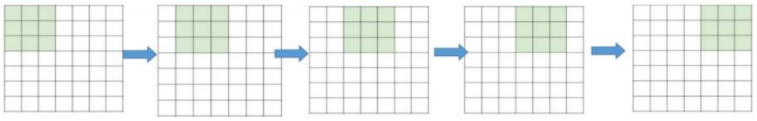

Figura 4. Ejemplo de paso de *stride* sobre una matriz 7x7.

Fuente: Understanding of a convolutional neural network, Saad Albawi, Tareq Abed Mohammed, 2017.

- ***Padding*.** Es visto también como una herramienta en las redes neuronales convolucionales que a diferencia del *stride* tiene como objetivo lograr que la información en los bordes de la imagen no se pierda, además con el *padding* nos aseguramos de que la imagen no sufra de reducciones que podría causar un mal análisis de la imagen. Esto también puede lograr a través de *zero-padding.* (Albawi et al. 2017)
- **No linealidad.** La no linealidad es una de las herramientas dentro de las redes neuronales convolucionales. Sin esta, el proceso de pensamiento de la red sería muy lento y al ser lineal su toma de decisiones sería muy restringido. (Teuwen y Moriakov 2019)

  Hay distintas definiciones de la no linealidad y cada una tiene sus ventajas como desventajas, sin embargo, al estar trabajando con capas intermedias no nos conviene trabajar con dos de ellas debido a la capacidad de hacer desaparecer el gradiente lo cual podría hacer más difícil la optimización, estas dos serían sigmoide y tangente hiperbólica, ambas formas de no linealidad.

  Habiendo visto las definiciones que no funcionan con las capas intermedias, llegamos a la conclusión que la más adecuada es la definición de ReLu, la cual está definida de la siguiente manera.

$$\mathrm{ReLU}(x) = \max(0, x), \quad x \in \mathbb{R}.$$

Figura 5: Definición de ReLu.

Fuente: Convolutional neural networks, Jonas Teuwen, Nikita Moriakov, 2019.

El beneficio de esta función es que ayuda a que la convergencia se lleve más rápido, pero es solo aplicable a procesos donde se sepa muy bien el peso de iniciación y el porcentaje de aprendizaje. (Teuwen y Moriakov 2019)

Aparte de la función ReLu (Bhandare et al. 2016) se encuentra la función *Softmax*, la cual se considera que es mucho más focalizada y especializadas (Teuwen y Moriakov 2019) que las mencionadas anteriormente. Su definición matemática es la siguiente:

$$\operatorname{softmax}(x)_i := \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}, \quad x \in \mathbb{R}^n,$$

Figura 6. Definición de función *Softmax.*

Fuente: Convolutional neural networks, Jonas Teuwen, Nikita Moriakov, 2019.

La función *Softmax* se encargará de predecir el resultado de la imagen, sin embargo, no se pude considerar que dentro de sus capacidades esté el poder predecir la incertidumbre o sus resultados ya no serán tan acertados si hay alguna incertidumbre como algún tipo de disturbio en la imagen. Este tipo de función se encuentra en la fila de la capa totalmente conectada.

- **Capas de *pooling*.** Tiene como objetivo reducir el espacio dimensional sin perder información importante de la imagen que ayudaría en el análisis de esta (Bhandare et al. 2016). Esto se debe a que después de pasar por la capa de convolución, la imagen estaría preparada para utilizar el clasificador que se escoja, sin embargo, se tiene que tener en cuenta que estos traen desventaja en cuanto al peso computacional. Es aquí donde las capas de *pooling* demuestran su gran importancia. (Bhandare et al. 2016)

  Dentro de estas se encuentran distintos tipos de capas de *pooling* teniendo en cuenta que cada una de ellas tiene una función distinta. Dentro de ellas está el *max pooling*, la cual básicamente se refiere a que se tome el valor máximo de cada punto ya sea que sus dimensiones se encuentren en 2D o 3D. El *average pooling* que se diferencia del anterior es debido que en vez de tomar el valor máximo toma un promedio y se da eso como resultado. (Teuwen y Moriakov 2019)

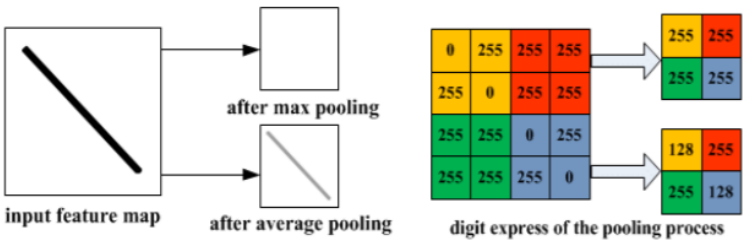

Figura 7. Función *max pooling*.

Fuente: Redes neuronales convolucionales en R, Jaime Durán Suárez, 2017.

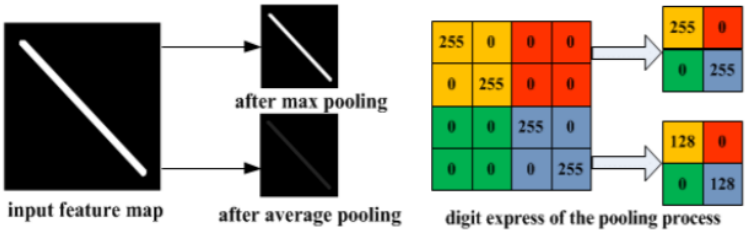

Figura 8. Función de *average pooling*.

Fuente: Redes neuronales convolucionales en R, Jaime Durán Suárez, 2017.

- **Capas completamente conectadas.** Es una de las capas más importantes y a la vez la última que se va a encontrar en la estructura de la red neuronal. Su característica más importante es que tiene la capacidad de determinar la clasificación de una imagen, es decir, en esta capa se podrá saber si la red neuronal ha sido bien programada para que así la imagen pueda ser asociada con su respectiva clase. Una de las condiciones de esta capa es que tiene un número de neuronas igual al número de clases. Esta capa se encuentra específicamente después de la última capa de *pooling* (Durán Suárez et al. 2017). Además, debe su nombre y la alta probabilidad de realizar la correcta clasificación es debido a la detallada y completa interconexión entre las neuronas de la última capa y los elementos que componen a la matriz. (Bhandare et al. 2016)

Arquitecturas de redes neuronales. Arquitecturas o estructuras se les denominan a las conexiones en una red neuronal. Una red neuronal artificial contiene nodos lo cuales se conectan por medio de sinapsis, y el comportamiento de la red es determinada por la estructura de las conexiones sinápticas, las cuales son direccionales, es decir, solo pueden propagarse en un solo sentidos. Las unidades estructurales de las neuronas se denominan capas, el conjunto de estas constituyen la red neurona.

Según la organización de las neuronas en la red formando capas o agrupaciones, se consideran dos tipos de arquitecturas básicas; redes monocapa y redes multicapa.

- **Redes monocapa.** Organizadas por una sola capa. Cada neurona está conectada con todas las demás que forman la estructura. Este tipo de red se utiliza para tareas auto asociativas, es decir, intentan asociar una información consigo misma. En la etapa de entrenamiento se almacena ciertas informaciones en los pesos de la red, por consiguiente, cuando se presenta una información en la entrada de la red, esta responde con información parecida a las almacenadas. Este tipo de red es útil para regenerar información de entrada que se encuentren distorsionadas o incompletas. (Palmer,A; Montaño, J.J.)
- **Redes multicapa.** Disponen de conjunto de neuronas agrupadas en dos o más capas. Mayormente este tipo de red está formada por una capa de entrada, una capa de salida y una o más capas intermedias u ocultas. Las tareas que realizan son denominadas heteros asociativas; intentan asociar pares de informaciones distintas

con las parejas de datos que se la da a la red en su entrenamiento. Este tipo de redes son útiles para la clasificación de patrones, donde se asocia una información de entrada con otra información de salida. (Palmer,A; Montaño, J.J.)

1.5.1.3 Estandarización de data. Dentro de lo métodos para la estandarización de data podemos encontrar al *subtraction* el cual puede llevarse a cabo de dos maneras, el primero es teniendo en cuenta que para poder realizarse todas las imágenes deben tener el mismo tamaño. Con lo anteriormente mencionado es mejor inclinarse por la segunda manera la cual no se necesita estandarizar el tamaño de las imágenes, además de ser una de las técnicas más populares. Esta se basa en sólo calcular la media por canal del *training data set* y de *testing data.* (convolutional neural networks, jonas)

Otro método que se encuentra es el *scaling*, el cual tiene como objetivo calcular las desviaciones estándar del training data set, luego esta entrada de datos se divide en tres los valores hallados para así obtener un valor de 1 en cada uno de los canales. (convolutional neural networks, jonas)

1.5.1.4 Data Augmentation. Existen varias técnicas, ayuda mucho a crear variaciones de la data a partir de la misma data. Es además de evitar que se produzca un *overfitting*. No es necesario usar todas estas técnicas ya que dependerá si es necesario o del resultado que se quiera lograr.

- **Rotación.** Su caracteriza principal como su nombre lo indica es que es capaz de dar vuelta la imagen tanto en una dimensión como en dos. Lo más común es que la imagen solo rote en la dimensión horizontal (Q. Zhang 2018). Hay otro tipo en la cual se puede rotar la imagen mediante un ángulo que puede estar entre los valores de 90, 180 y 270 grados.
- **Corte.** Consta del redimensionamiento de la imagen ya que esta consta de una altura y de un ancho, se procede a reducir la imagen tanto en el eje x como en el y donde ambos valores nuevos pertenezcan a estos valores de altura y ancho originales. Otra cosa que se debe tener en cuenta de esta técnica es que es capaz de reducir la imagen, pero sin variar la cantidad de píxeles de la imagen original. (Teuwen y Moriakov 2019)
- **Aumento de color y transformación gamma.** La primera técnica cambia ciertos parámetros de color para poder saber cuánto o no varía según la exposición a ciertos cambios en la intensidad de luz, mientras que la transformación gamma se basa en el ajuste del contraste de la imagen. (Teuwen y Moriakov 2019)

1.5.2 Algoritmos de Clasificación

1.5.2.1 Random Forest Classifier (RF). Es un meta estimador que ajusta un número de árboles de decisión en varios subgrupos de la base de datos y promedia los resultados para mejorar la precisión en la predicción y controlar el *overfitting*. El algoritmo es útil para realizar tareas de regresión y clasificación. Cada árbol crea un árbol de decisión seleccionando *k*

características de las *m* totales. Luego, crea *n* número de árboles variando la cantidad de características y muestras que se le pasan a los árboles de decisión. Luego se toman cada uno de los árboles generados y se le pide al sistema realizar una misma clasificación, obteniendo n salidas. Finalmente, se evalúan los resultados y se toma como salida la moda. (Anón 2020)

Ventajas:

- Buen funcionamiento sin ajuste de hiperparámetros (valores usados para el entrenamiento de los modelos).
- Respecto a la aplicación, aborda dos problemas de interés como lo son clasificación y regresión de datos.
- Se reduce el riesgo de *overfitting* por el uso aleatorio de múltiples arboles de decisión.
- Estabilidad frente a la aparición de nuevas muestras, por el uso de la moda como el factor de decisión del algoritmo. (Bagnato 2017)

Desventajas:

- Dependiendo de la naturaleza de los parámetros de entrada, un sobreajuste del modelo de decisión es posible.
- Dada la mayor cantidad de árboles presentes en el modelo, aumenta el coste computacional en grandes muestras de datos (equipos dedicados al entrenamiento por un periodo más prolongado).
- Dificultad en la interpretación de la estructura y criterios de decisión del modelo. (Bagnato 2017)

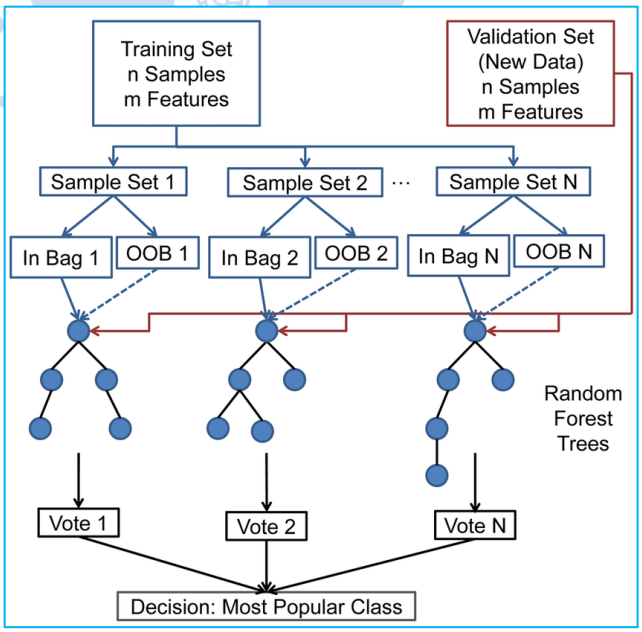

Figura 9. Estructura de algoritmo de clasificación Random Forest Classifier

Fuente: "Arboles de decisión y Random Forest"; Orellana, J. (2018).

2.5.2.2 K-Nearest Neighbors (KNN). Es uno de los métodos de clasificación más sencillos y fundamentales cuando se trabaja con datos. Suele ser una de las primeras opciones al momento de estudiar la naturaleza de la base de datos. Fue desarrollado por la necesidad de realizar un análisis discriminante cuando la cantidad de parámetros confiables para estimación son desconocidos y difíciles de determinar.

El parámetro clave dentro de la clasificación de los datos es la distancia geométrica entre muestras o distancia Euclidiana entre un conjunto de datos y un conjunto de prueba de datos. Siendo x_i una de las entradas del sistema, con p parámetros$(xi1, xi2, \dots, xip)$, n el número total de entradas y p el numero total de parámetros, la distancia Euclideana entre la muestra x_i y un dato $x_l(xl1, xl2, \dots, xlp)$ se define como en la figura 6. (Peterson 2009)

$$d(\boldsymbol{xi}, \boldsymbol{xl}) = \sqrt{(xi1 - xl1)^2 + (xi2 - xl2)^2 + \cdots + (xip - xlp)^2}$$

Los vecinos o datos cercanos son representados por la variable K y son los que determinaran la clase del elemento desconocido x_l tomando como referencia el valor de la distancia previamente calculado. Una técnica de uso recurrente para la clasificación de los elementos es la decisión por medio de distancias ponderadas, donde una función Voto de la clase se determina por medio de la suma de las inversas de las distancias calculadas desde el valor desconocido, como lo explica la ecuación siguiente.

$$Voto(y_j) = \sum_{c=1}^{k} \frac{1}{d(x_l, x_c)^n} 1(y_j, y_c)$$

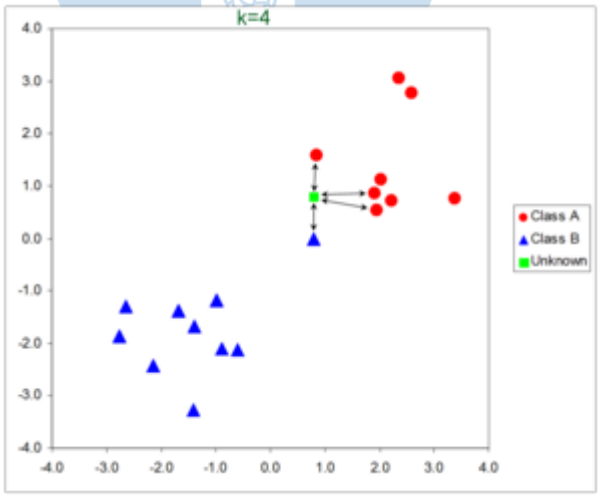

Figura 10. Muestras donde se comparan distancias Euclidianas para 4 muestras cerca del punto desconocido.

Fuente: "*K-nearest neighbor*"; Peterson, L. (2009).

La función $1(y_j, y_c)$ da un valor de 1 si las clases coinciden o caso contrario se determinará a 0. El valor del exponente n se asume como 1 de manera convencional, pero puede aumentarse para reducir la influencia de elementos más distantes al dato seleccionado. (Cunningham y Delany 2007)

Ventajas:

- El algoritmo es sencillo de implementar en la mayoría de *data sets*.
- No es necesaria la construcción de un modelo, el ajuste de una gran cantidad de parámetros o realizar suposiciones adicionales.
- Es un algoritmo versátil, pues puede emplearse para clasificación, predicción o búsqueda de parámetros.

Desventajas:

- El algoritmo suele emplear un mayor tiempo de ejecución a medida que el número de parámetros, predictores o variables independientes aumenta. Esto sucede pues será necesario reajustar los parámetros como n o k, incrementando la cantidad de cálculos para un dato en particular. (Harrison. Onel 2018)

2.5.2.3 Support Vector Classifier (SVC). Es un algoritmo desarrollado para la obtención de un espacio n-dimensional que posibilite la separación de los ejemplos positivos con relación a los negativos obtenidos durante el periodo de entrenamiento, maximizando así la brecha entre los casos positivos y negativos. En conclusión, un clasificador SVC binario puede ser interpretado en el espacio como un hiperplano que delimita los ejemplos negativos y positivos, los mismos que pueden ser tomados como puntos separados por un margen específico. El margen de separación para los puntos puede ser tomado como una medida relacionada a la tolerancia al error, puesto que un clasificador seleccionará de manera adecuada una instancia de prueba si posee un margen para el error amplio.

Este tipo de algoritmo ofrece una serie de ventajas con respecto a la categorización de texto, ya que, permite una optimización global, así como también, ayuda a evitar los defectos relacionados con el estrés de pruebas para espacios de dimensiones grandes. (Santana Mansilla, Costaguta, y Missio 2014)

1.5.3 Hiperparámetros

Son aquellos que un programador deberá ajustar para el correcto desempeño del algoritmo de aprendizaje desarrollado, ya que, de esta manera el modelo empleado será capaz de resolver el problema del aprendizaje automático. De esta forma, el programador se ve en la obligación de ajustar una serie de hiperparámetros en el entrenamiento de cualquier red neuronal; asimismo, los principales hiperparámetros a evaluar son los siguientes:

- **Número de épocas.** Es el valor entero que define el número de periodos de tiempo sobre los cuales se da el aprendizaje de una red, la cual, ira evolucionando en la medida que pasan los diferentes *batch* del conjunto de datos. Cabe resaltar que, una vez concluida una época de la red, se determinara un error especifico, el mismo que se buscara reducir conforme pasen las siguientes épocas; por lo tanto, el número de épocas se puede definir como el número de veces necesario que se pasan los datos de entrenamiento para lograr el aprendizaje reduciendo los errores.
- **Batch Size.** Se define como *batch size* al tamaño del vector de datos o números que serán entrada para la red neuronal, el mismo que determinará el tiempo de ejecución para una red, asimismo, tendrá un factor clave en la calidad de las predicciones de la red
- **Factor de aprendizaje.** Representa la rapidez con que se desarrolla el aprendizaje en una red, asimismo, indica el valor porcentual en el que se permite la variación de los pesos de una red por época**.**

  Para valores altos de factores de aprendizaje, habrá un acercamiento acelerado a los valores correctos de los pesos, sin embargo, existirá un movimiento constante alrededor de ese valor, debido a la falta de un ajuste fino. Por otro lado, valores bajos de factor de aprendizaje, provocarán un ajuste de datos lento, por lo cual, será necesario seleccionar un factor de aprendizaje óptimo para cada problema concreto. (Hernández et al. 2019) (Montesdeoca Santana 2016)

1.5.4 Optimizadores

- **Descenso estocástico de gradiente (SGD).** Al igual que el backpropagation tiene la función de disminuir, pero teniendo en cuenta que para el primer caso es más recomendable hacerlo cuando la data que se tiene no es muy extensa. SGD, permite reducir esta pérdida tomando como referencias un *batch* o conjunto de datos más reducido por lo que puede hacer el proceso de toma de decisiones mucho más rápido (González Diez 2019), además el SGD da la posibilidad que pueda seguir la dirección del gradiente negativo así solo haya tomado algunos ejemplos solamente. (Durán Suárez et al. 2017)

$$\theta = \theta - \alpha \nabla_\theta J\left(\theta; \boldsymbol{x}^{(i)}, y^{(i)}\right),$$

Figura 11. Optimizador SGD

Fuente: Redes convolucionales en R, Jaime Durán Suárez, 2017.

- ***ADAM-Optimizer*.** Es uno de los algoritmos de optimización de descenso de gradientes con más popularidad debido a su gran rapidez, la misma que lo posiciona por encima de otros optimizadores importantes permitiendo ser incluido en marcos de redes neuronales comunes, como *Tensor Flow* ,*Caffe* oCNTK.Por otro lado, *ADAM-Optimizer*

es un algoritmo que se basa en estimaciones adaptivas de momentos de orden superior, cuyas actualizaciones son estimadas mediante un promedio móvil entre el primer y segundo momento del gradiente ,asimismo, tiene la suficiente versatilidad tanto para, calcular las tasas de aprendizaje adaptativo para los distintos parámetros y ,para recoger un promedio en bajada exponencial de gradientes pasados (Z. Zhang 2018).

- **Optimizador *Adaptive Gradient Algorithm (*AdaGrad*).*** Es, un algoritmo que escala la tasa de aprendizaje para cada dimensión. De manera práctica, se utiliza la siguiente expresión:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\varepsilon I + diag(G_t)}} \cdot g_t$$

Este algoritmo se desempeña bien para pocos datos por que incrementa la ratio de aprendizaje más rápido si trata con parámetros frecuentes o más lento en caso se trate de parámetros poco frecuentes. Este algoritmo elimina la necesidad de proporcionar manualmente un valor de η, ajustándolo dinámicamente conforme se dé el entrenamiento. Sin embargo, puede darse el caso donde la tasa de aprendizaje se reduzca rápidamente dada la acumulación de gradientes en el inicio del entrenamiento. Como consecuencia, hay un punto donde el modelo no tendrá aprendizaje real ya que η se encontrará cercano a 0. (Gylberth 2018)

1.6 Estado del arte

La detección de macronutrientes y enfermedades en distintos cultivos por medio de clasificación de imágenes es una técnica que se ha venido usando durante los últimos años. Los avances más recientes son del 2017, donde (Tejada y Gara, 2017) desarrollaron una aplicación móvil llamada “LeafCheckIT” usando como algoritmo *Random Forest* con una combinación de DenseSFIT, RGB, CIELAB y YCrCb aplicadas al *data set* para la extracción de características de la hoja de banano orgánico y así identificar la deficiencia de macronutrientes como: Nitrógeno, Fósforo y Potasio. Este modelo obtuvo una precisión de aproximadamente 100%.

(Shah et al. 2019) Proporcionaron un sistema automatizado, utilizando procesamiento de imágenes con Machine Learning supervisado para la identificación de nutrientes en plantas y tomar diversas medidas para obtener sus rendimientos de calidad. (Tran et al. 2019) lograron predecir de manera temprana la deficiencia de nutrientes en las plantaciones de tomate empleando imágenes de las hojas y del fruto usando redes neuronales convolucionales. Se usó Inception-ResNet v2, *Autoencoder* y *Ensemble Averaging* con el objetivo de combinar los modelos y generar la salida deseada. Se obtuvo 87%, 79% y 91% para los 3 modelos respectivamente.

(Wulandhari et al. 2019), estimó la deficiencia de nutrientes presentes en la Okra, cultivo muy común en la india, a partir del aspecto de sus hojas usando una red neuronal

convolucional. Se utilizó *Transfer Learning* y *Fine Tunning.* Además, se utilizó la función *Softmax* y el optimizador Adam logrando una precisión de 95%. Sin embargo, al realizar la etapa de testeo sólo se obtuvo una precisión de 56% por lo que se decidió reducir el número de capas y aumentar el rango de aprendizaje consiguiendo una precisión de 83%.

(Amirtha et al. 2020) aplicaron herramientas de Machine Learning para la detección de deficiencia de nutrientes en cultivos. Su objetivo era crear un vehículo robótico el cual pueda realizar la toma de fotos de las hojas del cultivo que se quiera y luego esta imagen es comparada en una red neuronal convolucional con un data set previamente obtenido. Este procedimiento dará el porcentaje de deficiencia y la enfermedad relacionada a la falta de ese nutriente, estos resultados son mostrados en un *display* además de indicar que tipo de fertilizan.

Capítulo 2
Desarrollo y experimentación

2.1 Metodología

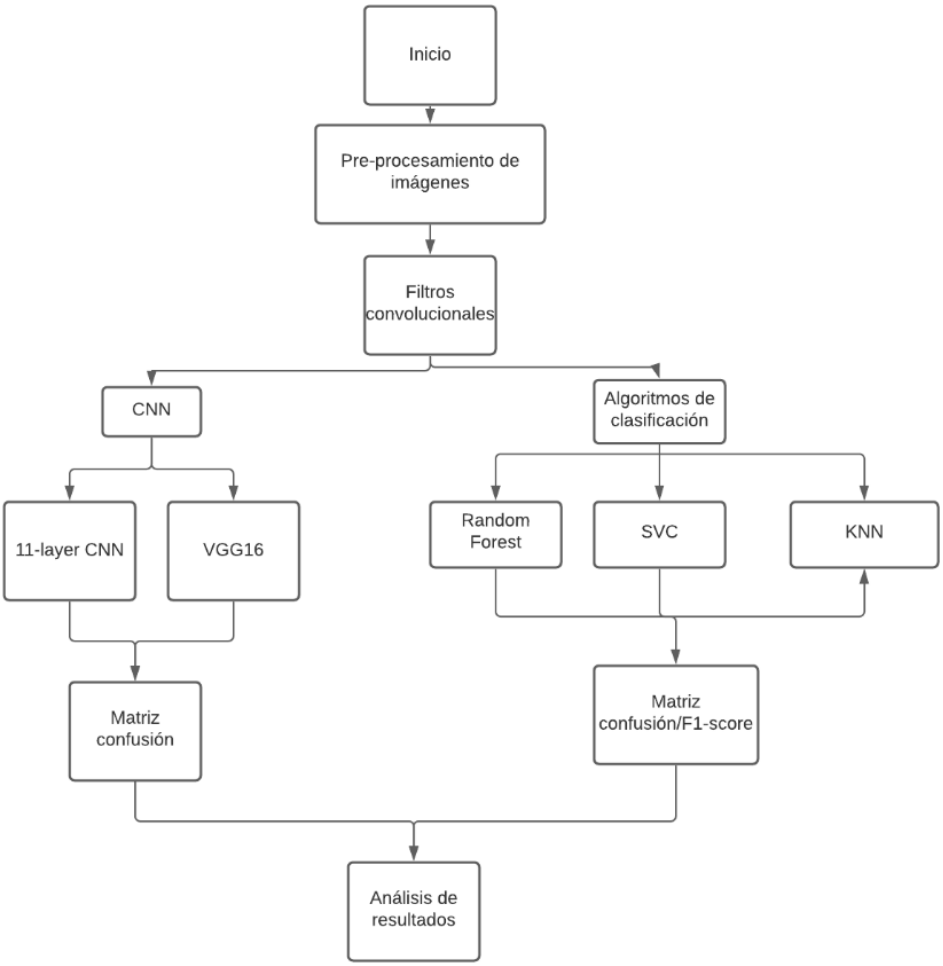

Figura 12. Diagrama de Flujo del Proceso.

Fuente: Elaboración propia.

2.2 Obtención de Data

Se utilizaron dos tipos de *data set* para la realización del presente trabajo. El primer *data set* es un conjunto de datos que comprende 530 imágenes de hojas de banano orgánico

donde se muestra la deficiencia de los macronutrientes más importantes: nitrógeno(N), fósforo(P) y potasio(K). Además, se encuentra una cuarta clase que está compuesta por hojas de banano orgánico sanas. Estas imágenes se encuentran distribuidas en 4 carpetas, obteniéndose 60 imágenes de *healthy*, 220 imágenes de *nitrogen*, 140 imágenes de *phosphorus* y 110 imágenes de *potassium*. Este *data set* de imágenes fue recopilado por un grupo de ingenieros de la Universidad Inmaculada Concepción en Filipinas mediante el uso de un iPhone 5 en la región de Davao del Norte.

Por otro lado, se trabajó con un *data set* de enfermedades presentes en el banano orgánico. Este *data set* se divide en tres carpetas: marchitamiento bacteriano del banano con 220 imágenes (BBW), *Banana black sigatoka* (BBS) con 43 imágenes, además se posee una carpeta de 360 imágenes de hojas sanas o *healthy*. Este conjunto de datos fue obtenido de un repositorio en *GitHub*.

2.3 Programación de Red Neuronal Convolucional

2.3.1 Data enfermedades

A continuación, se explicará de manera detallada el código que se ha desarrollado para la clasificación identificación de macronutrientes en la hoja de banano orgánico.

```
[11] import numpy as np
     import matplotlib.pyplot as plt
     import glob
     import cv2
     from keras.models import Model, Sequential
     from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
     from keras.layers.normalization import BatchNormalization
     import os
     import seaborn as sns
     from sklearn.metrics import confusion_matrix

     SIZE = 128 #Tamaños a los que se reducirá la imagen (Nos puede jugar en contra con pequeñas manchas)
```

Figura 13. Librerías importadas para implementación de algoritmos.

Fuente: Elaboración propia.

La figura 13 muestra la primera celda de código que es utilizada para importar todas las librerías que se necesiten. Se ha importado de *keras* las capas convolucionales como *dense*, *flatten*, *Conv2D* y *MaxPooling2D.*

La capa densa es una capa oculta que tiene como característica entregar todas las salidas de la capa anterior a cada una de las neuronas. *Flatten*, se utiliza para convertir la información en un array plano. *Conv2D*, es un tipo de convolución con don de *kernel* que se caracteriza por tener unas dimensiones diferentes en cuanto a la altura y al ancho que en su mayoría de veces son más pequeñas que la imagen original. Y, por último, las capas de *MaxPooling* tiene la característica de poder disminuir las dimensiones de las imágenes de la capa anterior que se convertirán en la entrada de la siguiente capa convolucional.

```
#Set de Entrenamiento
train_images = []
train_labels = []
for directory_path in glob.glob("/content/drive/My Drive/data enfermedades/*"): #Cambiar con la dirección de Drive
    print(directory_path)
    label = directory_path.split("/")[-1]
    for img_path in glob.glob(os.path.join(directory_path, "*")): #Para todos los elementos con extensión JPG
        #print(img_path)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (SIZE, SIZE))
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        train_images.append(img)
        train_labels.append(label)

train_images = np.array(train_images)
train_labels = np.array(train_labels)

print(len(train_labels))
```

Figura 14. Importación de la Data de entrenamiento para enfermedades.

Fuente: Elaboración propia.

Lo que se está realizando en la Figura 14 es separar las imágenes de entrenamiento. Para esto accedemos a la carpeta de *Google drive* donde se encuentra la data de enfermedades y se copia la ruta de acceso para poder empezar a trabajar. Luego, el resto de esa celda se encarga por pasar por cada una de las imágenes, leerlas, pasarlas RGB a BGR que es simplemente cambiar el orden de los canales de colores. Después de eso, se usa el comando *append* para llevar las imágenes de entrenamiento a las listas vacías de entrenamiento que se definieron en un comienzo de la celda. También se lleva a cabo un proceso de *resize* de la imagen a 128 megapíxeles ya que *para Deep Learning* se considera esta dimensión como la óptima para trabajar.

```
#Rótulos pasaran de Nombres a numeros.
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
#le.fit(test_labels)
#test_labels_encoded = le.transform(test_labels)
le.fit(train_labels)
train_labels_encoded = le.transform(train_labels)
```

Figura 15. Implementación de algoritmo para rotulación de datos.

Fuente: Elaboración propia.

Esta parte del código es muy importante porque permite dejar de trabajar con las etiquetas originales para cada una de las clases para que cada clase sea diferenciada por un número, en este caso del 1 a 3.

```
#Se divide la data en training data set y test data set
#Se divide la data (Usar la función del 20% implica que nos )
#x_train, y_train, x_test, y_test = train_images, train_labels_encoded, test_images, test_labels_encoded
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(train_images,train_labels_encoded,test_size=0.2)
```

```
[56] ###################################################################
     # Dividimos el array a valores entre 0 y 1 (valores típicos por cada pixel)
     X_train, X_test = X_train / 255.0, X_test / 255.0
```

```
[57] #Matriz con 1 donde coincida el rotulo o tipo de datos (1's y 0's)
     from keras.utils import to_categorical
     y_train_one_hot = to_categorical(y_train)
     y_test_one_hot = to_categorical(y_test)
```

Figura 16. División de la data a usar en el algoritmo de clasificación

Fuente: Elaboración propia.

En las celdas de códigos anteriores se puede ver la reasignación de la data de entrenamiento. Los valores de píxeles de las imágenes van dentro de un rango de 0 a 255, cada uno de estos se dividen entre 255 para escalar los valores y conseguir que se encuentren entre el rango de 0 y 1. Se podría trabajar directamente con los valores originales de pixeles de cada imagen, pero podría traer consecuencias al momento de evaluar la rapidez de entrenamiento del modelo.

Ahora se explicará el uso de los filtros convolucionales como herramienta de *Deep Learning*:

```
########### ARQUITECTURA DE RED - N ##################

activation = 'sigmoid' #Evaluar el cambio en la función de activación, si usamos ReLu o Leaky Re

feature_extractor = Sequential()
feature_extractor.add(Conv2D(32, 3, activation = activation, padding = 'same', input_shape = (SIZE, SIZE, 3)))
feature_extractor.add(BatchNormalization())

feature_extractor.add(Conv2D(32, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())
feature_extractor.add(MaxPooling2D())

feature_extractor.add(Conv2D(64, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())

feature_extractor.add(Conv2D(64, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())
feature_extractor.add(MaxPooling2D())

feature_extractor.add(Flatten())
```

Figura 17. Filtros convolucionales de la arquitectura en Red Neuronal

Fuente: Elaboración propia.

En la figura 17, se presenta los filtros convolucionales; inicialmente se agrega la función de activación que se hará de manera secuencial para el modelo que tendrá el nombre *features_extractor* el cuál primero se añadirá una capa convolucional *conv2D* con 32 filtros y después otra vez se agregará otra de 32; esto mismo se repite agregando dos veces una capa de 64 filtros. Además, se agregan las funciones *BatchNormalization* y *MaxPooling2D*, la

primera es la capa que normaliza las entradas y aplica una transformación que mantiene la salida media cercana a 0 y la desviación estándar de salida cercana a 1, y la segunda reduce la representación de entrada tomando el valor máximo sobre la ventana decidida para cada dimensión. Lo último añadido la función *flatten* encargada de "aplanar" las entradas en una sola columna.

```
#A Partir de aqui se usa para redes neuronales.
#Agregar capas para la predicción de deep learning
x = feature_extractor.output
x = Dense(128, activation = activation, kernel_initializer = 'he_uniform')(x)
prediction_layer = Dense(3, activation = 'softmax')(x)
```

Figura 18. Capas para predicción con *Deep Learning.*

Fuente: Elaboración propia.

Para el siguiente paso se desea trabajar con *Deep Learning*, por lo que a la última capa se le asigna como x que estará en función de la salida de nuestro modelo *feature_extractor*. Luego se aplica la función *dense* con una dimensión de 128, ya que es otra forma de agregar capas en lugar de usar la función secuencial. Después, se continua con la capa de salida para la capa de predicción, nombrada como *prediction_layer*, es igual a la función *dense* y dentro de ella el número 3 que representa al número de clases en nuestra data de enfermedades; como es un problema de multi clasificación la activación es *softmax*, y todo esto aplicada a la salida de nuestro modelo de la arquitectura de red que le hemos denominado x.

```
# Hacer un nuevo modelo combinando ambas feature extractor y "x"
cnn_model = Model(inputs=feature_extractor.input, outputs=prediction_layer)
cnn_model.compile(optimizer='rmsprop',loss = 'categorical_crossentropy', metrics = ['accuracy'])
print(cnn_model.summary())
```

Figura 19. Implementación y compilado de nuevo modelo.

Fuente: Elaboración propia.

Para la red convolucional se realiza un nuevo modelo en el cual nuestras entradas son las del modelo *features_extractor* y las salidas son las de la capa de predicción, nombradas como *prediction_layer*. Continuando con este nuevo modelo se le aplica la función *compile* para compilar los datos del diseño de nuestra red como: datos de optimización, pérdidas y medidas de precisión. Finalmente se mostrará el resumen de la red con la función *print*,

```
#Entrenamiento de las CNN con los datos del modelo
history = cnn_model.fit(X_train, y_train_one_hot, epochs=50, validation_data = (X_test, y_test_one_hot))

#graficar el entrenamiento, la precisión de validación y las pérdidas en cada época (Podria usarse TensorBoard para verlo)
#Se grafica el loss segun el numero de épocas o ciclos que realiza el entrenmiento
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(loss) + 1)
plt.plot(epochs, loss, 'y', label='Training loss')
plt.plot(epochs, val_loss, 'r', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Figura 20. Entrenamiento de la red neuronal convolucional.

Fuente: Elaboración propia.

En la figura 20, se usa la función *cnn_model.fit* es para indicar el entrenamiento y ajuste de una red neuronal con las entradas de *x_train* y las salidas y*_train_one_hot*; para 50 épocas debería ser relativamente rápido porque no se tiene demasiada *data* para procesar; y finalmente para validación se usa *y_text_one_hot* y *x_text.*.

En la parte siguiente están los comandos para graficar los resultados de cada época, su entrenamiento, precisión de validación y las pérdidas en cada época.

Como resultados de obtienen las siguientes imágenes:

```
Epoch 10/50
16/16 [==============================] - 137s 9s/step - loss: 0.5135 - accuracy: 0.8032 - val_loss: 0.7187 - val_accuracy: 0.6640
Epoch 11/50
16/16 [==============================] - 138s 9s/step - loss: 0.5247 - accuracy: 0.8032 - val_loss: 0.6849 - val_accuracy: 0.7200
Epoch 12/50
16/16 [==============================] - 141s 9s/step - loss: 0.5188 - accuracy: 0.8153 - val_loss: 0.7624 - val_accuracy: 0.6800
Epoch 13/50
16/16 [==============================] - 138s 9s/step - loss: 0.4991 - accuracy: 0.8133 - val_loss: 0.7369 - val_accuracy: 0.7360
Epoch 14/50
16/16 [==============================] - 138s 9s/step - loss: 0.4817 - accuracy: 0.8353 - val_loss: 0.6889 - val_accuracy: 0.7440
Epoch 15/50
16/16 [==============================] - 139s 9s/step - loss: 0.4621 - accuracy: 0.8373 - val_loss: 0.7449 - val_accuracy: 0.7360
Epoch 16/50
16/16 [==============================] - 142s 9s/step - loss: 0.4764 - accuracy: 0.8353 - val_loss: 1.1845 - val_accuracy: 0.6560
Epoch 17/50
16/16 [==============================] - 139s 9s/step - loss: 0.4600 - accuracy: 0.8333 - val_loss: 0.9502 - val_accuracy: 0.7200
Epoch 18/50
16/16 [==============================] - 140s 9s/step - loss: 0.4227 - accuracy: 0.8414 - val_loss: 0.9818 - val_accuracy: 0.6400
Epoch 19/50
16/16 [==============================] - 140s 9s/step - loss: 0.4665 - accuracy: 0.8353 - val_loss: 1.0722 - val_accuracy: 0.5360
Epoch 20/50
16/16 [==============================] - 142s 9s/step - loss: 0.4297 - accuracy: 0.8514 - val_loss: 0.8458 - val_accuracy: 0.7200
Epoch 21/50
16/16 [==============================] - 139s 9s/step - loss: 0.4628 - accuracy: 0.8273 - val_loss: 0.7850 - val_accuracy: 0.7040
Epoch 22/50
16/16 [==============================] - 139s 9s/step - loss: 0.4637 - accuracy: 0.8173 - val_loss: 0.5505 - val_accuracy: 0.7760
Epoch 23/50
16/16 [==============================] - 139s 9s/step - loss: 0.4218 - accuracy: 0.8534 - val_loss: 0.7067 - val_accuracy: 0.7440
Epoch 24/50
16/16 [==============================] - 141s 9s/step - loss: 0.3872 - accuracy: 0.8554 - val_loss: 0.5190 - val_accuracy: 0.8400
Epoch 25/50
16/16 [==============================] - 139s 9s/step - loss: 0.4182 - accuracy: 0.8353 - val_loss: 0.7879 - val_accuracy: 0.7120
```

Figura 21. Valores desde la época 1 hasta 25 de entrenamiento.

Fuente: Elaboración propia.

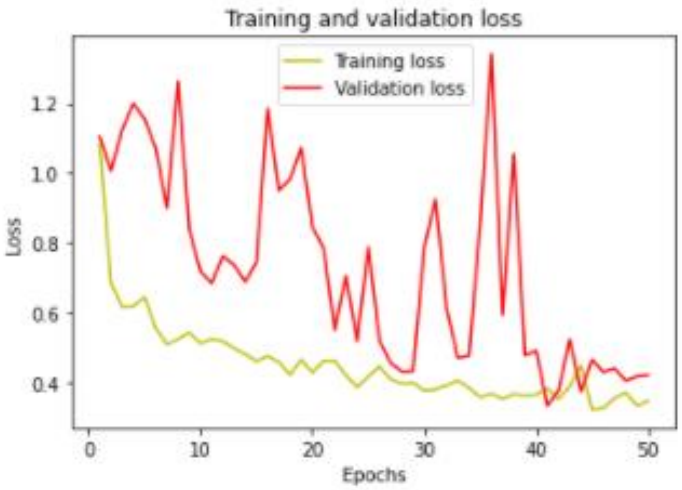

Figura 22. Gráficas de pérdida en red neuronal con 50 épocas -enfermedades.

Fuente: Elaboración propia.

Se muestra parte de los valores obtenidos del entrenamiento, en específico desde la época 1 hasta la 25, si continúa hasta la 50. Los datos de cada época con su entrenamiento y pérdidas se observan en una misma gráfica.

Analizando la gráfica en este caso, la data de entrenamiento lo está haciendo mejor que en el caso de la data de validación, ya que en esta última se presentas muchos picos que indican que la data aparentemente no representa lo que el entrenamiento está tratando de hacer. Al estar abajo la curva de entrenamiento nos da entender que es una buena data para entrenar la red.

La distancia entre curvas no indicaría el tamaño del error, en este caso es una distancia relativamente buena por lo que el error no es tan grande y se puede trabajar con este modelo.

El mejor ajuste será cuando ambas gráficas converjan juntas.

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
plt.plot(epochs, acc, 'y', label='Training acc')
plt.plot(epochs, val_acc, 'r', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Figura 23. Generación de los gráficos de precisión para red con 50 épocas.

Fuente: Elaboración propia.

En la figura 23, se muestra el procedimiento para obtener los gráficos resultantes de precisión de la data de entrenamiento y validación; esto es una estimación inicial.

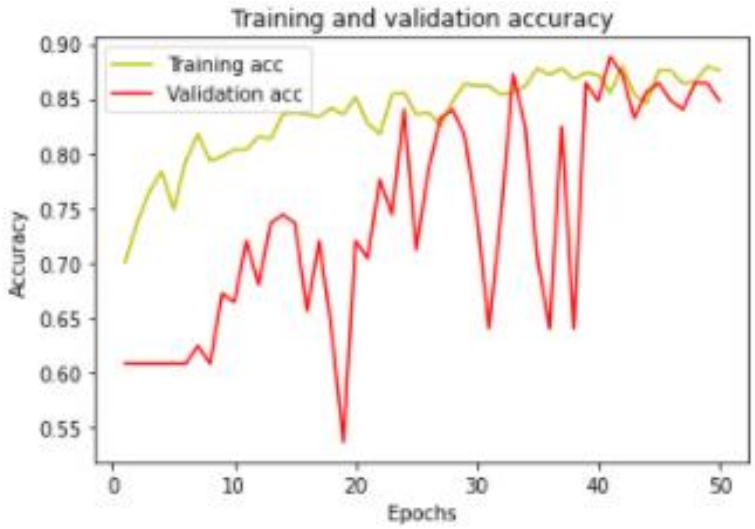

Figura 24. Precisión de entrenamiento y validación para red con 50 épocas – enfermedades.

Fuente: Elaboración propia.

```
prediction_NN = cnn_model.predict(X_test)
prediction_NN = np.argmax(prediction_NN, axis=-1)
#prediction_NN = le.inverse_transform(prediction_NN)

#Matriz confunsión- verificar la precisión de cada clase

cm = confusion_matrix(y_test, prediction_NN)
print(cm)
sns.heatmap(cm, annot=True)

#Verificar los resultados en algunas imágenes seleccionadas


n=9  #Selección del índice de imagen que se desea pobar
img = X_test[n]
plt.imshow(img)
input_img = np.expand_dims(img, axis=0) #Expand dims con la entrada (num images, x, y, c)
prediction = np.argmax(cnn_model.predict(input_img))  #argmax para convertir el categórico de nuevo a original
prediction = le.inverse_transform([prediction])  #Invertir el codificador de etiquetas al nombre original
print("Precision:",acc[len(acc)-1])
print("The prediction for this image is: ", prediction)
print("The actual label for this image is: ", y_test[n])
```

Figura 25. Implementación de la matriz confusión de red neuronal.

Fuente: Elaboración propia.

La predicción del modelo, es sobre la predición del *x_text*, representado como *prediction_NN=cnn_model.predict(x_test)*, sobre la imagen de entrada a probar. Lo que sucede en esta función de predicción es la salida del porcentaje de precisión de la imagen respectiva para cada clase, lo que hace la función *argmax* es otorgar a la imagen en la clase en la cual su precisión es la mayor con respecto a las otras.

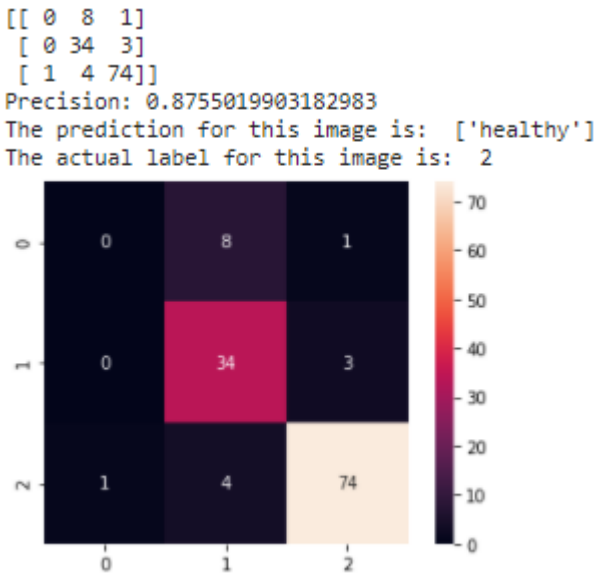

Figura 26. Matriz confusión para data de enfermedades, 50 épocas- enfermedades.

Fuente: Elaboración propia.

La matriz confusión muestra como están sucediendo las cosas para cada clase, por ejemplo, para la primera clase BBS ninguna imagen ha sido identificada correctamente, en la segunda clase BBW 34 imágenes han sido identificadas correctamente; finalmente para tercera clase 74 han sido clasificadas como correctas. En promedios la precisión del modelo es 0.8755.

En este caso en la programación quisimos conocer a que grupo se clasifica la imagen 9 (n=9) y el resultado dado fue que se encuentra en la clase *healthy.*

Modificando el número de épocas a 25 para observar en que cambia, se obtienen los siguientes resultados:

La gráfica de entrenamiento y validación, muestra la presencia de picos en la gráfica de validación, esto significa que los datos de validación son no representativos en comparación con los datos de entrenamiento, también hay una brecha entre ambas gráficas que representaría un error; para mejorar el modelo es recomendable aumentar las épocas para disminuir el error y para la mejora de la validación, aumentar la data podría ayudar.

Figura 27. Gráficas de pérdidas para sistema con 25 épocas. – enfermedades.

Fuente: Elaboración propia.

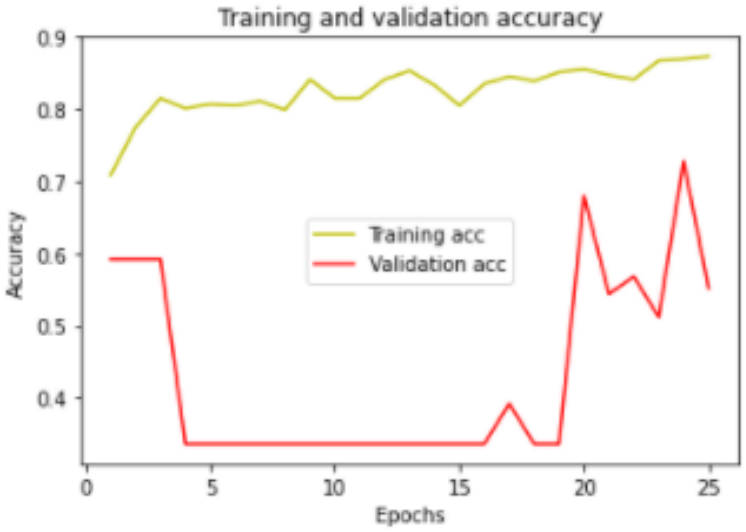

Figura 28. Precisión de entrenamiento y validación con 25 épocas. enfermedades.

Fuente: Elaboración propia.

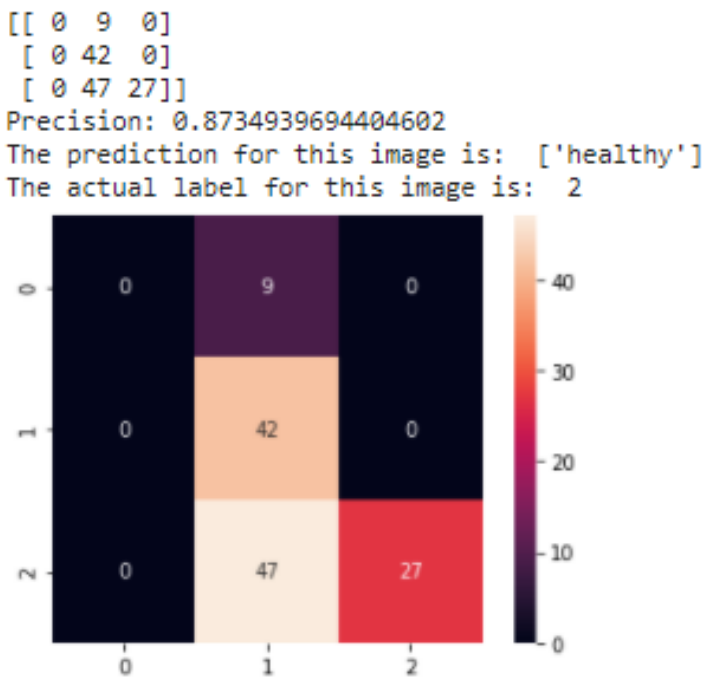

Figura 29. Matriz Confusión.de red neuronal con 25 épocas – enfermedades.

Fuente: Elaboración Propia.

La precisión resultante con 25 épocas es 87.349%, menor que con las 50 épocas. La imagen elegida para la predicción sigue siendo la misma y el resultado es el mismo, clasificada en la clase *healthy*, esto indica que el modelo si funciona, pero no asegura que tendrá una precisión correcta para la clasificación de otras imágenes.

2.3.2 Data de deficiencia de nutrientes

Los pasos en la programación son los mismos que aplicó con la data de enfermedades desde la importación de librerías hasta la matriz confusión, solo que ahora el cambio es en el directorio, el cual será el de la data de deficiencia nutrientes "/content/drive/My Drive/TRABAJO CONTROL/data de nutrientes*/" y para la predicción las capas para la salida serán 4 clases: saludables, nitrógenos, fosforo y potasio; estos indican el nutriente deficiente.

Se utilizó dos números de épocas para realizar comparaciones.

- **Con épocas=25:**

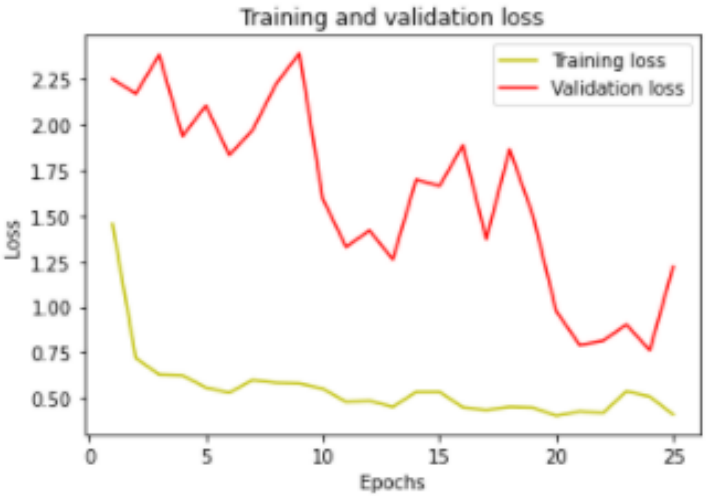

Figura 30. Gráficas de pérdida en entrenamiento y validación con 25 épocas – Enfermedades.

Fuente: Elaboración propia.

Figura 31. Gráficas de precisión. en entrenamiento y validación con 25 épocas – Enfermedades.

Fuente: Elaboración propia.

En este caso las curvas dan a conocer que el modelo no es adecuado para el entrenamiento ya que en principio la distancia de las curva indica que hay un error grande entre los datos de entrenamiento y de validación; el hecho de no converger muestra que el modelo no es el adecuado y se debe ajustar para mejorarlo. El modelo puede ser capaz de aprender, pero necesitará más épocas como en nuestro primer caso para que funcione correctamente

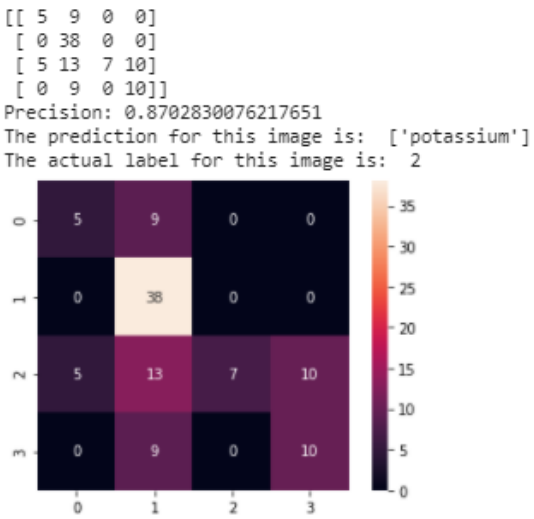

Figura 32. Matriz confusión para data de deficiencia de nutrientes, 25 épocas.

Fuente: Elaboración propia.

La matriz confusión de la figura 32 muestra una precisión del modelo de 87% para la deficiencia de potasio. Sin embargo, por lo anteriormente explicado, sobre las gráficas de entrenamiento y validación que presentan un error considerable, la clasificación se realizará pruebas esta vez con 50 épocas para mejorar el modelo reduciendo el error.

- **Con épocas=50:**

Figura 33. Gráficas de pérdida en entrenamiento y validación con 50 épocas – Enfermedades.

Fuente: Elaboración propia.

Figura 34. Gráficas de Precisión en Entrenamiento y Validación. Con 50 épocas – Enfermedades.

Fuente: Elaboración Propia.

Con el aumento de épocas el error, mostrado entre la brecha de las curvas disminuyó, aun así, las gráficas no convergen, sin embargo, el modelo ha mejorado por lo que se espera también un aumento de precisión, este modelo se puede considerar como valido para trabajar. Los picos bruscos en la curva de validación indican que los datos de validación no son representativos en comparación con la data de entrenamiento, aumentar la data de validación puede solucionar este problema.

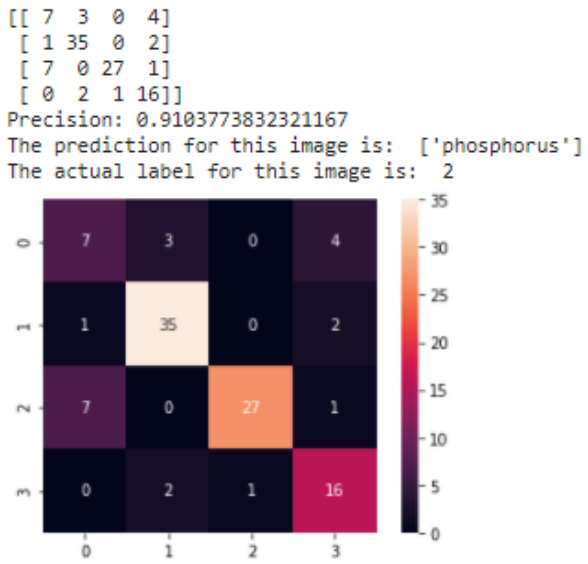

Figura 35. Matriz confusión data de deficiencia de nutrientes-50 épocas.

Fuente: Elaboración propia.

Como se esperaba la precisión ha aumentado y ha diferencia que el de 25 épocas que clasificó la imagen con deficiencia de potasio, este modelo la ha sido clasificado con deficiencia de fósforo, al tener mayor precisión este modelo se optaría por considerar esta la correcta, y además de considerarlo un modelo valido para trabajar.

2.4 Programación del Algoritmos de Clasificación

2.4.1 Filtros Convolucionales de una arquitectura de 11 capas con algoritmo de clasificación Random Forest

Lo que se ha decidido hacer, es trabajar el algoritmo de clasificación por medio de *Random Forest*, pero agregándole filtros convolucionales propio de las redes neuronales convolucionales utilizadas en el *Deep Learning*.

Los filtros convolucionales se han explicado al momento de desarrollar la explicación de la red neuronal convolucional, por lo que se procederá directamente a explicar el algoritmo de clasificación Random Forest:

```
[167] ############### RANDOM FOREST AÑADIDO AL FINAL DE LA ARQUITECTURA  ##################
      #usamos las caracteristicas extraídas de los filtros
      X_for_RF = feature_extractor.predict(X_train) #salida para el random forest

      #RANDOM FOREST
      from sklearn.ensemble import RandomForestClassifier
      RF_model = RandomForestClassifier(n_estimators = 100, random_state = 42)

      # modelo de entrenamiento para la data de entrenamiento
      RF_model.fit(X_for_RF, y_train)

      #Envía la data por el mismo proceso de extracción de características
      X_test_feature = feature_extractor.predict(X_test)
      #predicción usando el modelo de Random forest
      prediction_RF = RF_model.predict(X_test_feature)
      #se recupera etiqueta original
```

Figura 36. Implementación de Algoritmo Random Forest y extracción de características.

Fuente: Elaboración Propia.

Antes de trabajar con el algoritmo *Random Forest* se está utilizando las características extraídas por los filtros convolucionales explicados anteriormente con el fin de solo trabajar con las características más significativas de la imagen.

Luego se procede a importar el clasificador *Random Forest* por medio de *Sklearn.ensemble*. Una parte importante al momento de definir los parámetros del *Random Forest* es el número de estimadores que viene a ser los árboles de decisión que se evaluarán, mientras mayor sea el número de estimadores, mejores resultados se obtendrán; sin embargo, el tiempo de entrenamiento aumentaría. Por otro lado, se encuentra el parámetro *random_state*, el cual permite que se obtengan los mismos resultados al momento de ejecutar los ejemplos ya que sin ellos se consideraría como un algoritmo de árboles de decisión en los cuales la variabilidad es muy alta. Este parámetro permite reducir eso.

Después se realiza el mismo proceso de extracción de características para las imágenes de testeo. Se puede apreciar que la predicción se basará a partir del modelo de *Random Forest* usando las imágenes de entrenamiento con las características extraídas para las imágenes de testeo.

```
#accuracy
from sklearn import metrics
print ("Precision = ", metrics.accuracy_score(y_test, prediction_RF))

#matriz confusion para validar el accuracy de cada clase
cm = confusion_matrix(y_test, prediction_RF)
#print(cm)
sns.heatmap(cm, annot=True)


#resultado de las imágenes seleccionadas
n=68 #Seleccionar la imagen para testear
img = X_test[n]
plt.imshow(img)
input_img = np.expand_dims(img, axis=0)
input_img_features=feature_extractor.predict(input_img)
prediction_RF = RF_model.predict(input_img_features)[0]
prediction_RF = le.inverse_transform([prediction_RF])
y_test = le.inverse_transform(y_test)
print("la predicción para la imagen es: ", prediction_RF)
print("la etiqueta de la imagen es: ", y_test[n])
```

Figura 37 . Continuación de algoritmo Random Forest

Fuente: Elaboración propia.

Y en la última parte del código se puede apreciar cómo se va a sacar el porcentaje de precisión del modelo *Random Forest* usando las características extraídas de los filtros convolucionales. Se escoge una imagen para testeo, luego se expande en dimensiones y en base a las características extraídas empieza a predecir. Se revierte el *label encoder* que se hizo como paso previo al algoritmo de clasificación y explicado en el desarrollo de la red neuronal convolucional con el fin de volver a las etiquetas originales. Por último, se imprime los resultados de la predicción con la ayuda de la matriz confusión, donde se comprobará el buen funcionamiento del algoritmo al mostrar que los valores de la diagonal son mucho mayores que el de los alrededores que son generalmente 0.

Este algoritmo se ha utilizado tanto para la data de nutrientes como la data de enfermedades en el banano orgánico. Se obtuvieron los siguientes resultados:

2.4.1.1 Data de enfermedades. En este caso se ha asignado 0 a la clase BBS, 1 a la etiqueta BBW y 3 a la etiqueta *healthy*. La figura 38 se tiene una precisión del 89.6%. Respecto a la matriz confusión, observamos que ninguna imagen del set de pruebas pudo clasificarse correctamente como BBS, 40 de ellas se clasificaron correctamente como BBW (85% aproximadamente) y 72 de ellas clasificadas correctamente como *healthy* (98.6% aproximadamente). Para este caso no se cumple la premisa postulada líneas arriba respecto a la diagonal mayor.

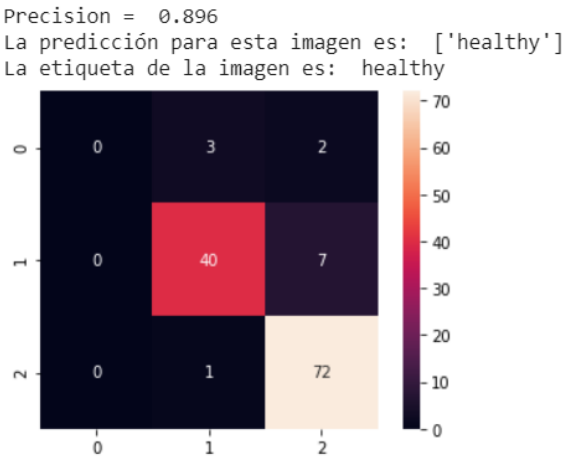

Figura 38. Matriz confusión para enfermedades con precisión del 89.6%.

Fuente: Elaboración propia.

Por otro lado, la figura 38 se tiene una precisión del 83.2%. Siendo esta precisión del modelo menor al 85%, se dice que, si bien el modelo no es completamente robusto respecto a las predicciones que hará, se encuentra cerca del porcentaje aceptable para los modelos (85%) con lo que aún hay un margen aceptable de mejora. La clasificación realizada para esta oportunidad entre el modelo y la etiqueta real se corresponden de manera correcta. Respecto a la matriz confusión, se observa que ninguna imagen de las 43 imágenes pudo clasificarse correctamente como *BBS*, 34 de las 220 se clasificaron correctamente como *BBW* (75.6% aproximadamente) y 70 de 360 clasificadas correctamente como *healthy* (100% aproximadamente). Para este caso no se cumple la premisa postulada líneas arriba respecto a la diagonal mayor de la matriz.

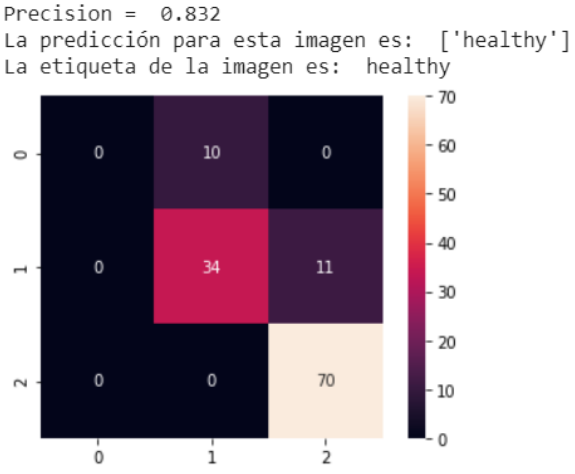

Figura 39. Matriz confusión para enfermedades con precisión del 83.2%.

Fuente: Elaboración propia.

Como se ve en ambas gráficas, la diagonal mayor de la matriz nos da a entender que el modelo no es capaz de diferenciar entre una muestra etiquetada como BBS de muestras *healthy* y BBW. Esto puede deberse a la falta de imágenes referentes a este tipo de datos, con lo que no hay suficiente información relevante en el set de entrenamiento empleado. Finalmente, se puede afirmar que tanto para muestras sanas como para muestras BBW la información es suficiente y los datos para predicción pueden ajustarse correctamente a las predicciones obtenidas en los árboles de decisión.

2.4.1.2 Data de deficiencia de nutrientes. En este caso, se ha asignado 0 a la etiqueta *healthy*, 1 a la etiqueta *nitrogen*, 2 a la etiqueta *phosphorus* y 3 a la etiqueta *potassium*. Según la figura 40 se tiene una precisión del 86.8%. La clasificación realizada entre el modelo y la etiqueta real se corresponden de manera correcta (en este caso, *potassium*). Respecto a la matriz confusión, observamos que 7 de 12 imágenes con la etiqueta *healthy* del set de pruebas pudo clasificarse correctamente, 47 de ellas se clasificaron correctamente como *nitrogen* (97.92% aproximadamente), 23 de ellas clasificadas correctamente con la etiqueta *phosphorus* (100% aproximadamente) y 15 de 23 fueron clasificadas correctamente con la etiqueta *potassium*. En este caso se cumple la premisa postulada líneas arriba respecto a la diagonal mayor.

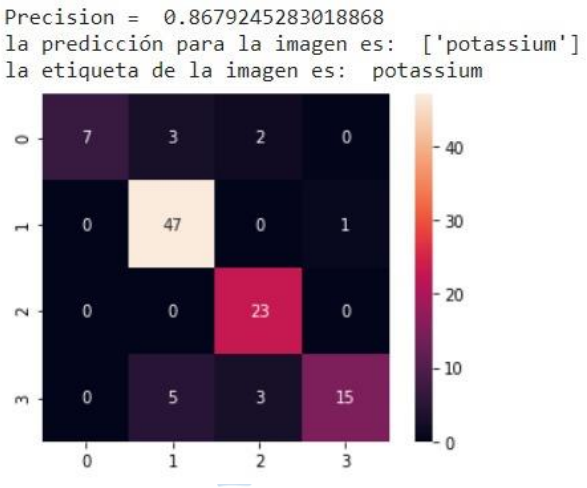

Figura 40. Matriz confusión de deficiencia de nutrientes con precisión de 86.8%.

Fuente: Elaboración propia.

Para la estimación de la figura 40 se tiene una precisión del 90.57%. Siendo esta precisión mayor al 85%, podemos decir que es un modelo robusto y confiable respecto a las predicciones que hará, aunque con un margen de mejora menor pero considerable. La clasificación realizada entre el modelo y la etiqueta real se corresponden de manera correcta (en este caso, *nitrogen*). Respecto a la matriz confusión, observamos que 8 de 12 imágenes con la etiqueta *healthy* del set de pruebas pudo clasificarse correctamente como especímenes sanos (75% del total de estas). 47 de ellas se clasificaron correctamente como *nitrogen* (100%

del total), 26 de ellas clasificadas correctamente con la etiqueta *phosphorus* (100% del total) y 15 de 21 fueron clasificadas correctamente con la etiqueta *potassium*. Se cumple la premisa postulada líneas arriba respecto a la diagonal mayor.

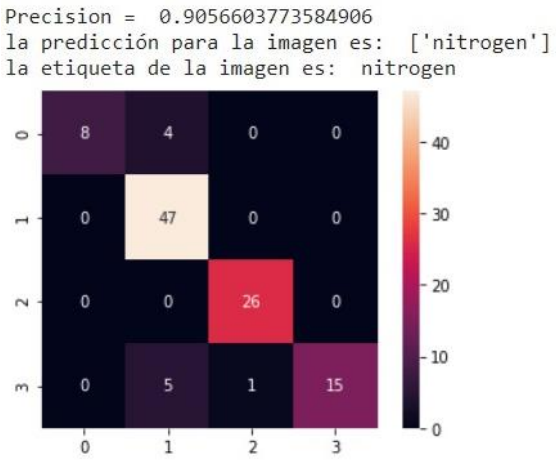

Figura 41. Matriz confusión de deficiencia de nutrientes con precisión de 90.56%.

Fuente: Elaboración propia.

Como se puede observar en ambas matrices confusión, las diagonales mayores nos dan a entender que el modelo cumple al momento de asociar las etiquetas con las predicciones. Si embargo, aún es necesario recopilar información para la etiqueta 3 y la etiqueta 0, dado que las muestras para el *test* aún no se corresponden en cantidad a la de sus pares correspondientes a las etiquetas 1 y 2. Finalmente se puede decir que, de ajustar los parámetros de árboles de decisión o añadiendo capas a los filtros, podremos aumentar la precisión del sistema, robusteciendo aún más la confiabilidad del mismo.

2.4.2 Filtros Convolucionales de una arquitectura de 11 capas con algoritmo de clasificación KNN

Para preparar las entradas que este algoritmo de clasificación utiliza, se utilizan del set de entrenamiento de la red neuronal, utilizando la arquitectura de la red anteriormente detallados en los apartados previos.

2.4.2.1 Data de enfermedades. En la primera celda se importó el módulo de *K-Neighbors Classifier* y se creó el objeto clasificador KNN en el cual se colocará el número de vecinos en este módulo.

Dentro de la función *K-Neighbors Classifier, n_neighbors* es el número de vecinos K, se explicará el porqué de su elección en la siguiente celda; la variable p es el número de clases, en este caso como es sobre la data de enfermedades p es igual a 3 y la métrica que se usará para medir distancias es la métrica euclidiana, la cual es la comúnmente más usada para medir cercanías entre puntos.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

X_para_KNN=feature_extractor.predict(X_train)
X_de_prueba=feature_extractor.predict(X_test)
KNN_model=KNeighborsClassifier(n_neighbors=23, p=3, metric='euclidean')
```

Figura 42. Código de preparación de modelo KNN para data de enfermedades.

Fuente: Elaboración propia

La celda de a continuación es para la elección del K o *n_neighbors* se recomienda utilizar a partir de la raíz cuadrada del número de elementos de la entrada y a partir de ese número elegir su impar siguiente, el cual será el que usará como K; luego se puede ir modificando de tal manera de adecuarlo para una mejor precisión.

```
import math
math.sqrt(len(X_para_KNN))

22.315913604421397
```

Figura 43. Celda para elección de K

Fuente: Elaboración propia

La siguiente celda como se muestra en la figura 44, escala las características de nuestro modelo y es necesario para todo algoritmo que calcula distancia.

```
KNN_model.fit(X_para_KNN,y_train)

KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=12, p=3,
                     weights='uniform')
```

Figura 44. Escala del modelo KNN para data de enfermedades

Fuente: Elaboración propia

Se evalúa la precisión del clasificador con K=23 del modelo de prueba, el cual resulta con una precisión del 78%, considerada una buena precisión para este tipo de clasificador.

```
acc=KNN_model.score(X_de_prueba,y_test)
print(acc)

0.784
```

Figura 45. Precisión del algoritmo para data de enfermedades

Fuente: Elaboración propia

Para la elección de un K adecuado, sobre todo unos puntos de frontera en los que se pueda obtener la precisión más alta, se elige como punto de inicio el valor resultante de K en el código y valores continuos a ese. Se ejecuta el código de la Figura 46 donde se observa los distintos valores K y su precisión obtenida. En este caso el K que nos muestra la precisión más alta es el 23, por lo que fue una buena elección para comenzar.

```
k_range = range(23, 40)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25,30,35,40,45])
```

```
([<matplotlib.axis.XTick at 0x7fd7f723b9e8>,
  <matplotlib.axis.XTick at 0x7fd7f723bda0>,
  <matplotlib.axis.XTick at 0x7fd7f8f87978>,
  <matplotlib.axis.XTick at 0x7fd7f6be4c18>,
  <matplotlib.axis.XTick at 0x7fd7f6bef160>,
  <matplotlib.axis.XTick at 0x7fd7f6be4a90>,
  <matplotlib.axis.XTick at 0x7fd7f6bef710>,
  <matplotlib.axis.XTick at 0x7fd7f6befba8>],
 <a list of 8 Text major ticklabel objects>)
```

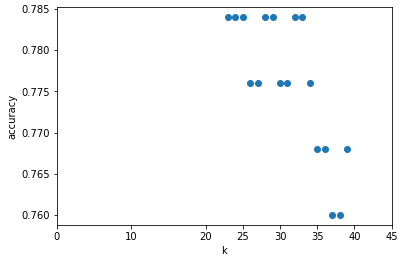

Figura 46. Código y Gráfica resultante de K vs precisión para la data de enfermedades.

Fuente: Elaboración propia

Para confirmar la precisión del modelo, se verá la matriz confusión sobre el conjunto de prueba *(X_prueba)* como se muestra en la figura 47.

```
#Predecir la respuesta para el conjunto de datos de prueba
y_pred = KNN_model.predict(X_de_prueba)
```

```
#Matriz confusión
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm, annot=True)
```

```
[[ 0  3  3]
 [ 0 23 18]
 [ 0  3 75]]
<matplotlib.axes._subplots.AxesSubplot at 0x7fd7f69969b0>
```

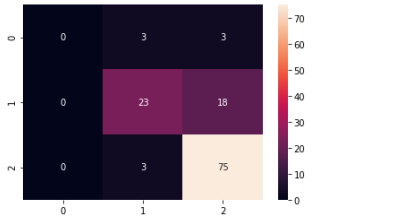

Figura 47. Matriz confusión para data de enfermedades- KNN

Fuente: Elaboración propia

Con lo resultante de la matriz confusión se observa que de nuestro set de prueba ninguna imagen pudo clasificarse correctamente como BBS, en el caso de BBW 23 imágenes se clasificaron correctamente y para la clasificación de *healthy* se clasificaron 75 imágenes.

2.4.2.2 Data de deficiencia de nutrientes. Se realiza el mismo procedimiento del código para la data de enfermedades, lo que lo diferencia serán los valores de las variables dentro de nuestro clasificador. En este caso el número de clases son 4 y además debido al cambio de data el valor de *n_neighbors*=K será distinto.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score

X_para_KNN=feature_extractor.predict(X_train)
X_de_prueba=feature_extractor.predict(X_test)
KNN_model=KNeighborsClassifier(n_neighbors=21, p=4, metric='euclidean')
```

Figura 48. Código de preparación de modelo KNN para data de deficiencia de nutrientes

Fuente: Elaboración propia

Como se mencionó anteriormente en la data de enfermedades, la figura 48 presenta la elección del K, se recomienda utilizar a partir de la raíz cuadrada del número de elementos de la entrada y a partir de ese número elegir su impar siguiente, el cual será el que se usará como K; luego se puede ir modificando de tal manera de adecuarlo para una mejor precisión.

```
import math
math.sqrt(len(X_para_KNN))
```

```
20.591260281974
```

Figura 49. Escala del modelo KNN para data de deficiencia de nutrientes

Fuente: Elaboración propia

El código de la Figura 61 es usado para escalar el modelo:

```
KNN_model.fit(X_para_KNN,y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=21, p=4,
                     weights='uniform')
```

Figura 50. Escala del modelo KNN para data de deficiencia de nutrientes

Fuente: Elaboración propia

La precisión del clasificador con K=21 del modelo de prueba, el cual resulta con una precisión del 78%, considerada una buena precisión para este tipo clasificador.

```
acc=KNN_model.score(X_de_prueba,y_test)
print(acc)
```

```
0.7547169811320755
```

Figura 51. Precisión del algoritmo para data de deficiencia de nutrientes

Fuente: Elaboración propia

Como en la data de enfermedades, para la elección de un K adecuado, se elige como punto de inicio el valor resultante de K en el código y valores continuos a ese para tener un rango y observar la precisión de ese rango de K.

Para confirmar la precisión del modelo se observará la matriz confusión y el reporte sobre nuestra prueba *(X_prueba),* que nos detalla los aciertos y errores de la Figura 52.

```
k_range = range(21, 31)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25,30,35])
```

```
([<matplotlib.axis.XTick at 0x7f9395df1860>,
  <matplotlib.axis.XTick at 0x7f9395df1828>,
  <matplotlib.axis.XTick at 0x7f9395df1be0>,
  <matplotlib.axis.XTick at 0x7f9395da0a58>,
  <matplotlib.axis.XTick at 0x7f9395da0048>,
  <matplotlib.axis.XTick at 0x7f9395da0b38>],
 <a list of 6 Text major ticklabel objects>)
```

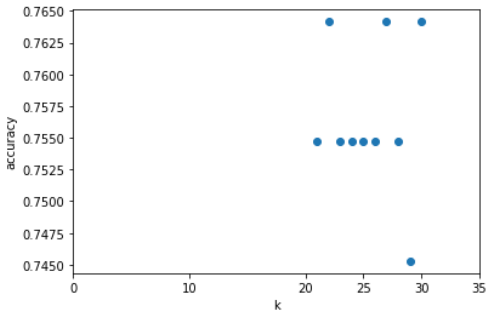

Figura 52. Código y Gráfica resultante de K vs precisión para la data de deficiencia de nutrientes.

Fuente: Elaboración propia.

```
#Matriz confusión
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm, annot=True)
print(classification_report(y_test,y_pred))
```

```
[[ 2  6  1  0]
 [ 0 44  1  1]
 [ 1  0 30  0]
 [ 0  9  7  4]]
              precision    recall  f1-score   support

           0       0.67      0.22      0.33         9
           1       0.75      0.96      0.84        46
           2       0.77      0.97      0.86        31
           3       0.80      0.20      0.32        20

    accuracy                           0.75       106
   macro avg       0.75      0.59      0.59       106
weighted avg       0.76      0.75      0.70       106
```

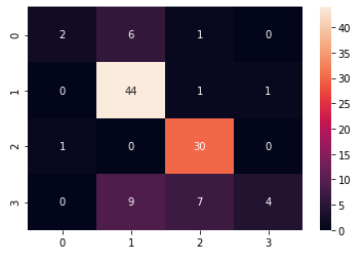

Figura 53. Matriz confusión para data de deficiencia de nutrientes- KNN.

Fuente: Elaboración propia.

Lo que se observa de la matriz confusión es que se clasifican correctamente solo 2 imágenes para la clase *healthy*, en la clase *nitrogen* se clasifican 44 imágenes correctamente, en la clase *phosphorus* 30 imágenes clasificadas correctamente y en la clase *potassium* solo 4 fueron clasificadas correctamente

Observando el reporte, la puntuación *f1-score*, de falsos positivos, es mala para las clases *healthy* y *potassium* con 33% y 32% respectivamente, esto se ve también reflejado en la matriz confusión por la cual sabemos que solo pocas imágenes se clasificaron correctamente de estas clases. En los casos de las clases *nitrogen* y *phosphorus* tienen una puntuación buena de 84% y 86%.

2.4.3 Filtros Convolucionales de una arquitectura de 11 capas con Algoritmo de clasificación SVC

```
import numpy as np
import matplotlib.pyplot as plt
import glob
import cv2
from keras.models import Model, Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
import os
import seaborn as sns
from sklearn import metrics
from sklearn import svm

SIZE = 128 #Tamaños a los que se reducirá la imagen (Nos puede jugar en contra con pequeñas manchas)
```

Figura 54. Librerías importadas.

Fuente: Elaboración propia.

En la figura 55 se han importado las librerías teniendo en cuenta que al trabajar el algoritmo SVC es necesario importar su librería correspondiente para su aplicación más adelante en el código.

```
#Set de Entrenamiento
train_images = []
train_labels = []
for directory_path in glob.glob("/content/drive/My Drive/data enfermedades/*"): #Cambiar con la dirección de Drive
    print(directory_path)
    label = directory_path.split("/")[-1]
    for img_path in glob.glob(os.path.join(directory_path, "*")): #Para todos los elementos con extensión JPG
        #print(img_path)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (SIZE, SIZE))
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        train_images.append(img)
        train_labels.append(label)

train_images = np.array(train_images)
train_labels = np.array(train_labels)
```

```
/content/drive/My Drive/data enfermedades/bbw
/content/drive/My Drive/data enfermedades/bbs
/content/drive/My Drive/data enfermedades/healthy
```

Figura 55. Set de entrenamiento.

Fuente: Elaboración propia.

Se puede apreciar que al igual que en el *Random Forest* se está utilizando el mismo método. Se utiliza se accede a una carpeta en *Google drive* donde se encuentra la data y luego esta pasa por un *resize* para que las imágenes lleguen a tener 128 megapíxeles y luego pasan por los canales RGB. Luego estas imágenes mediante el comando *append* pasan a llenar la lista vacía de entrenamiento ubicada al inicio de la celda de código.

```
#Rótulos pasaran de Nombres a numeros.
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
#le.fit(test_labels)
#test_labels_encoded = le.transform(test_labels)
le.fit(train_labels)
train_labels_encoded = le.transform(train_labels)
```

Figura 56. Rotulación de las imágenes.

Fuente: Elaboración propia.

La figura 56 muestra cómo trabajar con clases presentes dentro del data set, pero asociándolas a un número. En consecuencia, se consigue que la clase BBW sea 0, BBS sea 1 y la clase *healthy* sea 3.

```
[28] #Se divide la data en training data set y test data set
     #Se divide la data (Usar la función del 20% implica que nos )
     #x_train, y_train, x_test, y_test = train_images, train_labels_encoded, test_images, test_labels_encoded
     from sklearn.model_selection import train_test_split
     X_train,X_test,y_train,y_test=train_test_split(train_images,train_labels_encoded,test_size=0.2)
```

Figura 57. División de data set.

Fuente: Elaboración propia.

En la figura 57, se puede observar como la data es divida tanto para data de entrenamiento como en data de testeo. Es importante que haya un buen balance de los datos ya que, de lo contrario, provocará resultados poco favorecedores.

```
[29] ######################################################################
     # Dividimos el array a valores entre 0 y 1 (valores típicos por cada pixel)
     X_train, X_test = X_train / 255.0, X_test / 255.0

[30] from keras.utils import to_categorical
     y_train_one_hot = to_categorical(y_train)
     y_test_one_hot = to_categorical(y_test)
```

Figura 58. Algoritmo de división de data.

Fuente: Elaboración propia.

Se sigue el mismo procedimiento que para el algoritmo *Random Forest* con filtros convolucionales donde cada píxel se divide entre 255, obteniéndose valores entre 0 y 1.

```
activation = 'sigmoid' #Evaluar el cambio en la función de activación, si usamos ReLu o Leaky Re

feature_extractor = Sequential()
feature_extractor.add(Conv2D(32, 3, activation = activation, padding = 'same', input_shape = (SIZE, SIZE, 3)))
feature_extractor.add(BatchNormalization())

feature_extractor.add(Conv2D(32, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())
feature_extractor.add(MaxPooling2D())

feature_extractor.add(Conv2D(64, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())

feature_extractor.add(Conv2D(64, 3, activation = activation, padding = 'same', kernel_initializer = 'he_uniform'))
feature_extractor.add(BatchNormalization())
feature_extractor.add(MaxPooling2D())

feature_extractor.add(Flatten())
```

Figura 59. Filtros convolucionales.

Fuente: Elaboración propia.

Se han utilizado los mismos filtros convolucionales de la red neuronal convolucional. Conforme se vayan cambiando las arquitecturas de red neuronal estos filtros van a ir cambiando y los resultados variarán.

A continuación, se pasará a explicar el algoritmo SVC.

```
[32] X_test_SVC = feature_extractor.predict(X_test)
     X_train_SVC = feature_extractor.predict(X_train)
```

```
[33] from sklearn.utils import Bunch
     from sklearn.model_selection import GridSearchCV, train_test_split
     #Optimización de parámetros
     param_grid = [
       {'C': [1, 10, 100, 1000], 'kernel': ['linear']},
       {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001], 'kernel': ['rbf']},
      ]
     svc = svm.SVC()
     clf = GridSearchCV(svc, param_grid)
     clf.fit(X_train_SVC, y_train)
```

```
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0)
```

Figura 60. Algoritmo SVC.

Fuente: Elaboración propia.

El objetivo de este algoritmo es encontrar un hiperplano óptimo que separa la data de tal manera que este hiperplano no se encuentre ni demasiado cerca ni lejos de cualquiera de los puntos cercanos de esta por lo que se pueden identificar dos parámetros importantes. Uno de ellos es el hiperparámetro C la cual tiene la función de determinar la posición del hiperplano, en otras palabras, el error. Si este parámetro es pequeño las observaciones no se vuelven tan significativas por lo que el margen sería mayor, caso contrario sucede si el valor de C es mayor. El otro hiperparámetro es el *gamma*, el cual tiene como función determinar cuanta va a ser la curvatura del límite de decisión. Esto quiere decir que, a mayor valor de este hiperparámetro, mayor será la curvatura y esta será menos con un *gamma* menor.

```
[35] #Muestra de resultados
     print("reporte de clasificación - \n{}:\n{}\n".format(
         clf, metrics.classification_report(y_test, y_pred)))
     print("Precisión: {}".format(metrics.accuracy_score(y_test,y_pred)))
```

```
reporte de clasificación -
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0):
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         5
           1       0.83      0.73      0.78        48
           2       0.83      0.96      0.89        72

    accuracy                           0.83       125
   macro avg       0.55      0.56      0.56       125
weighted avg       0.80      0.83      0.81       125


Precisión: 0.832
/usr/local/lib/python3.6/dist-packages/sklearn/metrics/_classification.py:1272: UndefinedMetricWarning: Precision and
  _warn_prf(average, modifier, msg_start, len(result))
```

Figura 61. Muestra de resultados.

Fuente: Elaboración propia.

2.4.3.1 Data de enfermedades. En los resultados mostrados a partir del reporte de clasificación, se puede ver distintas métricas en las cuales se aprecia que tanta precisión tiene este algoritmo al clasificar las distintas clases.

Una observación muy importante es el hecho que tanto para la precisión, *recall* y *f1-score* su valor es 0. Esto representa una alerta pues la clase cero no está siendo clasificada. Al abrir una celda temporal para poder visualizar de una mejor manera como estaban siendo asociadas, se observó que la clase cero representa a la enfermedad BBS la cual solo cuenta con 43 imágenes siendo esta carpeta mucho menor en número que las demás. El *f1- score* hace referencia a los falsos positivos, por lo que mientras más cercano a 1 este sea, la clasificación será mejor. Además, es una forma de determinar qué tan balanceado es el *data set*, lo cual no sucede en este caso.

En cuanto a la clase 1 y 2, se obtiene un f1-score de 78% y 89% respectivamente, lo cual es un buen indicativo, pero aún mejorable solucionando el problema del desbalanceo de data.

A continuación, se podrá observar en la matriz confusión y confirmar que la clase no está siendo clasificada ya que es en la diagonal donde los valores más altos deberían encontrarse y no en los alrededores.

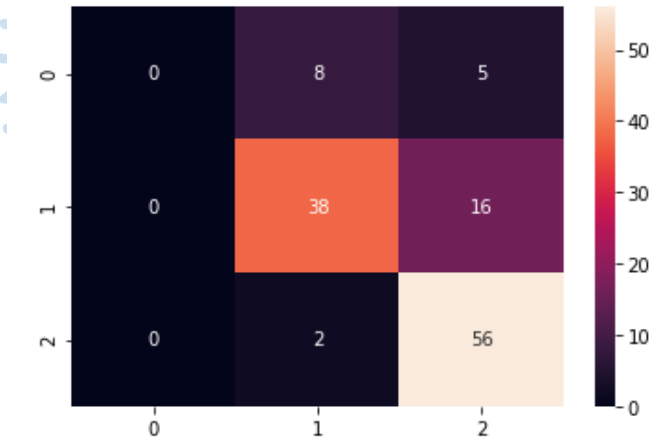

Figura 62. Matriz confusión con la data de enfermedades.

Fuente: Elaboración propia.

2.4.3.2 Data de nutrientes

```
#Muestra de resultados
print("reporte de clasificación - \n{}:\n{}\n".format(
    clf, metrics.classification_report(y_test, y_pred)))
print("Precisión: {}".format(metrics.accuracy_score(y_test,y_pred)))
```

```
reporte de clasificación -
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0):
              precision    recall  f1-score   support

           0       0.75      0.75      0.75         8
           1       0.92      0.98      0.95        50
           2       0.86      1.00      0.93        19
           3       0.96      0.76      0.85        29

    accuracy                           0.91       106
   macro avg       0.87      0.87      0.87       106
weighted avg       0.91      0.91      0.90       106


Precisión: 0.9056603773584906
```

Figura 63. Muestra de resultados.

Fuente: Elaboración propia.

Se observa que las clases 0, 1, 2, y 3 correspondientes a *healthy, nitrogen, phosphorus* y *potassium* respectivamente. Se han obtenido valores de *f1-score* de 75%, 95%, 93% y de 85%, respectivamente. Todos son valores muy cercanos al 1 por lo que se puede deducir que la data se encuentra bien balanceada y que se está clasificando de manera correcta. Además, al momento de evaluar la precisión se obtiene 75%, 92%, 86% y 96%, respectivamente y de manera global se obtiene un accuracy de aproximadamente 91%.

Los buenos resultados obtenidos se confirman con la matriz confusión donde los mayores valores se encuentran en la diagonal y los alrededores está siendo completados con 0 y 1. Esto confirma la correcta clasificación de las clases.

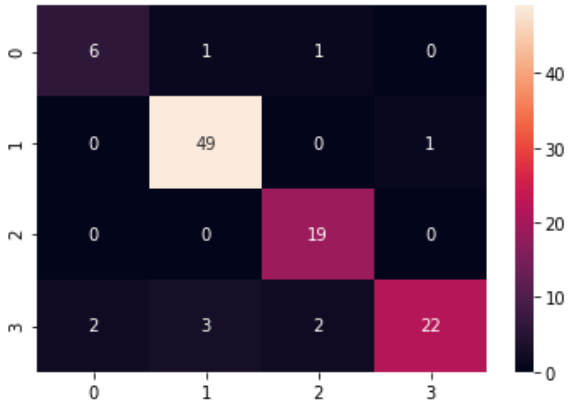

Figura 64. Matriz confusión con la data de nutrientes.

Fuente: Elaboración propia.

2.4.4 Filtros Convolucionales de arquitectura VGG16 con algoritmo Random Forest

Procedemos de la misma manera que en las pruebas anteriores: se importan las librerías mencionadas en capítulos previos. Además, se crea la variable *size* para definir el tamaño de las imágenes, junto con la lista sin elementos de imágenes y rótulos para clasificar.

```
import numpy as np
import matplotlib.pyplot as plt
import glob
import cv2
from keras.models import Model, Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from keras.layers.normalization import BatchNormalization
import os
import seaborn as sns
from keras.applications.vgg16 import VGG16
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
SIZE = 256
leaf_images = []
leaf_labels = []
```

Figura 65 - Importación de librerías para implementación de modelo *VGG16* con algoritmo *RandomForest*.

Fuente: Elaboración propia.

2.4.4.1 Data de nutrientes. Se recorren las carpetas y se procesan las imágenes para transformar la información en vectores por medio de la librería *cv2*. Terminado eso, se añade a la lista el elemento por cada iteración y lo mismo sucede con la etiqueta correspondiente a la imagen.

```
for directory_path in glob.glob("/content/drive/My Drive/data de nutrientes/*"):
    print(directory_path)
    label = directory_path.split("/")[-1]
    for img_path in glob.glob(os.path.join(directory_path, "*")):
        print(img_path)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (SIZE, SIZE))
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        leaf_images.append(img)
        leaf_labels.append(label)
```

Figura 66. Tratamiento de imágenes para modelo *VGG16* y algoritmo *RandomForest*, datos de nutrientes.

Fuente: Elaboración propia.

Después, por medio de la librería Numpy, se transforman las listas en *arrays* para trabajar de mejor forma con la red más adelante.

```
leaf_images=np.array(leaf_images)
leaf_labels=np.array(leaf_labels)
```

Figura 67. Transformación de listas generadas por el tratamiento de imágenes en arrays para ingresar a algoritmo.

Fuente: Elaboración propia.

Para transformar las etiquetas en valores numéricos se emplea el preprocesado y se construye un nuevo *array*.

```
le = preprocessing.LabelEncoder()
le.fit(leaf_labels)
leaf_labels_encoded = le.transform(leaf_labels)
```

Figura 68. Transformación de las etiquetas en valores numéricos.

Fuente: Elaboración propia.

Para segmentar el set de datos, se emplea la función de *train_test_split* de la librería *SciKit Learn*, con una división del 80% para entrenar el modelo y el 20% restante para probar la validez de la red.

```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test=train_test_split(leaf_images,leaf_labels_encoded,test_size=0.2)
```

Figura 69. División de la data para entrenamiento y prueba con nutrientes.

Fuente: Elaboración propia.

Luego, se divide el array entre 255 para fijar el valor de los pixeles entre 0 y 1.

```
x_train, x_test = x_train / 255.0, x_test / 255.0
```

Figura 70 . Tratamiento de arrays de prueba y entrenamiento.

Fuente: Elaboración propia.

Finalmente, por medio de la función *to_categorical* se indica la diferencia de la etiqueta de las otras clases.

Después, para trabajar con el modelo *VGG16*, se llama la función con el mismo nombre y se importan los pesos denominados bajo el nombre de *imagenet*. De esta manera se trabaja con un set de pesos pre entrenados con la librería *ImageNet*, que tiene más de 15 millones de imágenes en alta resolución clasificadas en 22 000 categorías, rotuladas manualmente .(Yang

et al., 2019) Luego se coloca “False” con el parámetro *include_top* para no incluir las 3 capas densas a la salida de la red trabajada.

```
from keras.utils import to_categorical
y_test_one_hot = to_categorical(y_test)
y_train_one_hot = to_categorical(y_train)
```

Figura 71. Tratamiento de los arrays de respuesta de prueba y entrenamiento.

Fuente: Elaboración propia.

```
#Aqui comenzamos a emplear el peso de las VGG
VGG_model = VGG16(weights='imagenet', include_top=False, input_shape=(SIZE, SIZE, 3))
```

Figura 72. Implementación del modelo de red convolucional *VGG16*, pesos importados de *ImageNet*.

Fuente: Elaboración propia.

Debido a la cantidad de parámetros que se suele emplear para entrenar al modelo *VGG16*, es necesario usar un número considerable de imágenes para extraer de manera correcta las características del set. Sin embargo, dada la limitada cantidad de imágenes, se recomienda cambiar los parámetros entrenables a no entrenables para emplear los pesos preestablecidos proporcionados por la red.

```
#La VGG del modelo suele emplear 14M de parámetros entrenables.
#Pero como necesitamos los parámetros intáctos para performear
#el transfer learning, negamos las capas entrenables.
for layer in VGG_model.layers:
  layer.trainable = False
```

Figura 73. Modificación de las capas con parámetros entrenables para arquitectura *VGG16*.

Fuente: Elaboración propia.

Luego, se extraen los datos que han pasado por el filtro convolucional y se ajustan al modelo *RandomForest* con las entradas *X_para_RF* y *y_train*.

```
feature_extractor=VGG_model.predict(x_train)
X_para_RF = feature_extractor.reshape(feature_extractor.shape[0], -1)
RF_model = RandomForestClassifier(n_estimators = 100, random_state = 42)
RF_model.fit(X_para_RF, y_train)
```

Figura 74. Procesamiento de los datos con filtros convolucionales de arquitectura VGG16.

Fuente: Elaboración propia.

Para conseguir las predicciones del modelo respecto al set de prueba, se aplican las líneas de código que aparecen en las figuras 75 y 76.

```
X_test_feature = VGG_model.predict(x_test)
X_test_features = X_test_feature.reshape(X_test_feature.shape[0], -1)
```

Figura 75. Obtención de predicciones con filtros convolucionales de arquitectura VGG 16, datos de nutrientes.

Fuente: Elaboración propia.

```
prediction_RF = RF_model.predict(X_test_features)
#prediction_RF = le.inverse_transform(prediction_RF)
```

Figura 76. Predicción del modelo con algoritmo RandomForest.

Fuente: Elaboración propia.

Para obtener la precisión, se usa la función *accuracy_score* de la librería *metrics* para medir la eficacia del modelo.

```
print ("Accuracy = ", metrics.accuracy_score(y_test, prediction_RF))
```

```
Accuracy =  0.9622641509433962
```

Figura 77. Línea de código para obtener la precisión del modelo y resultado con data de nutrientes.

Fuente: Elaboración propia.

Finalmente, para obtener la matriz confusión se detallan las siguientes líneas de código.

```
from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, prediction_RF)
#print(cm)
sns.heatmap(cm, annot=True)
```

Figura 78. Implementación de matriz confusión para modelo con arquitectura *VGG16* y algoritmo *Random Forest*, datos de nutrientes.

Fuente: Elaboración propia.

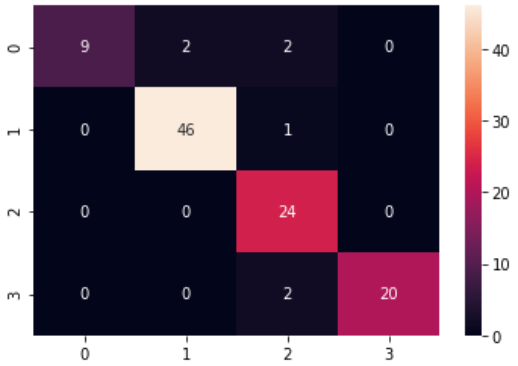

Figura 79. Matriz confusión para modelo con arquitectura VGG16 y algoritmo *RandomForest*, datos de nutrientes.

Fuente: Elaboración propia.

Para probar la eficacia de la red, se testea de manera aleatoria los elementos del array a partir de un entero seleccionado con el módulo *random* de *numpy* y se indica cuáles son las etiquetas reales y las proporcionadas por el modelo.

```
n=np.random.randint(0, x_test.shape[0])
img = x_test[n]
plt.imshow(img)
input_img = np.expand_dims(img, axis=0)
input_img_feature=VGG_model.predict(input_img)
input_img_features=input_img_feature.reshape(input_img_feature.shape[0], -1)
prediction_RF = RF_model.predict(input_img_features)[0]
prediction_RF = le.inverse_transform([prediction_RF])
a=le.inverse_transform(y_test)
print("La predicción para esta imagen es: ", prediction_RF)
print("La etiqueta real es: ", a[n])
```

Figura 80. Obtención de la predicción para un dato aleatorio en el set de prueba de nutrientes.

Fuente: Elaboración propia.

Figura 81. Resultados de predicción de modelo con filtros convolucionales y algoritmo *Random Forest*, datos de nutrientes.

Fuente: Elaboración propia.

2.4.4.2 Data de enfermedades. De la misma forma se tratará a la data de enfermedades. Siendo así, parte del código que se tendrá que cambiar es referido a la ubicación de las imágenes.

```
for directory_path in glob.glob("/content/drive/My Drive/data enfermedades/*"):
    print(directory_path)
    label = directory_path.split("/")[-1]
    for img_path in glob.glob(os.path.join(directory_path, "*")):
        print(img_path)
        img = cv2.imread(img_path, cv2.IMREAD_COLOR)
        img = cv2.resize(img, (SIZE, SIZE))
        img = cv2.cvtColor(img, cv2.COLOR_RGB2BGR)
        leaf_images.append(img)
        leaf_labels.append(label)
```

Figura 82. Tratamiento de imágenes para modelo con filtros convolucionales de arquitectura *VGG16* y algoritmo *Random Forest*.

Fuente: Elaboración propia.

```
print ("Precision = ", metrics.accuracy_score(y_test, prediction_RF))
```

```
Precision =  0.888
```

Figura 83. Línea de código para obtener la precisión del modelo y resultado con data de enfermedades.

Fuente: Elaboración propia.

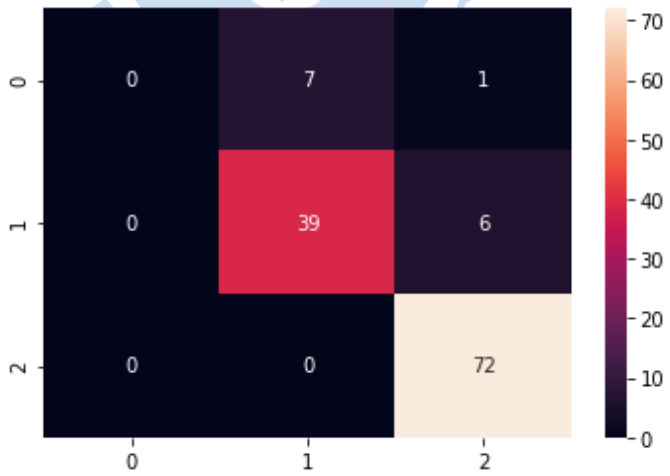

Figura 84 . Matriz confusión para modelo con filtros convolucionales de arquitectura VGG16 y algoritmo RandomForest, datos de enfermedades.

Fuente: Elaboración propia.

Figura 85. Resultados de predicción de modelo con filtros convolucionales y algoritmo RandomForest, datos de enfermedades.

Fuente: Elaboración propia.

2.4.5 Filtros Convolucionales de arquitectura VGG16 con Algoritmo KNN

Para preparar las entradas que este algoritmo de clasificación se utilizarán los filtros convolucionales de VGG16, utilizando su estructura previamente usada en los apartados pasados.

2.4.5.1 Data de enfermedades. Se utilizará la estructura de los filtros convolucionales *VGG16* para preparar las entradas del KNN.

```
#Aqui comenzamos a emplear el peso de las VGG
VGG_model = VGG16(weights='imagenet', include_top=False, input_shape=(SIZE, SIZE, 3))
#La VGG del modelo suele emplear 14M de parámetros entrenables.
#Pero como necesitamos los parámetros intáctos para performear
#el transfer learning, negamos las capas entrenables.
for layer in VGG_model.layers:
  layer.trainable = False
```

Figura 86. Código filtros convolucionales de *VGG16*.

Fuente: Elaboración propia.

Después se importarán las funciones de las librerías, funciones que utilizará el clasificador KNN.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
from sklearn.metrics import accuracy_score
```

Figura 87. Importación de funciones.

Fuente: Elaboración propia.

Debido a que se está trabajando con la data de enfermedades, el número de clase es 3 por lo que p=3 y para el número de vecinos K se toma como referencia el valor usado con KNN y la arquitectura de 11 capas, para comprobar mediante un rango de K, el vecino con mejor precisión. En este caso el K con mejor precisión del rango de 15 a 25 es 15 y se elegirá este para el modelo.

```
feature_extractor=VGG_model.predict(X_train)
X_para_KNN=feature_extractor.reshape(feature_extractor.shape[0],-1)
X_de_prueba=VGG_model.predict(X_test)
X_de_prueba_reshaped=X_de_prueba.reshape(X_de_prueba.shape[0],-1)
```

```
KNN_model=KNeighborsClassifier(n_neighbors=15, p=3, metric='euclidean')
KNN_model.fit(X_para_KNN,y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=15, p=3,
                     weights='uniform')
```

Figura 88. Código de preparación de modelo KNN con *VGG16* para data de enfermedades.

Fuente: Elaboración propia.

```
k_range = range(15, 23)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba_reshaped, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25])
```

```
([<matplotlib.axis.XTick at 0x7fbf95e5dc50>,
  <matplotlib.axis.XTick at 0x7fbf95e5dc18>,
  <matplotlib.axis.XTick at 0x7fbf95e5d860>,
  <matplotlib.axis.XTick at 0x7fbf95e88518>],
 <a list of 4 Text major ticklabel objects>)
```

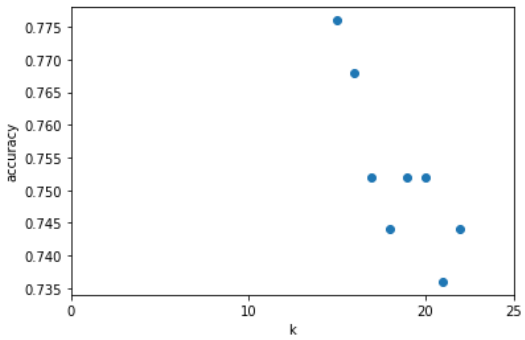

Figura 89. Código y gráfica resultante de K vs Precisión para la data de enfermedades.

Fuente: Elaboración propia.

Se evalúa la precisión del clasificador con K=15 del modelo de prueba, el cual resulta con una precisión del 77%.

```
acc=KNN_model.score(X_de_prueba_reshaped,y_test)
print(acc)
```

```
0.776
```

Figura 90. Escala del modelo KNN para data de enfermedades.

Fuente: Elaboración propia

Para confirmar la precisión del modelo, se verá la matriz confusión sobre el conjunto de prueba *(X_prueba).*

Con lo resultante de la matriz confusión en la figura 91, se observa que de nuestro *set* de prueba solo una imagen pudo clasificarse correctamente como BBS, en el caso de BBW 38 imágenes se clasificaron correctamente y para la clasificación de *healthy* se clasificaron 58 imágenes.

Observando el reporte, la puntuación *f1-score*, de falsos positivos, es mala para la clasificación de la clase BBS con 18%. En los casos de las clases BBW y *healthy* tienen un porcentaje bueno de 72% y 82%.

```
#Predecir la respuesta para el conjunto de datos de prueba
y_pred = KNN_model.predict(X_de_prueba_reshaped)
#Evaluate model
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm,annot=True)
print(classification_report(y_test,y_pred))
```

```
[[ 1  3  6]
 [ 0 38 17]
 [ 0  2 58]]
              precision    recall  f1-score   support

           0       1.00      0.10      0.18        10
           1       0.88      0.69      0.78        55
           2       0.72      0.97      0.82        60

    accuracy                           0.78       125
   macro avg       0.87      0.59      0.59       125
weighted avg       0.81      0.78      0.75       125
```

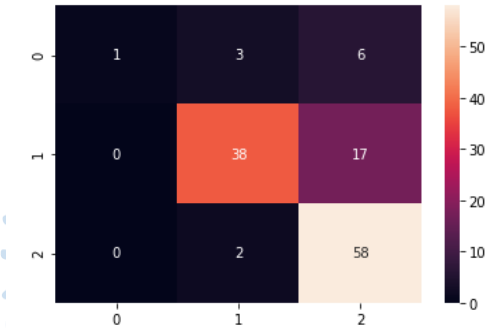

Figura 91. Matriz confusión para data de enfermedades- KNN con VGG16.

Fuente: Elaboración propia.

2.4.5.2 Data de deficiencia de nutrientes. Se realiza el mismo procedimiento del código en la data de enfermedades, pero se realiza cambios en los valores de las variables dentro del clasificador KNN.

```
feature_extractor=VGG_model.predict(X_train)
X_para_KNN=feature_extractor.reshape(feature_extractor.shape[0],-1)
X_de_prueba=VGG_model.predict(X_test)
X_de_prueba_reshaped=X_de_prueba.reshape(X_de_prueba.shape[0],-1)
```

Figura 92. Código de preparación de modelo KNN con *VGG16* para data de deficiencia de nutrientes.

Fuente: Elaboración propia.

Para esta data el número de clases es de 4 y el valor de *n_neighbors=K* será 21, ya que es el valor con mejor precisión en nuestro clasificador; esto último se puede observar en la figura 93 donde el rango de K va desde 21 a 31, siendo 21 tiene la mejor precisión.

```
k_range = range(21, 31)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba_reshaped, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25,30,35])
```

```
([<matplotlib.axis.XTick at 0x7fbb3b850668>,
  <matplotlib.axis.XTick at 0x7fbb3b850630>,
  <matplotlib.axis.XTick at 0x7fbb3b850278>,
  <matplotlib.axis.XTick at 0x7fbb3b7f0eb8>,
  <matplotlib.axis.XTick at 0x7fbb3b802390>,
  <matplotlib.axis.XTick at 0x7fbb3b802828>],
 <a list of 6 Text major ticklabel objects>)
```

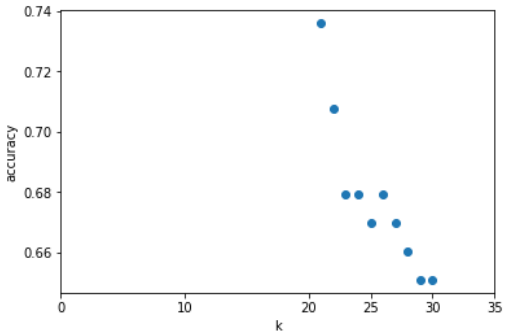

Figura 93. Código y Gráfica resultante de K vs precisión para la data de deficiencia de nutrientes.

Fuente: Elaboración propia.

El código de la figura siguiente es usado para escalar el modelo:

```
KNN_model=KNeighborsClassifier(n_neighbors=21, p=4, metric='euclidean')
KNN_model.fit(X_para_KNN,y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclidean',
                     metric_params=None, n_jobs=None, n_neighbors=21, p=4,
                     weights='uniform')
```

```
#Predecir la respuesta para el conjunto de datos de prueba
y_pred = KNN_model.predict(X_de_prueba_reshaped)
```

Figura 94. Escala del modelo KNN para data de deficiencia de nutrientes.

Fuente: Elaboración propia.

Se evalúa la precisión del clasificador con K=21 del modelo de prueba, el cual resulta con una precisión del 73%.

```
acc=KNN_model.score(X_de_prueba_reshaped,y_test)
print(acc)
```

```
0.7358490566037735
```

Figura 95. Escala del modelo KNN para data de deficiencia de nutrientes.

Fuente: Elaboración propia.

Para observar en detalle cómo trabaja nuestro modelo, se usa la matriz confusión sobre el conjunto de prueba *(X_prueba).*

```
#Predecir la respuesta para el conjunto de datos de prueba
y_pred = KNN_model.predict(X_de_prueba_reshaped)
#Evaluación del modelo
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm,annot=True)
print(classification_report(y_test,y_pred))
```

```
[[ 2  5  0  0]
 [ 0 54  0  0]
 [ 1  9 20  0]
 [ 1 12  0  2]]
              precision    recall  f1-score   support

           0       0.50      0.29      0.36         7
           1       0.68      1.00      0.81        54
           2       1.00      0.67      0.80        30
           3       1.00      0.13      0.24        15

    accuracy                           0.74       106
   macro avg       0.79      0.52      0.55       106
weighted avg       0.80      0.74      0.69       106
```

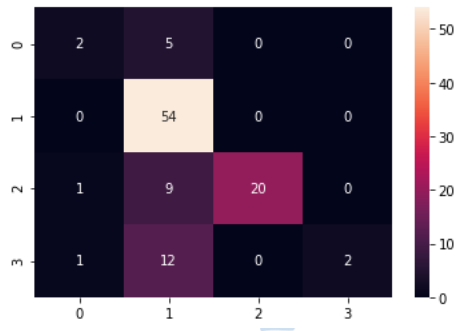

Figura 96. Matriz confusión para data de deficiencia de nutrientes- KNN.

Fuente: Elaboración propia

Lo que se observa de la matriz confusión es que correctamente se clasifican solo 2 imágenes para la clase *healthy*, en la clase *nitrogen* se clasifican 54 imágenes correctamente, en la clase *phosphorus* 20 imágenes clasificadas correctamente y en la clase *potassium* solo 2 fueron clasificadas correctamente.

Observando el reporte, la puntuación *f1-score*, de falsos positivos, es mala para las clases *healthy* y *potassium* con 36% y 24% respectivamente, esto se ve también reflejado en la matriz confusión por la cual se sabe que solo pocas imágenes se clasificaron correctamente

de estas clases. En los casos de las clases *nitrogen* y *phosphorus* tienen una puntuación buena de 81% y 80%.

2.4.6 Filtros Convolucionales de arquitectura VGG16 con Algoritmo de clasificación SVC

2.4.6.1 Data de enfermedades

```
reporte de clasificación -
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0):
              precision    recall  f1-score   support

           0       0.60      0.25      0.35        12
           1       0.78      0.90      0.84        42
           2       0.97      0.97      0.97        71

    accuracy                           0.88       125
   macro avg       0.78      0.71      0.72       125
weighted avg       0.87      0.88      0.87       125


Precisión: 0.88
```

Figura 97. Precisión y *f1-score* con data de nutrientes.

Fuente: Elaboración propia

Se aprecia en el reporte de clasificación la precisión y el *f1-score* para cada una de las clases. Se observa que todas clases están siendo clasificadas, sin embargo, la clase 0 que representa BBS sigue siendo muy bajo. Esto demuestra que el uso de estos filtros convolucionales mejora los resultados, pero sigue siendo presente un desbalanceo de la *data*.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
#print(cm)
sns.heatmap(cm, annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbcfbaa24e0>
```

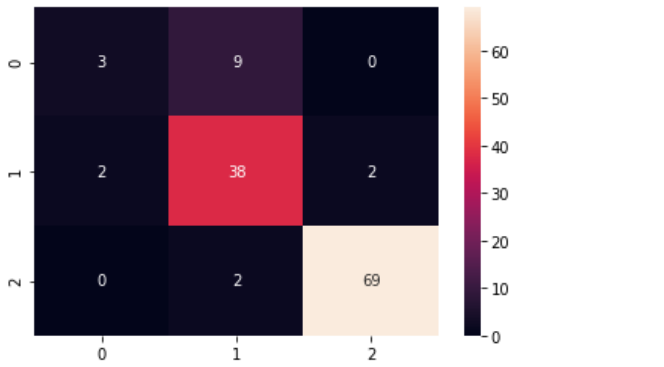

Figura 98. Matriz confusión con data de enfermedades.

Fuente: Elaboración propia.

En la matriz confusión de la figura 98, se puede apreciar cómo están siendo clasificadas las clases de una manera visual. La diagonal debe contiene los valores más altos y se confirma que la clase 0 sigue sin estar correctamente clasificada ya que el mayor número se encuentra fuera de la diagonal.

2.4.6.2 Data de nutrientes

```
[15] #Muestra de resultados
     print("reporte de clasificación - \n{}:\n{}\n".format(
         clf, metrics.classification_report(y_test, y_pred)))
     print("Precisión: {}".format(metrics.accuracy_score(y_test,y_pred)))
```

```
reporte de clasificación -
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0):
              precision    recall  f1-score   support

           0       1.00      0.42      0.59        12
           1       0.87      0.98      0.92        41
           2       0.97      1.00      0.99        34
           3       0.90      0.95      0.92        19

    accuracy                           0.92       106
   macro avg       0.94      0.83      0.85       106
weighted avg       0.92      0.92      0.90       106


Precisión: 0.9150943396226415
```

Figura 99. Precisión y *f1-score* con data de nutrientes.

Fuente: Elaboración propia.

Para el caso de la data de nutrientes se ha obtenido una precisión de aproximadamente a 92%, cuya precisión por clase varían entre el 87% y 97%. Mientras que el *f1-score* presenta que de todas las clases la que tiene más problemas para ser identificada es la clase 0 perteneciente a *healthy*, esto se justifica porque de todas las clases es la que tiene menos imágenes.

Se observa que con la matriz confusión en la figura 100 de la data de nutrientes se han obtenido los mayores valores dentro de la diagonal, lo cual demuestra que las clases están siendo bien identificadas. Sin embargo, aún se puede notar la dificultad para clasificar la clase 0, esto se puede deber a la cantidad de imágenes como se mencionó antes o a la similitud en apariencia que esta clase tiene con la de *nitrogen*.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
#print(cm)
sns.heatmap(cm, annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbd120df470>
```

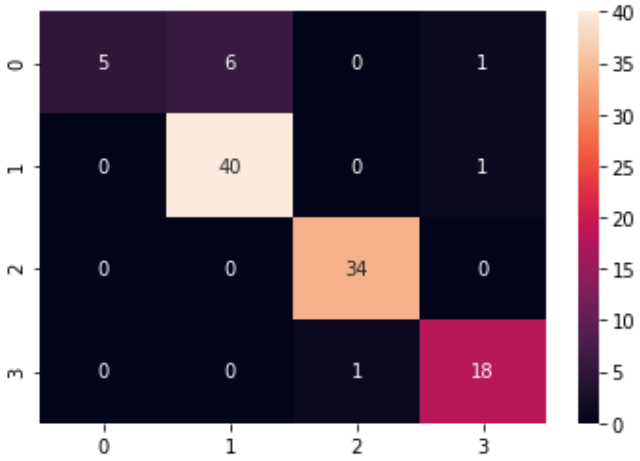

Figura 100. Matriz confusión con data de nutrientes.

Fuente: Elaboración propia.

2.4.7 Rebalanceo de Data de enfermedades

Como consignan las matrices confusión de los resultados anteriores respecto a la data de enfermedades, la clase BBS no se correspondía respecto a la clasificación que el algoritmo hacía. Para poder evaluar la eficacia de los algoritmos, se procedió a balancear el *data set* de enfermedades, igualando la cantidad de imágenes a 43 por cada clase.

2.4.7.1 Filtros convolucionales de una arquitectura de 11 capas y algoritmo de clasificación *Random Forest*. La figura 101, muestra 3 matrices confusión que corresponden a cada uno de los 3 intentos que fueron desarrollados para evaluar la precisión del código referentes a las 3 clases de enfermedades: BBS, BBW y *healthy*. Como resultado de ello, se obtuvieron 3 valores de precisión distintos para cada intento, así pues, la matriz (a) nos muestra un valor de precisión del 57.7%, mientras que con las matrices (b) y (c) se obtuvieron valores 61.5% y 69.2% respectivamente, lo cual denota una gran variabilidad en su precisión producto de la poca robustez del algoritmo. Este hecho se puede justificar en la escasa cantidad de datos con el que se alimenta el sistema, ya que el directorio con el que se está trabajando en esta ocasión solo cuenta con 43 imágenes por clase.

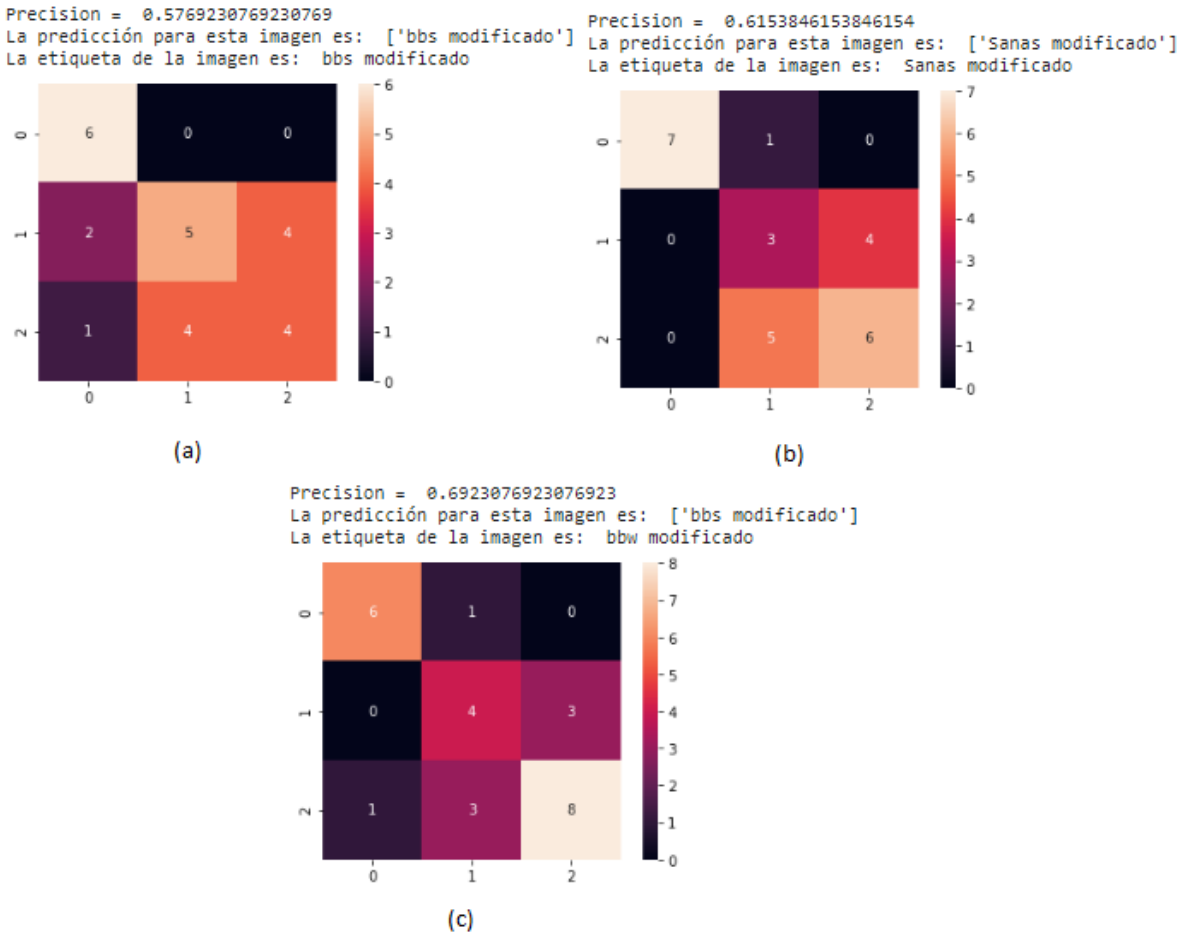

Figura 101. Matrices confusión - Data de Enfermedades rebalanceada.

Fuente: Elaboración propia.

Para el primer intento (a), observando la diagonal mayor permite entender que 6 imágenes del set pudieron clasificarse correctamente como BBS (100 % del total), 5 de ellas se clasificaron correctamente como BBW (45 % aproximadamente) y 4 de ellas clasificadas correctamente como especímenes sanos (representando el 44.4 %). Por otro lado, la predicción de la etiqueta fue la correcta pese a la baja precisión (57.7%), aspecto del cual no se puede fiar totalmente.

Para el segundo intento (b), la diagonal mayor muestra que 7 de 8 imágenes del set pudieron clasificarse correctamente como BBS (87.5 % del total), 3 se clasificaron correctamente como BBW (43 % aproximadamente), mientras que 6 fueron clasificadas correctamente como especímenes sanos (representando el 54.5 %). Por otro lado, al igual que en el anterior intento, la predicción de la etiqueta fue la correcta pese a la baja precisión (61.5%).

En el tercer intento (c), 6 de 7 imágenes del set pudieron clasificarse correctamente como BBS (85.7 % del total), 4 se clasificaron correctamente como BBW (57.14 % aproximadamente), mientras que 8 fueron clasificadas correctamente como especímenes

sanos (representando el 66.67 %). Finalmente, a diferencia de los demás intentos, la predicción de la etiqueta fue incorrecta pese a poseer la precisión más elevada de los 3 intentos (69.23%), lo cual muestra que no se debe fiar de un valor de precisión bajo.

2.4.7.2 Filtros convolucionales de arquitectura VGG16 y algoritmo de clasificación *Random Forest*.

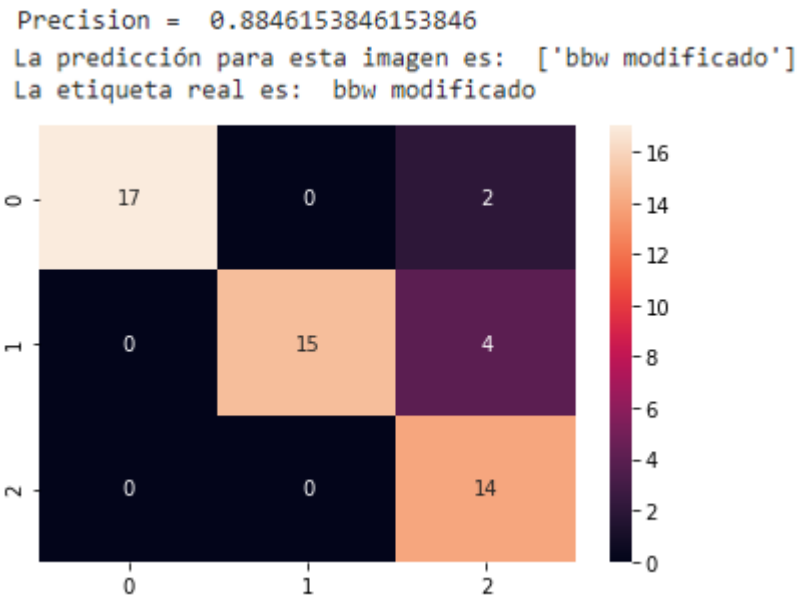

Figura 102. Matriz confusión con data de Enfermedades rebalanceada.

Fuente: Elaboración propia.

Para la Figura 102, se obtuvo una precisión del 88.46%. Dicho valor de precisión para el modelo es mayor al 85%, con lo cual, se puede decir que es un modelo robusto presentando, además, un margen de mejora amplio. La clasificación realizada entre el modelo y la etiqueta real se corresponden de manera correcta, en donde el programa fue capaz de reconocer la enfermedad BBW sin problemas. Respecto a la matriz confusión, las diagonales mayores dan a entender que el modelo cumple al momento de asociar las etiquetas con las predicciones, asimismo se observa que 17 imágenes del set pudieron clasificarse correctamente como BBS (90 % aproximadamente del total), 15 de ellas se clasificaron correctamente como BBW (79 % aproximadamente) y 14 de ellas clasificadas correctamente como especímenes sanos (representando el 100 %).

2.4.7.3 Filtros convolucionales de una arquitectura de 11 capas y algoritmo de clasificación *KNN*. Para este caso el rango que se utilizará para evaluar el número de vecinos K adecuado, se sacará un nuevo rango que esté dentro de 10 que es la raíz cuadrada del número de elementos de la entrada.

```
import math
math.sqrt(len(X_para_KNN))
```

```
10.14889156509222
```

Figura 103. Resultado raíz cuadrada del número de elementos de la entrada.

Fuente: Elaboración propia.

El rango elegido es de 5 a 15, con estos valores de K se procederá a calcular la precisión del modelo para cada punto y elegir el valor con mejor precisión para nuestro modelo.

```
k_range = range(5, 15)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25])
```

```
([<matplotlib.axis.XTick at 0x7fd1fe959780>,
  <matplotlib.axis.XTick at 0x7fd1fe959748>,
  <matplotlib.axis.XTick at 0x7fd1fe959390>,
  <matplotlib.axis.XTick at 0x7fd1fe8ff6a0>],
 <a list of 4 Text major ticklabel objects>)
```

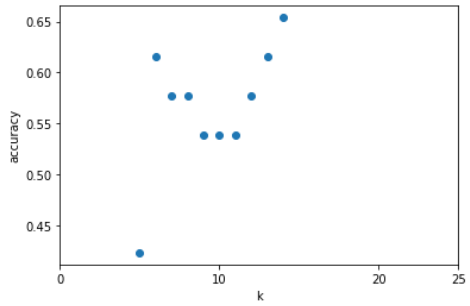

Figura 104. Código y Gráfica resultante de K vs precisión para la data de enfermedades modificada.

Fuente: Elaboración propia.

En este caso el K=15 es el valor óptimo, usando este valor para nuestro modelo se obtiene una precisión de 65%.

```
acc=KNN_model.score(X_de_prueba,y_test)
print(acc)
```

```
0.6538461538461539
```

Figura 105. Precisión del algoritmo para data de deficiencia de nutrientes.

Fuente: Elaboración propia.

Para observar con mayor detalle cómo funciona el modelo se ejecuta la matriz confusión a continuación:

```
#Matriz confusión
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm, annot=True)
print(classification_report(y_test,y_pred))
```

```
[[9 0 0]
 [1 6 0]
 [6 2 2]]
              precision    recall  f1-score   support

           0       0.56      1.00      0.72         9
           1       0.75      0.86      0.80         7
           2       1.00      0.20      0.33        10

    accuracy                           0.65        26
   macro avg       0.77      0.69      0.62        26
weighted avg       0.78      0.65      0.59        26
```

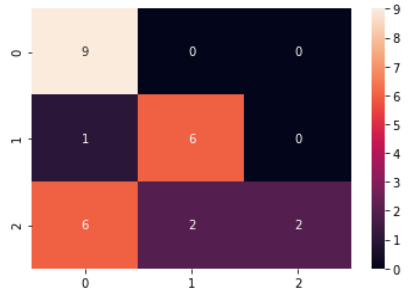

Figura 106. Matriz confusión para data de enfermedades modificada-KNN.

Fuente: Elaboración propia.

Se puede observar en la matriz confusión el modelo pudo clasificar correctamente 9 imágenes de la clase BBS, 6 imágenes de la clase BBW y 2 imágenes de la clase *healthy*. Observando el *f1-score*, la clase de *healthy* se clasifica mal con un porcentaje 33%, la medida de *f1-score*.

2.4.7.4 Filtros Convolucionales de arquitectura *VGG16* con Algoritmo *KNN.* El código es el mismo utilizado anteriormente con la primera data de enfermedades, lo que varía es el número de vecinos K. Para la elección de este, se recomienda utilizar a partir de la raíz cuadrada del número de elementos de la entrada y a partir de ese número elegir su impar siguiente.

```
import math
math.sqrt(len(X_para_KNN))
```

```
10.14889156509222
```

Figura 107. La raíz cuadrada del número de elementos de la entrada.

Fuente: Elaboración propia.

Para comprobar que el K=11 es el que mejor precisión tiene, elegiremos un rango dentro del K elegido previamente, este se encuentra entre 8 y 17. Se concluye que K=12 es el que mejor precisión tiene por lo que será el usado.

```
k_range = range(8, 17)
scores = []
for k in k_range:
    knn = KNeighborsClassifier(n_neighbors = k)
    knn.fit(X_para_KNN, y_train)
    scores.append(knn.score(X_de_prueba_reshaped, y_test))
plt.figure()
plt.xlabel('k')
plt.ylabel('accuracy')
plt.scatter(k_range, scores)
plt.xticks([0,10,20,25])
```

```
([<matplotlib.axis.XTick at 0x7f8e3b49c518>,
  <matplotlib.axis.XTick at 0x7f8e3971e8d0>,
  <matplotlib.axis.XTick at 0x7f8e3971e518>,
  <matplotlib.axis.XTick at 0x7f8e396f3198>],
 <a list of 4 Text major ticklabel objects>)
```

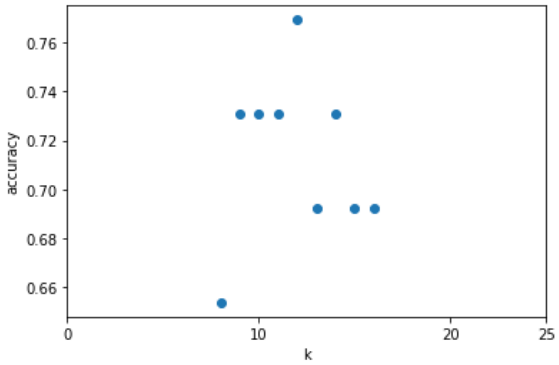

Figura 108. Código y Gráfica resultante de K vs precisión para la data de enfermedades modificada.

Fuente: Elaboración propia.

La precisión obtenida para el K establecido anteriormente es de 77%.

```
acc=KNN_model.score(X_de_prueba_reshaped,y_test)
print(acc)
```

```
0.7692307692307693
```

Figura 109. Precisión del algoritmo para data de deficiencia de nutrientes 2.

Fuente: Elaboración propia**.**

Para mayor detalle sobre la clasificación de las imágenes se evalua la matriz confusión:

```
#Evaluate model
cm= confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm,annot=True)
print(classification_report(y_test,y_pred))
```

```
[[5 1 1]
 [0 9 2]
 [1 1 6]]
              precision    recall  f1-score   support

           0       0.83      0.71      0.77         7
           1       0.82      0.82      0.82        11
           2       0.67      0.75      0.71         8

    accuracy                           0.77        26
   macro avg       0.77      0.76      0.76        26
weighted avg       0.78      0.77      0.77        26
```

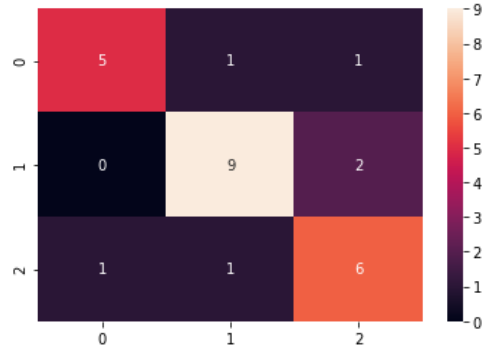

Figura 110 . Matriz confusión para data de enfermedades modificada KNN.

Fuente: Elaboración propia.

Se observa de la matriz confusión de la figura 110, clasifica correctamente 5, 9 y 6 imágenes de las clases BBS, BBW y *healthy* respectivamente. Observando la puntuación *f1-score* que considera tanto la precisión como la memoria para calcular, el que mayor porcentaje tiene y por lo tanto clasifica mejor es la clase de BBW con una precisión de 82%. El porcentaje obtenido de las demás clases también es considerado bueno, con una precisión de 77% para la clase BBS y 71% para la clase *healthy*.

2.4.7.5 Filtros convolucionales de una arquitectura de 11 capas y algoritmo de clasificación SVC. Se observa que para *healthy*, BBS y BBW se ha obtenido una precisión de 100%, 80% y 71% respectivamente. Los resultados son muchos mejores ya que todas las clases son leídas correctamente. Se observa un *f1-score* de 100%, 73% y 77% respectivamente lo cual es también un buen indicador que no hay mucha presencia de falsos positivos. Además, se presenta un buen indicador de *recall* con el cual se puede determinar que la exhaustividad con la que puede clasificar el modelo es alta.

```
#Muestra de resultados
print("reporte de clasificación - \n{}:\n{}\n".format(
    clf, metrics.classification_report(y_test, y_pred)))
print("Precisión: {}".format(metrics.accuracy_score(y_test,y_pred)))
```

```
reporte de clasificación -
GridSearchCV(cv=None, error_score=nan,
             estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                           class_weight=None, coef0=0.0,
                           decision_function_shape='ovr', degree=3,
                           gamma='scale', kernel='rbf', max_iter=-1,
                           probability=False, random_state=None, shrinking=True,
                           tol=0.001, verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                         {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                          'kernel': ['rbf']}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring=None, verbose=0):
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        14
           1       0.80      0.67      0.73         6
           2       0.71      0.83      0.77         6

    accuracy                           0.88        26
   macro avg       0.84      0.83      0.83        26
weighted avg       0.89      0.88      0.88        26


Precisión: 0.8846153846153846
```

Figura 111. Precisión y f1-score de data de Enfermedades rebalanceada.

Fuente: Elaboración propia.

```
cm = confusion_matrix(y_test, y_pred)
#print(cm)
sns.heatmap(cm, annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fe8e0346208>
```

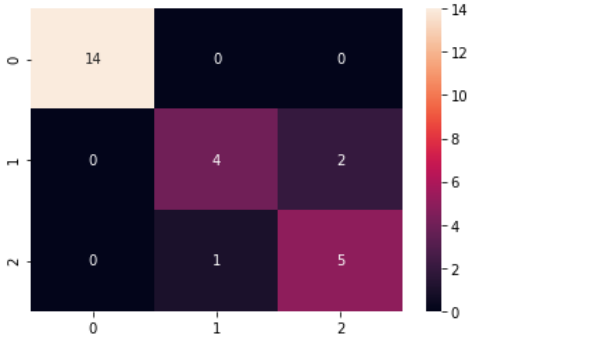

Figura 112. Matriz confusión de data de enfermedades rebalanceada.

Fuente: Elaboración propia.

La buena clasificación es corroborada con la matriz confusión ya que la diagonal tiene los valores más altos que hacen referencia a los verdaderos positivos. Alrededor de la diagonal se encuentran números cercanos al cero que hacen referencia a los verdaderos negativos.

2.4.7.6 Filtros convolucionales de arquitectura *VGG16* con algoritmo SVC. Al cambiar los filtros convolucionales por los de la arquitectura *VGG16*, se observa mejores resultados. La clase 0 que antes no era clasificada, es ahora identificada con una precisión del 100%, un *f1-score* de 80%. Esto indica que la data está correctamente balanceada y que la presencia de falsos positivos es muy baja.

```
[38] #Muestra de resultados
     print("reporte de clasificación - \n{}:\n{}\n".format(
         clf, metrics.classification_report(y_test, y_pred)))
     print("Precisión: {}".format(metrics.accuracy_score(y_test,y_pred)))

     reporte de clasificación -
     GridSearchCV(cv=None, error_score=nan,
                  estimator=SVC(C=1.0, break_ties=False, cache_size=200,
                                class_weight=None, coef0=0.0,
                                decision_function_shape='ovr', degree=3,
                                gamma='scale', kernel='rbf', max_iter=-1,
                                probability=False, random_state=None, shrinking=True,
                                tol=0.001, verbose=False),
                  iid='deprecated', n_jobs=None,
                  param_grid=[{'C': [1, 10, 100, 1000], 'kernel': ['linear']},
                              {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001],
                               'kernel': ['rbf']}],
                  pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
                  scoring=None, verbose=0):
                   precision    recall  f1-score   support

                0       0.92      1.00      0.96        12
                1       1.00      0.67      0.80         6
                2       0.89      1.00      0.94         8

         accuracy                           0.92        26
        macro avg       0.94      0.89      0.90        26
     weighted avg       0.93      0.92      0.92        26


     Precisión: 0.9230769230769231
```

Figura 113. Precisión y f1-score de data de enfermedades rebalanceada.

Fuente: Elaboración propia.

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
#print(cm)
sns.heatmap(cm, annot=True)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f81167d5f98>
```

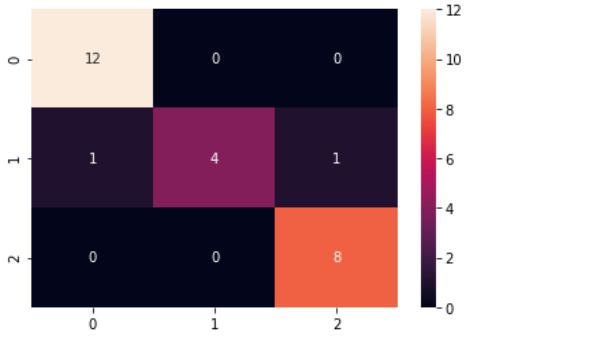

Figura 114 . Matriz confusión de data de Enfermedades modificada.

Fuente: Elaboración propia.

La matriz confusión confirma lo que las otras métricas dieron como resultado. La diagonal muestra valores mayores en comparación a los alrededores. Esto demuestra que la clasificación llevada a cabo es la correcta y que la clase BBS al cambiar los filtros convolucionales y balancear la data correctamente ha conseguido ser identificada en mayor porcentaje que cuando se probó con la data original.

.

.

UNIVERSITAS STUDIORUM PIURENSIS

Capítulo 3
Discusión de resultados

Se han utilizado una Red Neuronal Convolucional y tres tipos de algoritmos de clasificación: *Random Forest*, (KNN) y (SVC). Cada uno de estos usa filtros convolucionales de una arquitectura de 11 capas y de otra arquitectura *VGG16*.

Los resultados de la red neuronal de 11 capas con 25 y 50 épocas para la data de enfermedades, se obtuvo un 87.349% y 87.55% de precisión respectivamente. Se puede apreciar la ligera mejora de este indicador al aumentar el número de épocas. Con la data de deficiencia de nutrientes se obtuvo 87.028% para 25 épocas y 91.038% para 50 épocas. En este caso al aumentar el número de épocas al doble se aprecia una mejora en la precisión de alrededor de 4%.

Al evaluar el algoritmo *Random Forest* con los filtros convolucionales de la arquitectura de 11 capas se encontró para la data de enfermedades con una precisión de 89.6%. Para la data de deficiencia nutrientes se halló una precisión de 90.57%. Luego se simuló este algoritmo con los filtros convolucionales de una arquitectura *VGG16*, se halló una precisión de 88.88% y 96.23% para la data de enfermedades y deficiencia de nutrientes respectivamente.

El algoritmo KNN con filtros convoluciones de una arquitectura de 11 capas dio una precisión para la data de enfermedades de 78.4% y para la data de deficiencia de nutrientes 75.47%. Con los filtros convolucionales de *VGG16* se obtiene una precisión de 77.65% para la data de enfermedades y 73.58% para la de deficiencia de macronutrientes.

Por último, con el algoritmo SVC con filtros convolucionales de una arquitectura de 11 capas se obtuvo una precisión de 83% para la data de enfermedades y para la data de deficiencia de macronutrientes se encontró una precisión de 91%. Al probarlo con los filtros de la *VGG16* se encontró una precisión de 88% para la data de enfermedades y 91.51% para la de deficiencia de macronutrientes.

Se puede apreciar que los porcentajes de precisión para la data de enfermedades con la red neuronal convolucional y los tres algoritmos de clasificación es alta, sin embargo, al revisar el indicador *f1-score* se descubrió que el valor de este indicador para la clase BBS era de 0. Esto indica que esa clase no está siendo clasificada y que por ende hay un problema en

el balanceo de la data. Se procedió a reducir el número de imágenes de las dos otras clases (BBW y *healthy*) de manera que todas tuvieran la misma cantidad de imágenes.

Después de realizar esta modificación, se volvió a simular para los tres algoritmos de clasificación para este *data set*. Para *Random Forest* con filtros convoluciones de 11 capas se obtuvo 62% de precisión y 88.46% con los filtros convolucionales de la *VGG16.* Para KNN con los filtros convoluciones de 11 capas y *VGG16* se encontró 65% y 76.92% de precisión respectivamente. Por último, para el SVC y filtros convolucionales de 11 capas se encontró una precisión de 88.46% y 92.3% con los filtros convolucionales de *VGG16*. Estos resultados son óptimos ya que el *f1-score* de cada una de las clases para los 3 algoritmos sale mayor al 70%.

El algoritmo que mejor se comporta para los dos tipos de data y que mejor resultados ha dado tanto en la precisión, *f1-score* y en la matriz confusión es *Random Forest*, le sigue SVC, luego la CNN y por último KNN. La diagonal de la matriz confusión para el algoritmo *Random Forest* es la que ha obtenido los valores más altos, con lo cual se comprueba que la clasificación se ha llevado correctamente para la data de enfermedades y la data de deficiencia de macronutrientes.

Si se quiere ser más específico, la data de nutrientes ha dado mejores resultados con el algoritmo *Random Forest* mientras que la data de enfermedades ha obtenido mejores resultados con SVC, debido a que *Random Forest* no da buenos resultados cuando el *data set* es muy reducido.

Además, un factor determinante por la cual inclinarse por un algoritmo de clasificación, es en específico, uno híbrido en vez de una CNN es el tiempo de simulación ya que mientras un algoritmo puede demorar segundos o a lo mucho unos minutos, una CNN demorará en correr completamente aproximadamente en una hora y media como sucedió en este proyecto y la GPU necesaria sería mucho mayor.

Conclusiones

Primera. Hacer el debido análisis de suelo determina cuales son las deficiencias del suelo de cultivo, sin embargo, estos análisis tienden a acaparar una cantidad considerable de días lo que hace difícil tener un seguimiento continuo y actualizado del estado del suelo de cultivo.

Segunda. El aprendizaje automático y, en particular, las redes neuronales están avanzando rápidamente hasta el punto de que son parte esencial en muchos procesos. Bajo este enfoque, la agricultura de precisión se muestra como una forma muy útil de aprovechar dichas tecnologías, para lo cual, será necesario el desarrollo de investigaciones para la implementación de modelos y técnicas que ayuden a sustituir los modelos tradiciones, generando así, un impacto positivo que ayude a mejorar la producción agraria en la región.

Tercera. Antes de que las imágenes ingresen por la entrada del modelo estas deberán ser estandarizadas, para esto existen diversas técnicas de data standarization y data augmentation que ayudará a tener la data en el mismo contexto y así reducir el porcentaje de error al momento de correr el modelo.

Cuarta. Las gráficas de entrenamiento y validación de pérdidas obtenidas del uso de redes neuronales nos indican como está funcionando esta última; se concluyó que, para mejorar el modelo, una de las soluciones es aumentar el número de épocas. Por lo que se probó con 25 y 50 épocas, se observó como en las gráficas obtenidas con el uso de 50 épocas el error disminuyó. Aumentar la data es otra alternativa que puede ayudar específicamente si se quiere mejorar la validación del modelo.

Quinta. Al comparar el tiempo en el que demora en correr la red neuronal convolucional respecto a los filtros convolucionales con el algoritmo Random Forest, se puede llegar a concluir que el uso de este último es mucho más rápido para generar resultados que la CNN ya que este algoritmo termina de correr en 7 segundos mientras que la CNN puede demorar hasta dos horas como sucedió en el caso de este proyecto.

Sexta. Al comparar los resultados obtenidos tanto para la data de macronutrientes como para la data de enfermedades, se puede decir que se consiguen mejores resultados con la data de deficiencia de macronutrientes debido a que hay un mejor equilibrio entre la cantidad de imágenes que contiene para cada clase, mientras que para la data de

enfermedades no existe una relación proporcional entre las imágenes de cada una de las clases. Además, el fondo de las imágenes de enfermedades ya está tratadas con un fondo blanco que no permite obtener mejores resultados. La recomendación es usa un fondo negro que otorga mejor contraste.

Séptima. La matriz confusión nos muestra el nivel de precisión para cada una de las clases, se podrá caracterizar la estructura llevada a cabo en el algoritmo como apta o no para poder obtener de ella resultados confiables. Se considerarán confiables cuando sobrepasen el 85% de precisión como se observó con las CNN y los algoritmos de clasificación híbridos, mientras que, aquellos que estén por debajo de este nivel, estarán sujetos a métodos para poder mejorar su precisión ya sea mediante el mejoramiento de la red con la que se esté trabajando, así como también, el trabajo con una mayor cantidad de datos, árboles de decisiones, etc.

Octava. En el presente trabajo se concluye que el uso de Random Forest con filtros convolucionales de una arquitectura VGG16 ayuda a desarrollar un proceso de clasificación para las imágenes óptimo para el data set de macronutrientes, en donde el criterio de selección se basa en la máxima cantidad de votos dados por cada uno de los árboles para una etiqueta en específico. Este algoritmo híbrido, un método muy eficaz para la clasificación. En desempeño le sigue SVC para el data set de enfermedades, CNN y finalmente KNN con la precisión más baja.

Referencias bibliográficas

Aditi Shah, Prema Grupta, Porf. Y. M. Ajgar. “Macro-Nutrient Deficiency Identification in Plants Using Image Processing and Machine Learning” 2019.

Albawi, Saad, Tareq Abed Mohammed, y Saad Al-Zawi. 2017. «Understanding of a convolutional neural network». Proceedings of 2017 International Conference on Engineering and Technology, ICET 2017 2018-Janua(April 2018):1-6. doi: 10.1109/ICEngTechnol.2017.8308186.

Amirtha T, Gokulalaskshmi T, Umamaheswari P, T Rajasekar M. Tech. “Machine Learning based nutrient deficiency detection in crops”. International journal of recent Technology and Engineering. Volume-8 Issue6, Marzo 2020.

Anón. 2020. Scikit Learn User Guide.

Bagnato, Juan Ignacio. 2017. «Random Forest, el poder del Ensamble». Recuperado (https://www.aprendemachinelearning.com/random-forest-el-poder-del-ensamble/).

Bhandare, Ashwin, Maithili Bhide, Pranav Gokhale, y Rohan Chandavarkar. 2016. «Applications of Convolutional Neural Networks». International Journal of Computer Science and Information Technologies 7(5):2206-15.

Chollet, Francois. 2018. «16-Keras». Deep Learning with Python 97-111. doi: 10.1007/978-1-4842-2766-4_7.

Cortés Antona, Carlos. 2017. «Herramientas Modernas En Redes Neuronales: La Librería Keras». Universidad Autónoma de Madrid 60. doi: 1617 032 CO.

Cunningham, P´adraig, y Sarah Jane Delany. 2007. «K -Nearest Neighbour Classifiers». Multiple Classifier Systems (April 2007):1-17. doi: 10.1016/S0031-3203(00)00099-6.

DECHEN, ANTONIO ROQUE, y Gilmar Ribeiro NACHTIGALL. 2007. «Elementos requeridos à nutrição de plantas.» Embrapa Uva e Vinho-Capítulo em livro científico (ALICE).

Díaz-Alejo, Moreno. s. f. «Análisis comparativo de arquitecturas de redes neuronales para la clasificación de imágenes». 85.

Durán Suárez, Jaime, Alejandro Del, Real Torres, y Durán Suárez. 2017. «Redes Neuronales Convolucionales en R .78.

Gutiérrez Castorena, Edgar Vladimir, Ma del Carmen Gutiérrez Castorena, y Carlos Alberto Ortiz Solorio. 2015. «Manejo integrado de nutrientes en sistemas agrícolas intensivos: revisión». Revista mexicana de ciencias agrícolas 6(1):201-15.

Gylberth, Roan. 2018. «An Introduction to AdaGrad».

Harrison. Onel. 2018. «Machine Learning Basics with the K-Nearest Neighbors Algorithm | by Onel Harrison | Towards Data Science».

Hernández, Emmanuel F. Ramırez, Héctor Rodrıguez Rangel, Vıctor González Huitrón, Juan J. Flores, y Vicenç Puig. 2019. «Optimización Distribuida de Redes Convolucionales para la Clasificación de Imágenes». Research in Computing Science 148:213-26.

Infoagro Systems S.L. s. f. «El Cultivo de Platano». Recuperado 4 de septiembre de 2020 (https://www.infoagro.com/frutas/frutas_tropicales/platano.htm#:~:text=Los suelos aptos especialmente en materias nitrogenadas.).

J. A. Tejada y G. P. P. Gara, «LeafcheckIT: a banana leaf analyzer for identifying macronutrient deficiency», en Proceedings of the 3rd International Conference on Communication and Information Processing - ICCIP ’17, Tokyo, Japan, 2017, pp. 458-463, doi: 10.1145/3162957.3163035

Kumar, Arun, y Supriya P. Panda. 2019. «A Survey: How Python Pitches in IT-World». Proceedings of the International Conference on Machine Learning, Big Data, Cloud and Parallel Computing: Trends, Prespectives and Prospects, COMITCon 2019 248-51. doi: 10.1109/COMITCon.2019.8862251.

Maeda Gutiérrez, Valeria. 2019. «Comparación de arquitecturas de redes neuronales convolucionales para la clasificación de enfermedades en tomate».

Martos, Antonio Jesús Ortiz, María Teresa Martín Valdivia, L. Alfonso Ureña López, y Miguel Ángel García Cumbreras. 2005. «Detección automática de spam utilizando regresión logística bayesiana». Procesamiento del Lenguaje Natural (35):127-33.

Ministerio de Agricultura y Riego. s. f. «Clasificación de Suelos». Recuperado 24 de agosto de 2020a (http://minagri.gob.pe/portal/objetivos/43-sector-agrario/suelo/330-clasificacion#:~:text=En los desiertos predominan los,arcillosos y alcalinos (vertisoles).).

Montesdeoca Santana, Besay. 2016. «Estudios de predicción en series temporales de datos meteorológicos utilizando redes neuronales recurrentes».

Palmer,A., & Montaño, JJ (2002). Redes Neuronales Artificiales aplicadas al Análisis de Datos(Tesis doctoral).Universitat de les illes Balears, Mallorca.

Pedregosa, Fabian, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot,

y Édouard Duchesnay. 2011. «Scikit-learn: Machine learning in Python». Journal of Machine Learning Research 12(January):2825-30.

Peterson, Leif E. 2009. «K-nearest neighbor - Scholarpedia».

Reichel, Helena, Silvio Belalcázar, Gladys Múnera, Emilio Arévalo, y Javier Narváez. 1996. «Primer Reporte del Virus del Rayado del Banano (BSV) Afectando Plantaciones de Plátano Wuso AAB Simmonds), Caña de azúcar (Sochorum offícinorum) y Achira (Canna edulis) en Colombia». Ciencia y Tecnología Agropecuaria 1(1):35-39.

Rumelhart, David E., Geoffrey E. Hinton, y Ronald J. Williams. 1986. «Learning representations by back-propagating errors». Nature 323(6088):533-36. doi: 10.1038/323533a0.

Rumiche, Vega. 2015. «Tesis para optar el título de ingeniero agrónomo presentado por br. giov ana jandira vega rumiche».

Santana Mansilla, Pablo Fernando, Rosanna Nieves Costaguta, y Daniela Missio. 2014. «Aplicación de Algoritmos de Clasificación de Minería de Textos para el Reconocimiento de Habilidades de E-tutores Colaborativos».

Srivastava, Nitish. 2014. «Dropout: A simple way to prevent neural networks from overfitting».

StackOverflow. s. f. Aprendizaje Keras.

Tato, Ange, y Roger Nkambou. 2018. «IMPROVING ADAM OPTIMIZER».

T. Tran, J. Choi, T. H. Le, and J. Kim, “applied sciences A Comparative Study of Deep CNN in Forecasting and Classifying the Macronutrient Deficiencies on Development of Tomato Plant,” 2019.

Teuwen, Jonas, y Nikita Moriakov. 2019. «Convolutional neural networks». en Handbook of Medical Image Computing and Computer Assisted Intervention.

Torres, Swing. 2012. «Guía práctica para el manejo de banano orgánico en el valle del Chira». Hidalgo Impresores E.I.R.L. 72.

Vegas, Ulises. 2013. «menejo integrado». 1-25.

Waddell, P. 2010. «Introduction to Python , Numpy Introduction to Python». (June):1-26.

Lili Ayu Wulandhari, Alexander Agung Santoso Gunawan, Arie Qurania, Prihastuti Harsani, Triastinurmiatiningshi, Ferdy Tarawan, Riska Fauzia Hermawan. 2019. “Plant nutrient defiency detection using deep convolutional neural network” ICIC Express Letters. Volume 13, number 10, octubre.

Zhang, Quan. 2018. «Convolutional Neural Networks».

Zhang, Zijun. 2018. «Improved Adam Optimizer for Deep Neural Networks». Pp. 1-2 en 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS).

Zhuang, Fuzhen, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, y Qing He. 2020. «A Comprehensive Survey on Transfer Learning». Proceedings of the IEEE 1-31. doi: 10.1109/JPROC.2020.3004555.

(2013, mayo 8). Retrieved from http://randomforest2013.blogspot.com/2013/05/randomforest-definicion-random-forests.html

A., D. (n.d.). Ubunlog. Retrieved from https://ubunlog.com/spyder-entorno-desarrollo-python/

webdesigncusco. (2019, diciembre 12). Retrieved from https://webdesigncusco.com/principales-caracteristicas-de-sublime-text-3/

Yang, K., Qinami, K., Fei-Fei, L., Deng, J., & Russakovsky, O. (2019). Towards fairer datasets: Filtering and balancing the distribution of the people subtree in the ImageNet hierarchy. FAT* 2020 - Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency, 547–558. https://doi.org/10.1145/3351095.3375709