MANUAL TÉCNICO

Clasificación temática y automática de imágenes de la red social Pinterest con deep learning.

Aldo Isaac Hernández Antonio [1], Ana María Lizanette Becerra Cortés[1], Braulio José Baca Barbosa[1], Diana Martínez Frías [1], Diego Armando Gutiérrez Ayala[1], Mariana Esmeralda Centeno Reyes [1], Juan Carlos Gómez Carranza [1*]
[1]Departamento de Ingeniería Electrónica, División de Ingenierías Campus Irapuato-Salamanca, Universidad de Guanajuato.
{ai.hernandezantonio, aml.becerracortes, bj.bacabarbosa, d.martinezfrias, da.gutierrez.ayala, me.centenoreyes, jc.gomez}@ugto.mx

Para llevar a cabo la categorización de imágenes en Pinterest, se separó el código en tres fases:

En la primera etapa se lleva a cabo el preprocesamiento, aquí se encuentra un módulo de utilería encargado de la obtención de imágenes y tableros.

La segunda fase contiene todos los modelos de la red neuronal utilizada para la vectorización de las imágenes. Además de otro módulo de utilería responsable de almacenar y cargar el estado de ejecución de los modelos, dar creación y cargar lotes, entre otras cosas.

La última etapa es la encargada de la clasificación de las imágenes. Está compuesta por un módulo con funciones de utilería y el archivo responsable de llevar a cabo la clasificación.

1. Primer Código: prep/paths.py

Su objetivo principal es proporcionar funciones para conseguir las imágenes que nos son útiles, y desechar las que no. Se utilizan las siguientes bibliotecas para lograrlo.

```
from pandas import read_csv
from PIL import Image
from time import time
import numpy as np
import os
```

Contiene las funciones descritas a continuación:

La primera recibe la ruta de una imagen como parámetro, simplemente se encarga de verificar si la imagen proporcionada está corrupta o no.

```
def __is_corrupted(img_path):
  try:
    img = Image.open(img_path)
    img.verify()
    return False
  except:
    return True
```

find_imgs obtiene la estructura de directorios y lo recorre con la intención de catalogar las imágenes.
En el inicio de la función, se definen las variables de los archivos de salida, así como algunos contadores para conocer la cantidad de imágenes en total, las imágenes útiles y las corruptas.

Se define el tiempo de inicio, se abre cada imagen dentro de la ruta recibida y se agregan al archivo del total de imágenes. Si la imagen en cuestión se puede abrir, se añade al archivo de imágenes funcionales; de lo contrario, se incluye en el de imágenes corruptas.

```
def find_imgs(data_path):
  every_img = open("../output/every_image.txt", "w")
  working_imgs = open("../output/0_working_images.txt", "w")
  corrupt_imgs = open("../output/1_corrupt_images.txt", "w")

  # Counters
  total = 0
  good = 0
  bad = 0

  start = time()
  for dir, sub, files in os.walk(data_path):
    files = [dir + "\\" + file for file in files]
    for file in files:
      if os.path.isfile(file) and file.endswith(".jpg"):
        every_img.write(file + "\n")
        total += 1
        if(not __is_corrupted(file)):
          working_imgs.write(file + "\n")
          good += 1
        else:
          corrupt_imgs.write(file + "\n")
          bad += 1
```

Al salir del ciclo, se obtiene el tiempo de finalización y se cierran los archivos empleados. La función devuelve las cantidades de cada categoría de imágenes, así como el tiempo de término.

```
  end = time() - start

  corrupt_imgs.close()
  working_imgs.close()

  return (total, good, bad, end)
```

get_data() crea un diccionario conteniendo todos los usuarios y sus respectivos tableros y pines. Podría considerarse una ampliación de la función anterior, ya que además de obtener las imágenes útiles, descarta las imágenes corruptas e incluso ignora aquellas sin categoría o cuya categoría sea 'other'. Asimismo, devuelve un timer para dar a conocer el tiempo de ejecución.

```
def get_data(data_path):
  # Each user in data_path should be a user
  data = {
    "path": data_path,
    "users": {user: {} for user in os.listdir(data_path)}
  }

  timer = time()
  for user, boards in data["users"].items():
    # per_board is the boards folder for each user
    user_path = f"{data_path}\\{user}\\per_board"
    if os.path.exists(user_path):
      # Get everything in that folder
      board_names = os.listdir(user_path)
      # Filter files. Only folders should be boards
      board_names = [board for board in board_names if
        os.path.isdir(f'{user_path}\\{board}')]
      # Each board will have its category and its corresponding pins
      boards.update({
        board: {"category": "", "pins": []} for board in board_names
      })

      # Iterate over each board
      for board, content in boards.items():
        board_path = f"{user_path}\\{board}"
        # if the board category file exists, it will be set, else, it will
        # be an empty string
        if os.path.exists(f"{board_path}\\board_category.txt"):
          with open(f"{board_path}\\board_category.txt", "r") as file:
            content["category"] = "".join(file).strip()

        # Same process as the boards, only with pins
        pin_ids = os.listdir(board_path)
        pin_ids = [pin for pin in pin_ids if
          os.path.isdir(f'{board_path}\\{pin}')]
        content["pins"] += pin_ids

  timer = time() - timer

  return (data, timer)
```

write_user_to_csv almacena la información de los usuarios en tres archivos del tipo CSV (Comma Separated Value), separando las imágenes funcionales de las que no.

Recibe la variable booleana ***Path*** que funciona como bandera. Cuando es activada, su propósito es añadir una columna extra a los archivos con la ubicación exacta de cada imagen.

```
def write_user_to_csv(data, path: False):
  output_categories = open("../output/6_categories.txt", "w")

  if path:
    output_raw_csv = open("../output/2_raw_data.csv", "w")
    output_raw_csv.write(
     "user,board,pin,category,corrupted,rejected,path\n"
    )

    output_reject_csv = open("../output/4_rejected_data.csv", "w")
    output_reject_csv.write("user,board,pin,category,corrupted,path\n")

    output_clean_csv = open("../output/3_clean_data.csv", "w")
    output_clean_csv.write("user,board,pin,category,path\n")

  else:
    output_raw_csv = open("../output/2_raw_data.csv", "w")
    output_raw_csv.write("user,board,pin,category,corrupted,rejected\n")

    output_reject_csv = open("../output/4_rejected_data.csv", "w")
    output_reject_csv.write("user,board,pin,category,corrupted\n")

    output_clean_csv = open("../output/3_clean_data.csv", "w")
    output_clean_csv.write("user,board,pin,category\n")
```

Recorre la estructura ***data***, devuelta por la función descrita anteriormente, y las va catalogando en cada archivo, dependiendo de si se pueden abrir y si contienen una categoría aceptable.

```
  # Flags, only for debugging purposes
  corrupted = False
  rejected = False

  categories = set()

  timer = time()
  for user, boards in data["users"].items():
    for board, content in boards.items():
      category = content["category"]
      categories.add(content["category"])
```

```
    for pin in content["pins"]:
      pin_path =
        f"{data['path']}\\{user}\\per_board\\{board}\\{pin}\\pin_closeup_image.jpg"

    if(os.path.exists(pin_path)):
      if __is_corrupted(pin_path):
        corrupted = True
        rejected = True

        if path:
          output_raw_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{rejected},{pin_path}\n"
          )
          output_reject_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{pin_path}\n"
          )
        else:
          output_raw_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{rejected}\n"
          )
          output_reject_csv.write(
            f"{user},{board},{pin},{category},{corrupted}\n"
          )
    else:
      corrupted = False
      if category == "" or category == "other":
        rejected = True
        if path:
          output_raw_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{rejected},{pin_path}\n"
          )
          output_reject_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{pin_path}\n"
          )
        else:
          output_raw_csv.write(
            f"{user},{board},{pin},{category},{corrupted},{rejected}\n"
          )
          output_reject_csv.write(
            f"{user},{board},{pin},{category},{corrupted}\n"
          )
      else:
        rejected = False
        if path:
```

```
            output_raw_csv.write(
              f"{user},{board},{pin},{category},{corrupted},{rejected},{pin_path}\n"
            )
            output_clean_csv.write(
              f"{user},{board},{pin},{category},{pin_path}\n"
            )
          else:
            output_raw_csv.write(
              f"{user},{board},{pin},{category},{corrupted},{rejected}\n"
            )
            output_clean_csv.write(f"{user},{board},{pin},{category}\n")
```

Después, guarda las rutas a las imágenes en un archivo `.txt`. Asimismo, en un cuarto archivo `.csv`, carga todas las categorías encontradas en el conjunto de datos.
Por último, devuelve el tiempo que tardó en ejecutarse.

```
      timer = time() - timer

      path_txt = open("../output/5_path.txt", "w")
      path_txt.write(data["path"])
      path_txt.close()

      categories = list(categories)
      categories.sort()

      for cat in categories:
            if(cat == ""): output_categories.write("none" + "\n")
            else: output_categories.write(cat + "\n")

      output_reject_csv.close()
      output_clean_csv.close()
      output_raw_csv.close()
      output_categories.close()

      return timer
```

`import_data_from_csv` carga el archivo de imágenes "limpias" generado por la función anterior. Accede a cada una de las imágenes utilizando el archivo de rutas creado, utiliza dicha información para formar un diccionario, el cual será retornado para así poder trabajar con las imágenes útiles.

```
def import_data_from_csv(path):
  timer = time()
  data_path = ""
```

```
  with open(path + "5_path.txt", "r") as file:
    data_path = data_path.join(file).strip()
    del file

    clean_data = read_csv(path + "3_clean_data.csv")

    data = {
      "path": data_path,
      "users": {
        user: {
          board: {
            "category": "",
            "pins": []
          } for board in clean_data[clean_data["user"] ==
              [user]["board"].tolist()
        } for user in set(clean_data["user"].tolist())
      }
    }

    del data_path

    for user, board, pin, category in zip(clean_data["user"].tolist(),
      clean_data["board"].tolist(), clean_data["pin"].tolist(),
      clean_data["category"].tolist()):
        data["users"][user][board]["category"] = category
        data["users"][user][board]["pins"].append(pin)

    del user, board, pin, category, clean_data
    timer = time() - timer
    return data, timer
```

get_paths_from_data se encarga de crear una lista cuyo contenido son todas las rutas de cada imagen, esto es en caso de que se prefiera generar los archivos creados por ***write_user_to_csv*** sin ruta.

```
def get_paths_from_data(data):
  paths = []

  for user, boards in data["users"].items():
    for board, content in boards.items():
    for pin in content["pins"]:
      paths.append(
        f'{data["path"]}\\{user}\\per_board\\{board}\\{pin}\\pin_closeup_image.jpg'
      )
```

```
  return paths
```

get_categories_list recibe la ruta del archivo que contiene todas las categorías. Con esta información elabora un diccionario en el que las claves representan las categorías y los valores representan el índice de cada categoría, comenzando en 0.

```
def get_categories_list(path):
  categories = {}

  with open(path + "\\6_categories.txt", "r") as file:
    i = 0
    for row in file:
      categories.update({ row.strip(): i })
      i += 1

  del i
  del row
  del file

  return categories
```

get_categories_from_csv utiliza el diccionario creado por *get_categories_list()* para reemplazar las categorías incluidas en el archivo **csv** de imágenes limpias por su respectivo índice. Además, devuelve la lista de índices de categorías.

```
def get_categories_from_csv(path):
  data_path = ""
  with open(path + "5_path.txt", "r") as file:
            data_path = data_path.join(file).strip()
  del file

  cats_list = get_categories_list(path)

  clean_data = read_csv(path + "3_clean_data.csv")
  categories = clean_data["category"].replace(cats_list).tolist()

  return categories
```

La última función, *print_data,* simplemente muestra en pantalla la estructura de usuarios, tableros y pines de una forma legible.

```
def print_data(data):
  print(f'From {data["path"]}:')
```

```
  for user, boards in data["users"].items():
    print(f'User: {user}')
    for board, content in boards.items():
      if(content["category"] != ""):
        print(f'\tBoard: {board} ({content["category"]})')
      else:
        print(f'\tBoard: {board} (NO CATEGORY)')

      for pin in content["pins"]:
        pin_path =
f"{data['path']}\\{user}\\per_board\\{board}\\{pin}\\pin_closeup_image.jpg"

        print(f'\t\t - {pin}', end=" ")
        if not os.path.exists(pin_path):
          print("(pin not available)")
        elif __is_corrupted(pin_path):
          print("(corrupted)")
        else:
          print()

      print()
```

Por último, el programa contiene un driver code por si se ejecuta directamente. Éste se encarga de ejecutar todas las funciones necesarias para crear los archivos mencionados anteriormente.

```
if __name__ == "__main__":
  # Path to database
  data_path = "D:\\aldoi\\Documents\\DICIS\\Veranos\\Verano 2023\\data"

  # If there are files from which we can fetch data
  if os.path.isdir("../output/"):
    print("Output directory exists")
    file = "../output/3_clean_data.csv"
    if os.path.exists(file):
      print(f'{file} exists, importing data')
      data, get_timer = import_data_from_csv("../output/3_clean_data.csv")
      print(
        f'Took {get_timer:.3f} seconds ({(get_timer / 3600):.3f} hours) to
          fetch data from csv file'
      )
    else:
      print(f"{file} doesn't exist, fetching data")

      data, get_timer = get_data(data_path)
```

```
    print(
      f'Took {get_timer:.3f} seconds ({(get_timer / 3600):.3f} hours) to
        fetch the images and verify them'
    )

    write_timer = write_user_to_csv(data, True)
    print(
      f'Took {write_timer:.3f} seconds ({(write_timer / 3600):.3f} hours)
        to write data into .csv file'
    )

    # Once filtered, read clean data
    data = import_data_from_csv("../output/")
    del file
else:
    print("Output directory doesn't exist. It will be created")
    os.mkdir("../output/")

    data, get_timer = get_data(data_path)
    print(
      f'Took {get_timer:.3f} seconds ({(get_timer / 3600):.3f} hours) to
        fetch the images and verify them')

    write_timer = write_user_to_csv(data, True)
    print(
      f'Took {write_timer:.3f} seconds ({(write_timer / 3600):.3f} hours)
        to write data into .csv file'
    )

    # Once filtered, read clean data
    data = import_data_from_csv("../output/3_clean_data.csv")
```

2. Segundo Código: feature_extractors/feature_utils.py

Como ya se mencionó, este código es el encargado de proporcionar las funcionalidades necesarias para ejecutar cada uno de los modelos de vectorización. Para ello se requieren las siguientes bibliotecas, además del modelo implementado por Keras.

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

from keras.applications.convnext import preprocess_input
import keras.utils as image
```

```
import matplotlib.pyplot as plt
import numpy as np
import time
import json
import os
```

La función ***new_stats*** genera un diccionario ***stats*** vacío, este objeto nos será de utilidad para conocer información importante sobre cada modelo: los lotes, su vectorización y el tiempo que se tomó en cada etapa.

```
def new_stats(TOTAL, BATCH_SIZE, TOTAL_BATCHES):
  stats = {
    "total_images": TOTAL,
    "batch_size": BATCH_SIZE,
    "total_batches": TOTAL_BATCHES,
    "batch_times": [],
    "vectorize_times": [],
    "avg_vectorize_times": [],
    "start_time": time.time(),
    "end_time": 0,
    "elapsed_time": 0,
    "last_batch": 0,
    "last_image": 0
  }
  return stats
```

restoration_point tiene la responsabilidad de encontrar un archivo de restauración llamado stats.json, actualizarlo con el estado actual del modelo y permitir la reanudación de la ejecución desde el punto en que quedó.

```
def restoration_point(data_path, model_name, TOTAL, BATCH_SIZE):
  open_mode = "w"

  if(TOTAL % BATCH_SIZE != 0): TOTAL_BATCHES = (TOTAL // BATCH_SIZE) + 1
  else: TOTAL_BATCHES = (TOTAL // BATCH_SIZE)

  if(os.path.isfile(f"{data_path}/extractors/{model_name}/stats.json")):
    with open(f"{data_path}/extractors/{model_name}/stats.json", "r")
      as output_stats:
        data = output_stats.read()

        if(data != ""):
          try:
            stats = json.loads(data)
            open_mode = "a"
```

```
            print(f"Found restoration point.")
          except Exception as ERR:
            print(
              f"An error has occurred while reading stats.json\n
                Error: {str(ERR)}"
            )
            open_mode = "w"
            stats = new_stats(TOTAL, BATCH_SIZE, TOTAL_BATCHES)
```

En caso de que el archivo encontrado esté vacío o no exista, se procede a generar uno nuevo utilizando la función ***new_stats***.
La función devuelve una cadena llamada ***open_mode***, la cual representa la forma de abrir el archivo csv que almacena las imágenes vectorizadas.

```
        else:
          print("stats.json exists but it's empty. Creating default stats")
          open_mode = "w"

          stats = new_stats(TOTAL, BATCH_SIZE, TOTAL_BATCHES)
  else:
    print("stats.json file doesn't exist. Creating default stats dict.")
    open_mode = "w"

    stats = new_stats(TOTAL, BATCH_SIZE, TOTAL_BATCHES)

    output_stats = open(f"{data_path}/extractors/{model_name}/stats.json", "w")
    json.dump(stats, output_stats)
    output_stats.close()

  return open_mode, stats
```

plot_times representa gráficamente la relación entre los datos proporcionados como argumento y su índice.

```
def plot_times(data, title, TOTAL_BATCHES, xlabel, ylabel):
  plt.figure(dpi = 120)
  plt.title(title)
  plt.xlabel(xlabel)
  plt.ylabel(ylabel)
  plt.plot(np.arange(0, TOTAL_BATCHES), data, "-", color = "red")
  plt.grid(color = "darkgray", linestyle = "dashed", linewidth = "0.5")

  return plt
```

save_state, como lo indica su nombre, tiene como propósito guardar el estado actual del modelo y actualizar el archivo ***stats.json*** con dicha información. Asimismo, utiliza la función ***plot_times*** para graficar el tiempo de creación de lotes, el tiempo de vectorización y su promedio.

```
def save_state(stats, output_stats, model_name, data_path, last_batch, last_image):
  stats["last_batch"] = last_batch
  stats["last_image"] = last_image

  stats["end_time"] = time.time()
  stats["elapsed_time"] = stats["end_time"] - stats["start_time"]

  json.dump(stats, output_stats)

  fig = plot_times(
    stats["batch_times"],
    f"Batch creation times:
      {model_name} ({stats['total_batches']} batches of {stats['batch_size']}
        images) (interrupted)",
    len(stats["batch_times"]),
    "Batch",
    "Time (s)"
  )
  fig.savefig(
    f"{data_path}/extractors/{model_name}/batch_times.png",
    dpi = 280
  )

  fig = plot_times(
    stats["vectorize_times"],
    f"Vectorizing times: model {model_name} (interrupted)",
    len(stats["vectorize_times"]),
    "Batch",
    "Time (s)"
  )
  fig.savefig(
    f"{data_path}/extractors/{model_name}/vectorizing_times.png",
    dpi = 280
  )

  fig = plot_times(
    stats["avg_vectorize_times"],
    f"Average vectorizing times: model {model_name} (interrupted)",
    len(stats["avg_vectorize_times"]),
    "Batch",
```

```
  "Time (ms)"
  )
  fig.savefig(
    f"{data_path}/extractors/{model_name}/avg_vectorizing_times.png",
    dpi = 280
  )
```

create_batch se encarga de crear un lote al cargar en memoria la cantidad de imágenes especificada. Posteriormente, realiza el preprocesamiento de estas imágenes y las agrega a una lista de imágenes preprocesadas, que finalmente es devuelta como resultado.

```
def create_batch(BATCH_SIZE, image_paths):
  batch = []
  for i in range(BATCH_SIZE):
    img = image.load_img(image_paths.pop(0), target_size = (128, 128))
      temp = image.img_to_array(img)
      temp = np.expand_dims(temp, axis = 0)
      temp = preprocess_input(temp)

      batch.append(temp)

  return batch
```

Finalmente, *vectorize_batch* recorre la lista de imágenes recibida como argumento. obtiene el vector de características representativo de cada imagen y realiza una predicción para determinar su posible ubicación en el modelo.
El resultado es almacenado en el archivo de vectores proporcionado y la memoria es liberada.

```
def vectorize_batch(TOTAL, BATCH_SIZE, i, output, batch, model):
  j = 0
  vectorizing_times = []

  while(len(batch) > 0):
    print(
      f"\t\t - Vectorizing from batch {i}: ({(BATCH_SIZE * i) + j + 1}/{TOTAL},
      {(((BATCH_SIZE * i) + j + 1)/TOTAL) * 100 :.2f}%)"
    )
    try:
      vector_elapsed = time.time()
      tmp_img = batch.pop(0)
      feat_vector = model.predict(tmp_img)
      vector_elapsed = time.time() - vector_elapsed
      vectorizing_times.append(vector_elapsed)

      out = ",".join([str(i) for i in feat_vector.flatten()])
      output.write(out + "\n")
```

```
      j += 1
      del tmp_img
    except KeyboardInterrupt:
      raise KeyboardInterrupt(
        "Program was interrupted by the user while vectorizing the images.
        Last image index will be returned.",
        (BATCH_SIZE * i) + j,
        (sum(vectorizing_times) / BATCH_SIZE) * 1000)

    print()

  return (BATCH_SIZE * i) + j, (sum(vectorizing_times) / BATCH_SIZE) * 1000
```

3. Tercer Código: feature_extractors/ ConvNeXt_base.py

Con el fin de explicar su propósito, se eligió este archivo en representación de los 13 modelos de red neuronal utilizados. Todos estos archivos llevan a cabo el mismo procedimiento, solamente cambia el modelo a emplear; éste, en particular, utiliza la arquitectura ConvNeXtBase implementada por el equipo de Keras.

Se utilizan tanto los métodos creados en el archivo feature_utils.py como los del archivo paths.py, además de hacer uso de las siguientes librerías.

```
from keras.applications.convnext import ConvNeXtBase

import feature_utils as utils
import json
import time
import os

import sys
sys.path.append("../prep/")
from paths import import_data_from_csv, get_paths_from_data
```

Al ejecutar el archivo, los datos creados en paths.py son importados, y a partir de la estructura proporcionada por dichos datos, se generan las rutas de las imágenes. Sucesivamente se crea una carpeta para almacenar los datos producidos por la ejecución.

```
# Driver program
if(__name__ == "__main__"):
  data_path = "D:\\data\\2023_aldo_beca\\output"
  data, _ = import_data_from_csv(data_path + "\\")
  image_paths = get_paths_from_data(data)
  del data
```

```
if(not (os.path.isdir(f"{data_path}/extractors"))):
  print("Creating feature extractors folder")
  os.mkdir(f"{data_path}/extractors")
```

Tras eso, el modelo es generado indicando los pesos pre-entrenados por ImageNet. También se le indica que no debe llegar a la última capa, lo que implica que el modelo devuelve el vector de características ubicado en la capa previa a la predicción. Este vector será utilizado para entrenar otro modelo, permitiéndonos realizar nuestras propias inferencias.

```
model = ConvNeXtBase(
  model_name = "convnext_base",
  weights = "imagenet",
  include_top = False,
  pooling = "avg"
)
model_name = model._name

if(not (os.path.isdir(f"{data_path}/extractors/{model_name}"))):
  print(f"Creating {model_name} feature extractors folder")
  os.mkdir(f"{data_path}/extractors/{model_name}")
```

Las imágenes por procesar se agruparon en lotes. En cada uno de éstos, se procederá a cargar todas sus respectivas imágenes en memoria, posteriormente se aplicará el preprocesamiento y se llevará a cabo la vectorización del lote en cuestión. Al concluir, los resultados obtenidos y el tiempo empleado serán registrados en el archivo ***stats.json***. Se repetirá este proceso hasta que ya no haya más ilustraciones pendientes.

Antes de comenzar, el código se encarga de buscar un punto de recuperación, utilizando el sistema de restauración implementado. Si lo encuentra, reanuda la ejecución a partir de ese punto.

```
BATCH_SIZE = 2000

# Find restoration point
open_mode, stats = utils.restoration_point(
  data_path,
  model_name,
  TOTAL = len(image_paths),
  BATCH_SIZE = BATCH_SIZE
)

TOTAL = stats["total_images"]
BATCH_SIZE = stats["batch_size"]
TOTAL_BATCHES = stats["total_batches"]

i = stats["last_batch"]
```

```
temp_paths = image_paths[stats["last_image"]:]

with open(f"{data_path}/extractors/{model_name}/pin_vectors.csv", open_mode)
  as output, open(f"{data_path}/extractors/{model_name}/stats.json", "w")
    as output_stats:
      while(len(temp_paths) > BATCH_SIZE):
        batch = []

        try:
          print(f"\t- Creating batch {i}/{TOTAL_BATCHES}...", end = "")
          batch_elapsed = time.time()
          batch = utils.create_batch(BATCH_SIZE, temp_paths)
          batch_elapsed = time.time() - batch_elapsed
          stats["batch_times"].append(batch_elapsed)
          print(
            f"\t! Batch created with length {len(batch)} in
              {batch_elapsed:.6f} seconds"
          )
        except KeyboardInterrupt as ERR:
          print(f" Batching process was interrupted by the user. Saving
            before losing information.\n{str(ERR)}")
          output.close()

          utils.save_state(
            stats,
            output_stats,
            model_name,
            data_path,
            i,
            (i * BATCH_SIZE)
          )

          break

        try:
          print(f"\t- Vectorizing batch {i}")
          vectorize_elapsed = time.time()
          stats["last_image"], avg_vectorize_time =
            utils.vectorize_batch(
              TOTAL,
              BATCH_SIZE,
              i - (open_mode == "a"),
              output,
              batch,
```

```
                model
              )
            vectorize_elapsed = time.time() - vectorize_elapsed
            stats["vectorize_times"].append(vectorize_elapsed)
            stats["avg_vectorize_times"].append(avg_vectorize_time)
            print(f"\t! Batch vectorized in {vectorize_elapsed:.6f} seconds,
              Average time: {avg_vectorize_time:.2f}ms\n")

          except KeyboardInterrupt as ERR:
            print(f" Vectorizing process was interrupted by the user. Saving
              before losing information.\n{str(ERR)}")
            output.close()

            utils.save_state(
             stats,
             output_stats,
             model_name,
             data_path,
             i,
             ERR.args[1]
           )

            break

        stats["last_batch"] = i
        i += 1
  else:
    batch = []

    try:
      print(f"\t- Creating last batch {i}/{TOTAL_BATCHES}...", end = "")
        batch_elapsed = time.time()
        batch = utils.create_batch(len(temp_paths), temp_paths)
        batch_elapsed = time.time() - batch_elapsed
        stats["batch_times"].append(batch_elapsed)
        print(f"\t! Batch created with length {len(batch)} in
          {batch_elapsed:.6f} seconds")
    except KeyboardInterrupt as ERR:
      print(f" Batching process was interrupted by the user. Saving before
        losing information.\n{str(ERR)}")
      output.close()

      utils.save_state(
        stats,
```

```
          output_stats,
          model_name,
          data_path,
          i,
          (i * BATCH_SIZE)
        )
        stats["avg_vectorize_times"].append(ERR.args[2])

      try:
        print(f"\t- Vectorizing batch {i}")
        vectorize_elapsed = time.time()
        stats["last_image"], avg_vectorize_time =
          utils.vectorize_batch(
            TOTAL,
            BATCH_SIZE,
            i - (open_mode == "a"),
            output,
            batch,
            model
          )
        vectorize_elapsed = time.time() - vectorize_elapsed
        stats["vectorize_times"].append(vectorize_elapsed)
        stats["avg_vectorize_times"].append(avg_vectorize_time)
        print(f"\t! Batch vectorized in {vectorize_elapsed:.6f} seconds,
          Average time: {avg_vectorize_time:.2f}ms\n")
      except KeyboardInterrupt as ERR:
        print(f" Vectorizing process was interrupted by the user. Saving
          before losing information.\n{str(ERR)}")
          output.close()

          utils.save_state(
            stats,
            output_stats,
            model_name,
            data_path,
            i,
            ERR.args[1]
          )

          stats["last_batch"] = i

          i += 1
stats["end_time"] = time.time()
stats["elapsed_time"] = stats["end_time"] - stats["start_time"]
```

```
json.dump(stats, output_stats)
```

Al salir del ciclo, se muestra en pantalla el tiempo que ha tomado crear el archivo de vectores de características. Asimismo, se generan las gráficas que representan el tiempo de creación de lotes, el tiempo de vectorización y su promedio, con ayuda de la función plot_times. Finalmente, son almacenadas para su posterior análisis.

```
print(
 f'Took {stats["elapsed_time"]:.6f} seconds to create feature vectors file'
)
print(
  f'Took {stats["elapsed_time"]/60  :.6f} minutes to create feature vectors file'
)
print(
  f'Took {stats["elapsed_time"]/3600:.6f}   hours to create feature vectors file'
)

fig = utils.plot_times(
  stats["batch_times"],
  f"Batch creation times: {model_name} ({TOTAL_BATCHES} batches of
    {BATCH_SIZE} images)",
  len(stats["batch_times"]), "Batch", "Time (s)"
)
fig.savefig(
  f"{data_path}/extractors/{model_name}/batch_times.png", dpi = 280
)

fig = utils.plot_times(
  stats["vectorize_times"],
  f"Vectorizing times: model {model_name}",
  len(stats["vectorize_times"]),
  "Batch",
  "Time (s)"
)
fig.savefig(
  f"{data_path}/extractors/{model_name}/vectorizing_times.png", dpi = 280
)

fig = utils.plot_times(
  stats["avg_vectorize_times"],
  f"Average vectorizing times: model {model_name}",
  len(stats["avg_vectorize_times"]),
  "Batch",
```

```
    "Time (ms)"
  )
  fig.savefig(
    f"{data_path}/extractors/{model_name}/avg_vectorizing_times.png", dpi = 280
  )
```

4. Cuarto Código: classifiers/ utils.py

Este archivo está compuesto por un conjunto de funciones que serán utilizadas para la clasificación de los pines. Utiliza las siguientes librerías y módulos.

```
import sys

sys.path.append("../prep")

from paths import get_categories_from_csv
import matplotlib.pyplot as plt
from datetime import datetime
from time import time
import numpy as np
import random
import json
import os
```

import_vectors importa los vectores de características de cada imagen según el modelo especificado. Devuelve un conjunto de arreglos que representan a la imagen como un vector y el tiempo empleado en esta tarea.

```
def import_vectors(data_path, vector_model):
  vector_path = f"{data_path}\\extractors\\{vector_model}\\pin_vectors.csv"

  if os.path.isfile(vector_path):
    vectors = []
    i = 0
    with open(vector_path, "r") as file:
      elapsed = time()
      for vec in file:
        vectors.append(np.array([float(val) for val in vec.strip().split(",")]))
        i += 1
        if i % 100000 == 0:
          print(f"\tLoaded {i} images")
      vectors = np.array(vectors)
      elapsed = time() - elapsed

    return vectors, elapsed
```

```
  else:
    print(f"\tVector model {vector_model} hasn't been evaluated\n")
    return None, 0
```

import_vectors_and_labels realiza la misma tarea que la función anterior, con la diferencia de que también puede devolver la categoría de cada imagen, si se le indica.

```
def import_vectors_and_labels(data_path, vector_model, load_labels=True):
  vector_path = f"{data_path}\\extractors\\{vector_model}\\pin_vectors.csv"

  if os.path.exists(vector_path):
    vectors = []
    with open(vector_path, "r") as file:
      for vec in file:
        vectors.append(np.array([float(val) for val in vec.strip().split(",")]))
    vectors = np.array(vectors)

    if load_labels:
      labels = get_categories_from_csv(data_path + "\\")
      return vectors, labels

    return vectors
  else:
    print(f"Vector model {vector_model} hasn't been evaluated")
    return None
```

La función *save_dictionary_as_json*, como su nombre lo indica, convierte el diccionario proporcionado en una estructura JSON y lo guarda como archivo de texto en la ruta especificada.

```
def save_dictionary_as_json(route, dictionary):
  dictionary = {str(key): val for key, val in dictionary.items()}
  with open(route, "w") as openFile:
    json.dump(dictionary, openFile)
```

new_exec_stats crea un diccionario con las estadísticas esenciales para cada ejecución.

```
def new_exec_stats():
  return {
    "exec_start": datetime.now().strftime("%d/%m/%Y %H:%M:%S"),
    "exec_end": "",
    "data_load_start": 0,
    "data_load_end": 0,
    "data_load_elapsed": 0,
    "samples": 0,
    "features": 0,
```

```
    "max_samples_per_category": 0,
    "sampled_dataset_size": 0,
    "train_start": 0,
    "train_end": 0,
    "train_elapsed": 0,
    "test_start": 0,
    "test_end": 0,
    "test_elapsed": 0,
    "accuracy": 0,
    "f1": 0,
  }
```

new_stats, a diferencia de la función anterior, no solo genera un nuevo diccionario de estadísticas, sino que también guarda las estadísticas de cada ejecución. Éstas son almacenadas en el modelo de clasificación especificado para su resguardo.

```
def new_stats(classifier_model, vec_model):
  return {
    "first_execution": datetime.now().strftime("%d/%m/%Y %H:%M:%S"),
    "classifier": classifier_model,
    "vectorizing_model": vec_model,
    "last_exec": 0,
    "execs": {"exec_0": new_exec_stats()},
  }
```

dataset_workaround resuelve un problema de implementación. Se encarga de eliminar del conjunto de datos las imágenes cuya categoría sea '*None*' para no utilizarlas en la clasificación.

```
def dataset_workaround(img_vectors, img_labels):
 none_categories = sorted(
  [i for i, cat in enumerate(img_labels) if cat == "None" or cat == "none"],
    reverse=True,
 )

 for id in none_categories:
  img_vectors.pop(id)
  img_labels.pop(id)
```

```
  return np.array(img_vectors), np.array(img_labels)
```

get_idx_per_category_dictionary genera una estructura compuesta por cada categoría y una lista de los índices de las imágenes pertenecientes a dicha categoría.

```
def get_idx_per_category_dictionary(img_labels):
  idx_per_category = {cat: [] for cat in set(img_labels)}

  for i, cat in enumerate(img_labels):
    idx_per_category[cat].append(i)

  return idx_per_category
```

get_idx_per_category_sampled_dictionary recibe la estructura creada por la función anterior y un valor de umbral llamado "threshold". Si una categoría no pasa el umbral proporcionado, se copia directamente a un nuevo diccionario "muestreado". En caso contrario, se toma una muestra aleatoria de threshold elementos. Finalmente, se retorna el nuevo diccionario, así como la lista de categorías que pasaron directamente.

```
def get_idx_per_category_sampled_dictionary(idx_per_category, threshold):
  less_than_thresh = []
  idx_per_category_sampled = dict.fromkeys(idx_per_category, [])

  for cat, idx_list in idx_per_category.items():
    if(len(idx_list) > 0 and len(idx_list) < threshold):
      idx_per_category_sampled[cat] = idx_list.copy()
      less_than_thresh.append(cat)
    else:
      idx_per_category_sampled[cat] = random.sample(idx_list, threshold)

  return idx_per_category_sampled, less_than_thresh
```

load_stats carga en memoria el archivo de estadísticas para cada modelo de clasificación, si no existe, lo crea y devuelve el número de ejecución actual.

```
def load_stats(data_path, classifier_model, vec_model):
  last_exec = 0
  if os.path.isfile(data_path):
    # Load previous stats file
    print("Previous stats data file exists. Loading...")

    with open(data_path, "r") as input_stats:
      data = input_stats.read()
```

```
    if data != "":
      try:
        # There is already a previous stats. We only update the current
        # execution statistics.
        stats = json.loads(data)
        stats["last_exec"] += 1
        last_exec = stats["last_exec"]

        stats["execs"].update({f"exec_{last_exec}": new_exec_stats()})

        print(f"\t* Loaded previous stats file. Execution: {last_exec}\n")
      except Exception as ERR:
        print(f"An error has occurred while reading stats.json. Creating
          default stats.\nError: {str(ERR)}\n")
        stats = new_stats(classifier_model, vec_model)
    else:
      print("\tstats.json exists but it's empty. Creating default stats.\n")
      stats = new_stats(classifier_model, vec_model)
  else:
    print("stats.json doesn't exist. This is a fully new execution.\n")
    stats = new_stats(classifier_model, vec_model)

  return stats, last_exec
```

get_confusion_matrix construye la respectiva matriz de confusión del modelo. Esto con el objetivo de comparar las predicciones realizadas por el modelo con las etiquetas reales de la colección de datos.

```
def get_confusion_matrix(prediction, ground_labels, n_categories):
  conf_mat = np.zeros((n_categories, n_categories))
  for pred, g_truth in zip(prediction, ground_labels):
    conf_mat[g_truth, pred] += 1

  conf_mat_sklearn = metrics.confusion_matrix(
    y_true = test_labels,
    y_pred = prediction
  )

  return conf_mat
```

plot_confusion_matrix hace honor a su nombre y simplemente grafica la matriz de confusión construida para cada modelo de clasificación.

```
def plot_confusion_matrix(conf_mat, classifier_model, vec_model):
```

```
  _, ax = plt.subplots(figsize = (20, 20))
  ax.matshow(conf_mat, cmap = plt.cm.Blues, alpha = 0.3)
  for i in range(conf_mat.shape[0]):
    for j in range(conf_mat.shape[1]):
      ax.text(
        x = j,
        y = i,
        s = conf_mat[i, j],
        va = "center",
        ha = "center",
        size = "xx-large"
      )

  plt.xlabel("Predictions", fontsize = 18)
  plt.ylabel("Ground truth", fontsize = 18)
  plt.title(
    f"{classifier_model} + {vec_model} Confusion Matrix",
    fontsize = 18
  )
  plt.show()
```

El *driver program* de este archivo se encarga de abrir los archivos de vectores para importarlos a la memoria utilizando ***import_vectors***. Para cada uno de ellos, se muestran en pantalla sus dimensiones y el tiempo empleado en cargarlo. Al final, libera la memoria utilizada.

```
if __name__ == "__main__":
  data_path = "D:\\aldoi\\Documents\\DICIS\\Veranos\\Verano 2023\\output"

  # Opening vector files
  vector_models = [
    "convnext_base",
    "convnext_large",
    "convnext_small",
    "convnext_tiny",
    "convnext_xlarge",
    "efficientnet_v2_b0",
    "efficientnet_v2_b1",
    "efficientnet_v2_b2",
    "efficientnet_v2_b3",
    "efficientnet_v2_l",
    "efficientnet_v2_m",
    "efficientnet_v2_s",
    "xception"
  ]
```

```
classifier_models = []

y = get_categories_from_csv(data_path + "\\")

for model in vector_models[3:]:
  print(f"* Model {model}:")
  x, elapsed = import_vectors(data_path, model)

  if x != None:
    print(f"\t\tModel {model} has a shape of {len(x)},
      {x[0].shape[0]}\n\t\tloaded in {elapsed} seconds"
    )

    print(f"\nCleaning memory of model {model}...")
    elapsed = time()
    del x
    elapsed = time() - elapsed
    print(f"Memory clean in {elapsed} seconds\n")
  else:
    continue
```

5. Quinto Código: classifiers/ classifiers.py

Este último archivo importa el módulo de utilerías utils.py descrito recientemente. Además, se importan varias funciones y algoritmos de la biblioteca sklearn, que incluyen el clasificador de vecinos más cercanos, una implementación de una red neuronal multicapa y un algoritmo para la búsqueda de hiperparámetros para un estimador.

```
import sys

sys.path.append("../prep")

from paths import get_categories_list
import utils

from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import normalize
from datetime import datetime
from sklearn import metrics
from sklearn import svm
```

```
from time import time
import numpy as np
import os
```

Se enlistan todos los archivos de vectores y modelos de clasificación para su posterior uso. Después de elegir un clasificador, se define el directorio donde se almacenarán los resultados y se obtiene la lista de categorías del directorio de preprocesamiento.

```
vector_models = [
  "convnext_base",
  "convnext_large",
  "convnext_small",
  "convnext_tiny",
  "convnext_xlarge",
  "efficientnet_v2_b0",
  "efficientnet_v2_b1",
  "efficientnet_v2_b2",
  "efficientnet_v2_b3",
  "efficientnet_v2_l",
  "efficientnet_v2_m",
  "efficientnet_v2_s",
  "xception",
]

classifier_models = ["svm", "knn", "ann"]

# Change subscript to evaluate different classifier
classifier_model = classifier_models[1]
max_samples = 3000 # Change if needed

data_path = "D:\\data\\2023_aldo_beca\\output"
output_path = f"{data_path}/classifiers"

# Get all categories as a dictionary
categories = get_categories_list(data_path)
```

Para cada clasificador y modelo, se crea su respectivo directorio. A continuación, se carga el archivo de estadísticas y se almacena el número de muestras en él. Posteriormente, se genera la carpeta correspondiente a la ejecución actual.

```
for vecmod in vector_models:
  # Change this so you can evaluate distinct model combinations
  vec_model = vecmod

  print(f"About to train {classifier_model} classifier from vectors
    obtained with: {vec_model}")
```

```
# Creating output folder structure
classifier_path = f"{output_path}/{classifier_model}"
vector_path = f"{classifier_path}/{vec_model}"
execs_path = f"{vector_path}/execs"
stats_path = f"{vector_path}/stats.json"

if not os.path.isdir(output_path):
  print("Creating classifiers folder")
  os.mkdir(f"{data_path}/classifiers")

if not os.path.isdir(classifier_path):
  print(f"Creating classifier {classifier_model} folder")
  os.mkdir(classifier_path)

if not os.path.isdir(vector_path):
  print(f"Creating {classifier_model}'s {vec_model} folder")
  os.mkdir(vector_path)

if not os.path.isdir(execs_path):
  print("Creating execution's folder")
  os.mkdir(execs_path)

# Loading stats file
stats, last_exec = utils.load_stats(
  stats_path,
  classifier_model,
  vec_model
)
stats["execs"][f"exec_{last_exec}"]["max_samples_per_category"] = max_samples

# Creating n-th execution folder
nth_exec_path = f"{execs_path}/exec_{last_exec}"

if not os.path.isdir(nth_exec_path):
  print(f"Creating {last_exec}-th execution's folder")
  os.mkdir(nth_exec_path)
```

Se lleva a cabo la importación de los vectores de características asociados a cada imagen, así como la obtención de las etiquetas reales para cada ilustración. Después, se hace uso de la función implementada para eliminar las imágenes que poseen una categoría no relevante (*None*).

```
# Load image features from specific model
print("Loading images' vectors...")
```

```
  stats["execs"][f"exec_{last_exec}"]["data_load_start"] = time()
  img_vectors, load_time = utils.import_vectors(data_path, vec_model)
  stats["execs"][f"exec_{last_exec}"]["data_load_end"] = (
    load_time + stats["execs"][f"exec_{last_exec}"]["data_load_start"]
  )
  stats["execs"][f"exec_{last_exec}"]["data_load_elapsed"] = load_time

  print(f"\t\t! {len(img_vectors)} images loaded in {load_time} seconds\n")

  # Get ground truth labels
  print("Loading ground truth labels...")
  img_labels = utils.get_categories_from_csv(data_path + "/")[: len(img_vectors)]
  print("\t! Ground truth labels loaded\n")

  # Working around the fact that some categories are literally "None"
  img_vectors, img_labels = utils.dataset_workaround(
    img_vectors,
    img_labels
  )
  img_vectors = normalize(img_vectors)  # Normalize using Euclidean norm

  stats["execs"][f"exec_{last_exec}"]["samples"],
  stats["execs"][f"exec_{last_exec}"]["features"] = img_vectors.shape
```

Sucesivamente, se realiza un muestreo aleatorio de las imágenes de acuerdo con las categorías definidas. Luego, se crea un nuevo conjunto de datos con las imágenes seleccionadas, junto con sus etiquetas correspondientes.

```
# Doing random sampling on images
  print("Random sampling of images...")
  sampling_elapsed = time()

  idx_per_category = {}
  idx_per_category_sampled = {}

  # Obtaining indices of each image
  print("\t* Getting image indices per category...", end=" ")
  idx_per_category = utils.get_idx_per_category_dictionary(img_labels)
  utils.save_dictionary_as_json(
    f"{data_path}/idx_per_category.json",
    idx_per_category
  )
  print("Indices per category dictionary done.")

  # Get the category with less images.
  print("\t* Getting random indices per category...", end=" ")
```

```
idx_per_category_sampled, less_than_thresh =
  utils.get_idx_per_category_sampled_dictionary(
    idx_per_category,
    max_samples
  )
utils.save_dictionary_as_json(
  f"{nth_exec_path}/idx_per_cat_samples.json",
  idx_per_category_sampled
)
print(f"Each category now has at max {max_samples} indices.")
print(f"\t* The following categories have less than {max_samples} samples:
  {less_than_thresh}")

# Concatenating each each list of indices of each category into one
print("\t* Unifying random indices per category lists...", end=" ")
list_idx_selected_images = []
for samples in idx_per_category_sampled.values():
  list_idx_selected_images.extend(samples)
np.random.shuffle(list_idx_selected_images)
print("Created random indices list.")


# With the shuffled indices, create a new dataset
print("\t* Creating new dataset from sampling list...", end=" ")
new_dataset_elapsed = time()
img_vectors_sampled = np.array(
  [img_vectors[id] for id in list_idx_selected_images]
)
img_labels_sampled = np.array(
  [img_labels[id] for id in list_idx_selected_images]
)
new_dataset_elapsed = time() - new_dataset_elapsed
print(f"New dataset created in {new_dataset_elapsed:.3f} seconds.")

sampling_elapsed = time() - sampling_elapsed
print(f"\t\t! Sampling done in {sampling_elapsed:.3f} seconds.\n")

stats["execs"][f"exec_{last_exec}"]["sampled_dataset_size"] =
  img_vectors_sampled.shape[0]

m_samples, n_features = img_vectors_sampled.shape
idxs = np.arange(m_samples)
```

Se realiza la partición de la colección de datos en dos conjuntos distintos: uno de prueba y otro de entrenamiento.

```
# Splitting dataset
print("Splitting dataset in 80:20 ratio...")
split_elapsed = time()
(
  train_data,
  test_data,
  train_labels,
  test_labels,
  idx_tr,
  idx_ts,
) = train_test_split(
  img_vectors_sampled,
  img_labels_sampled,
  idxs,
  test_size=0.20,
  random_state=1,
  shuffle=True,
  stratify=None,
)

split_elapsed = time() - split_elapsed
print(f"\tDataset split in {split_elapsed:.3f} seconds\n")
```

Seguidamente, se procede a encontrar los mejores hiperparámetros dependiendo del clasificador que esté siendo evaluado en el momento. Esto con la finalidad de determinar los valores óptimos para maximizar el rendimiento y la precisión del modelo.

```
# Find best hyperparameters
grid = {}

if classifier_model == "svm":
  cs = [0.01, 0.1, 1.0, 10.0, 100.0]
  classifier = svm.LinearSVC(max_iter=10000)
  grid["C"] = cs
elif classifier_model == "knn":
  ks = [1, 5, 10, 20, 50]
  classifier = KNeighborsClassifier(algorithm="kd_tree", p=2)
  grid["n_neighbors"] = ks
elif classifier_model == "ann":
  hidden_layer_neurons = 200
  classifier = MLPClassifier(
    hidden_layer_sizes=(hidden_layer_neurons,),
    activation='relu',
    solver='adam',
```

```
    max_iter=2500,
  )
  grid = {'batch_size': [32, 64, 128]}
```

Se inicia el entrenamiento al definir nuestro modelo a partir de una búsqueda de los mejores hiperparámetros para el clasificador utilizando la clase *GridSearchCV*. Posteriormente, se realiza el ajuste de los datos y etiquetas de entrenamiento al modelo. Al finalizar, se muestra en pantalla el tiempo empleado para entrenar el modelo.

```
# Training model
print(
  f"Training model {classifier_model} with vectors obtained from {vec_model}..."
)
stats["execs"][f"exec_{last_exec}"]["train_start"] = time()
print(
  f"\t* Finding the best hyperparameters for model {classifier_model}..."
)
model = GridSearchCV(
  classifier,
  grid,
  refit=True,
  scoring="f1_macro",
  n_jobs=-1,
  verbose=2
)
print("\t\t- Hyperparameters found.")

print(f"\t* Fitting parameters of model {classifier}...", end="\n\t\t- ")
model.fit(train_data, train_labels)
print("\t\t- Parameters fit.")

stats["execs"][f"exec_{last_exec}"]["train_end"] = time()
stats["execs"][f"exec_{last_exec}"]["train_elapsed"] =
  stats["execs"][f"exec_{last_exec}"]["train_end"] -
  stats["execs"][f"exec_{last_exec}"]["train_start"]
print(
  f'\t! Model trained in {stats["execs"][f"exec_{last_exec}"]["train_elapsed"]}
    seconds\n'
)
```

Se evalúa el modelo entrenado con los datos de prueba y se almacenan los resultados en el archivo de ejecución correspondiente.

```
# Testing model
print(
  f"Testing model {classifier_model} with vectors obtained from {vec_model}..."
```

```
)
stats["execs"][f"exec_{last_exec}"]["test_start"] = time()
prediction = model.predict(test_data)
stats["execs"][f"exec_{last_exec}"]["test_end"] = time()
stats["execs"][f"exec_{last_exec}"]["test_elapsed"] =
  stats["execs"][f"exec_{last_exec}"]["test_end"] -
  stats["execs"][f"exec_{last_exec}"]["test_start"]
print(
  f'\t! Model tested in {stats["execs"][f"exec_{last_exec}"]["test_elapsed"]}
    seconds\n'
)

# Saving predictions to file
with open(f"{nth_exec_path}/predictions.csv", "w") as preds_file:
  preds_file.write("sample_id,model_prediction,ground_truth\n")
  for id, pred, g_truth in zip(idx_ts, prediction, test_labels):
    preds_file.write(f"{id},{pred},{g_truth}\n")
```

La matriz de confusión es construida y graficada para su posterior análisis. Asimismo, se realiza el cálculo de algunas métricas de rendimiento, y las estadísticas se guardan en su archivo correspondiente. Por último, se muestra en pantalla un resumen de las estadísticas del clasificador evaluado.

```
# Compute confusion matrix
conf_mat = utils.get_confusion_matrix(
  prediction,
  test_labels,
  len(categories)
)

# Plotting confusion matrix
utils.plot_confusion_matrix(conf_mat, classifier_model, vec_model)

# Compute performance metrics
stats["execs"][f"exec_{last_exec}"]["accuracy"] = np.mean(
  prediction == test_labels
)
stats["execs"][f"exec_{last_exec}"]["f1"] = metrics.f1_score(
  test_labels,
  prediction,
  average="macro"
)

# Saving finish execution date
stats["execs"][f"exec_{last_exec}"]["exec_end"] =
```

```
  datetime.now().strftime("%d/%m/%Y %H:%M:%S")

# Saving json to file
utils.save_dictionary_as_json(stats_path, stats)

# Print stats
print(f"Classifier {classifier_model}:")
print(f"\t* Vectorizing model {vec_model}:")
print(f'\t\t- Loading data elapsed time:
  {stats["execs"][f"exec_{last_exec}"]["data_load_elapsed"]/60} minutes')
print(
  f'\t\t- Dataset shape:
  {(stats["execs"][f"exec_{last_exec}"]["samples"],
  stats["execs"][f"exec_{last_exec}"]["features"])}'
)
print(
  f'\t\t- Training elapsed time:
  {stats["execs"][f"exec_{last_exec}"]["train_elapsed"]/3600} hours'
)
print(
  f'\t\t- Testing elapsed time:
  {stats["execs"][f"exec_{last_exec}"]["test_elapsed"]/3600} hours'
)
print(
  f'\t\t- Accuracy metric:
  {stats["execs"][f"exec_{last_exec}"]["accuracy"] * 100}%'
)
print(f'\t\t- F1 metric: {stats["execs"][f"exec_{last_exec}"]["f1"]}')

print("=" * 100, end = "\n\n")
```