Classification of incident-related images using machine learning

Primary Topic: CV, Secondary Topic: TS
Course: 2023-2A – Group: 11 – Submission Date: 2023-04-16

Panteleimon Manouselis
University of Twente, MSc-Rob
p.manouselis@student.utwente.nl

Bishoy Essam Samir Yassa Gerges
University of Twente, MSc-Rob
bishoy.gerges@student.utwente.nl

ABSTRACT

This project aimed to develop a machine learning framework for incident recognition in images using a subset of the Incidents1M dataset **[1]**. We used a minimum of 400 samples per category from the 12 incident categories. Preprocessing and data augmentation were performed before extracting descriptors from the images using ResNet, a popular deep learning architecture for image classification. K-nearest neighbors (KNN) was used for classification, and k-fold cross-validation was performed to assess the generalization capabilities of the model. The performance of the model was evaluated using metrics such as accuracy, f-score, and precision. The results demonstrate the effectiveness of the proposed framework for incident recognition in images and its potential relevance for relief organizations.

KEYWORDS

Machine learning, computer vision, image recognition, incident recognition, Convolutional Neural Networks, KNN, k-fold cross validation, classification, relief organizations, natural disasters.

1 INTRODUCTION

Incident recognition in images is a crucial task that can provide critical information for relief organizations, particularly during natural disasters or other events that require human intervention. However, the manual processing of such data can be time-consuming and inefficient, especially in cases where the response should be immediate since human lives are risked. The above highlights the need for automatic recognition and classification of incident-related images using machine learning techniques. In recent years, machine learning and computer vision have become increasingly important in the field of incident recognition, as these approaches allow for the automatic processing of large-scale datasets, such as the Incidents1M dataset [1], which contains over 400,000 images describing natural disasters, damage, and incidents in the wild. In this project, we aimed to develop a machine learning framework using a subset of the Incidents1M dataset. Indicatively, four pictures from each class of incidents can be seen in Figure 1.

Our focus in this project was twofold: Firstly, data pre-processing and augmentation were performed before proceeding with the evaluation of the performance of Convolutional Neural Networks models, for incident recognition in images. We also used KNN for classification and k-fold cross validation to assess the generalization capabilities of the models. This report presents our methodology, results, and conclusions, which demonstrate the potential of machine learning for incident recognition in images and its relevance for relief organizations.

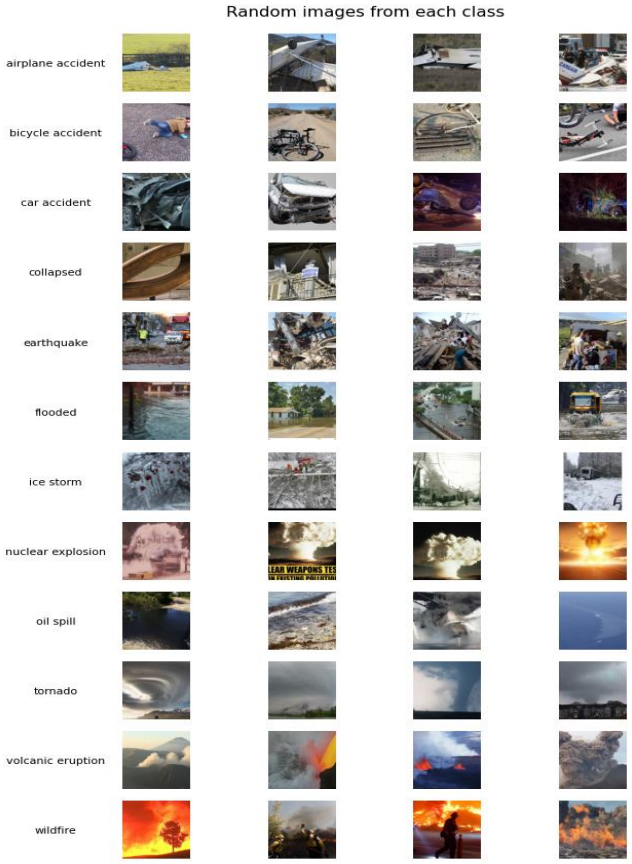

Figure 1 Images from each class of incidents

2 METHODOLOGY

The aim of this study was to develop a machine learning framework for incident recognition in images using a subset of the Incidents1M dataset [1]. In this section, we present a comprehensive overview of our methodology that involves multiple steps, including preprocessing the data, performing data augmentation, extracting descriptors using ResNet, classifying the images via KNN, and evaluating the model's performance with a range of metrics such as precision, recall, and the use of

visualization techniques like confusion matrices. This proposed framework is illustrated in Figure 2.

2.1 Data Preprocessing and Normalization

In the first step of this study, the dataset was preprocessed to ensure its quality and avoid code failure. Initially, we removed all the corrupted images from the dataset to ensure that they do not affect the performance of the models. We performed this step only once since the corrupted images were permanently deleted from the operating system and would not be present in any subsequent runs of the code.

We proceed to normalize our data. Normalization is an important step in preprocessing datasets as it helps to improve the generalization of the model by scaling the pixel values of the images to a common range. In our study, we normalized our dataset using the mean and standard deviation values of the ImageNet dataset, a large dataset of images used for training deep neural networks. This is a common practice in deep learning application since the ImageNet dataset is diverse and covers a wide range of image categories and variations, making it a good reference for normalization values. By using the ImageNet normalization values, we can ensure that our data is normalized consistently with other deep learning applications and models, which helps in achieving better performance and accuracy.

Next, we addressed the issue of low-quality images in the dataset by implementing a function that can detect if an input image is blurry or out of focus. To accomplish this, we first converted the image to a numpy array and then reshaped it into a 3-dimensional array using its height, width, and channels. The array was then normalized to a range of 0 to 255 by subtracting the minimum pixel value of the image and then dividing it by the difference between the maximum and minimum pixel values.

We then converted the RGB image to grayscale and applied the Laplacian filter, which is a 2D spatial derivative operator that measures the local sharpness of the image. The Laplacian filter highlights the edges and other discontinuities in the image, which helps in detecting blurred images. The variance of the Laplacian is then calculated, which is a measure of the local contrast of the image. If the variance is below a threshold value (which was selected through trial and error), the image is considered to be of low quality and is removed from the dataset. For reference one can see one of our removed low-quality images in Figure 3.

By implementing this function, we were able to remove low-quality images from the dataset and create a high-quality dataset suitable for our analysis. This preprocessing step helped us to ensure that the models were trained on high-quality data, which is crucial for achieving accurate results.

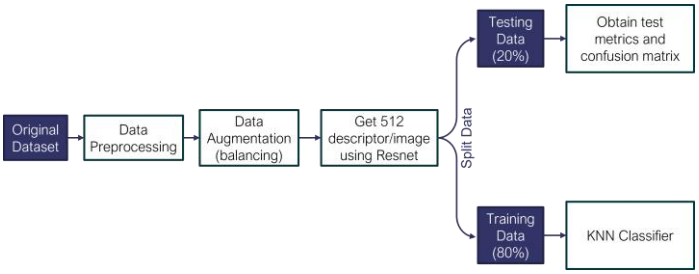

Figure 2: Proposed framework (the stages that data go through)

Figure 3 Low-quality removed image

2.2 Data Augmentation

An unbalanced dataset can lead to poor performance of machine learning models, especially when the dataset is dominated by one or a few classes with many samples, as the models may become biased towards those classes and fail to accurately classify the less represented classes. In contrast, a balanced dataset provides more equal representation of each class, which can improve the performance of machine learning models.

In this study, we noticed that our dataset was unbalanced, with the number of samples per class ranging from 179 to 926 after data preprocessing. The magnitude of the unbalance in our dataset can be seen in Figure 4.

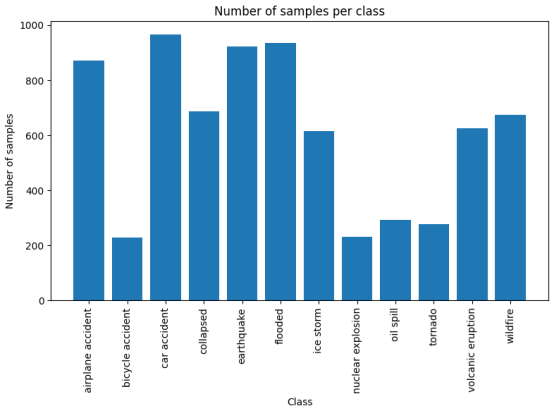

Figure 4 Unbalanced dataset

To tackle the issue of imbalanced classes, we employed data augmentation techniques. More specifically, we increase the number of samples in classes with few instances. We used two transformations, namely Random Horizontal Flip and Random Rotation, to create new samples. Random Horizontal Flip horizontally flips the image with a probability of 0.5, while Random Rotation rotates the image by a random angle between -10 and 10 degrees. By applying these transformations, we create new samples that are different from the original ones and increase the number of samples in underrepresented classes.

To prevent “over-sampling” of our underrepresented classes, we

set a maximum number of samples per class. This maximum is defined by the average value of samples per class in our dataset (569). After data augmentation, we also randomly undersampled our "strong" classes that have a large number of samples. The results of our data augmentation process can be seen in Figure 5.

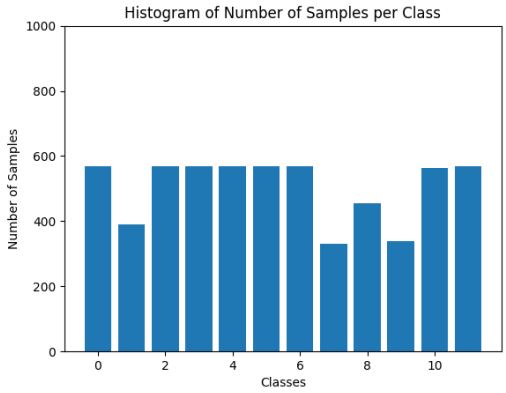

Figure 5: Balanced dataset after data augmentation

This data augmentation allowed us to create a more balanced dataset with a similar number of samples per class, which helped to improve the accuracy and generalization of our machine learning models. By augmenting the data in the described manner, we maintained a relatively constant number of training samples, while simultaneously mitigating the risk of overfitting and ensuring equal representation of all classes during the training phase. This allowed us to improve the performance and generalization ability of our model.

2.3 Descriptor Extraction

In order to classify the preprocessed and augmented images accurately, we extracted descriptors using a ResNet model. ResNet is a deep learning architecture that is widely used for image classification tasks. Its ability to extract high-level features from images has been demonstrated to be effective in accurately classifying them. In this study, we utilized the ResNet18 model, which consists of multiple layers (convolutional layers, pooling layers, and fully connected layers), to extract the descriptors from our images.

To extract the descriptors, we loaded a pre-trained ResNet18, we removed the last fully-connected layer and created a new sequential model using the remaining layers. We set the model to evaluation mode to extract the global features from the images in our balanced dataset. The global features were then stored in a NumPy array. By using ResNet, we were able to obtain meaningful representations of our images that will be used to classify our images.

2.4 Classification

After extracting the descriptors using ResNet18, we utilized the k-nearest neighbors (kNN) algorithm for image classification. kNN is a widely used classification algorithm in machine learning that works by finding the k-nearest neighbors of each test instance in the training set and assigning the class label that is most common among its neighbors. In this study, we used the scikit-learn library to implement the algorithm.

To find the optimal value of k for our dataset, we tested different values of k ranging from 1 to 80. We obtained the accuracy of the kNN classifier for each k value and observed that the accuracy generally decreased with an increase in k (however not monotonically). The highest accuracy of 0.75 was achieved for k=1, which decreased to 0.67 for k=80. This indicates that a smaller value of k was more appropriate for our dataset, as it resulted in better classification accuracy. Overall, the kNN algorithm provided a simple yet effective way to classify our images based on their descriptors.

2.5 Evaluation

To evaluate the performance of the models, we used k-fold cross-validation. Cross-validation is a widely used technique in machine learning that helps to reduce the risk of overfitting. It works by splitting the data into k-folds, training the model on k-1 folds, and testing it on the remaining fold. The cross-validation is performed for each fold, and the results are averaged to estimate the overall performance.

We performed k-fold cross-validation with different numbers of neighbors (kNN) and different numbers of folds (k-folds). To evaluate the performance of the models, we calculated the precision, recall, and confusion matrices for each fold. Precision and recall are metrics that are commonly used to evaluate the performance of classification models. Precision measures the proportion of true positive predictions out of all positive predictions, while recall measures the proportion of true positive predictions out of all actual positive instances (true positives plus false negatives) in the dataset. Confusion matrices help visualize the performance of the models by showing the number of true positives, false positives, true negatives, and false negatives. F1 score is another commonly used evaluation metric in machine learning and image classification. It is a harmonic mean of precision and recall, and it provides a single measure that balances the tradeoff between precision and recall. A high F1 score indicates that the model is performing well both in terms of precision and recall.

After performing k-fold cross-validation, we printed the results for each fold and each number of neighbors. We summarized the results by showing the accuracy of the predictions for each fold and the average accuracy for all folds. We also calculated the standard deviation of the accuracies to provide an estimate of the variation in the results.

Our experiments have shown that the kNN algorithm achieves similar performance across a range of k values from 1 to 30. We observed the highest accuracy within this range, and the variation between the accuracies of different folds was minimized. These results suggest that a relatively small number of neighbors can lead to stable and reliable predictions with the kNN algorithm.

3 RESULTS

In this section, experimental results will be shown for different experiments to test and validate the performance of our algorithm. We performed experiments in to three different situations: using the data without preprocessing, preprocessing the data by

resampling, and preprocessing the data by doing data augmentation.

In each experiment, we ran **k-fold cross validation** and measure the **accuracy**, **precision**, **recall** and **f1-score** metrics in addition to the **confusion matrix**. We ran the k-fold cross validation for different numbers of fold numbers (k) but for simplicity of the report, we will show the results of the 10-fold cross validation only. Accuracy measures the percentage of correctly classified images out of the total number of images in a dataset. Precision measures the ability of the classifier to not label a negative sample as positive. Recall measures the ability of the classifier to find all the positive samples.

For each experiment, the dataset was split into 80% training data and 20% test data. The test data was used to obtain the following metric results and confusion matrices.

3.1 Experiment 1: without data preprocessing

In this experiment, we ran our classification algorithm on the unbalanced dataset shown in Figure 4. That is, balancing or any preprocessing was not done. The metrics results are shown in Table 1 where one can see that an accuracy of about 59% is achieved. The best performing k value for the KNN classifier was k = 10, for which the results are obtained.

Table 1: Experiment 1 results: without data preprocessing

10-fold cross validation Average accuracy	Precision	Recall	F1 score
0.59024 +- 0.02098	0.67	0.75	0.67

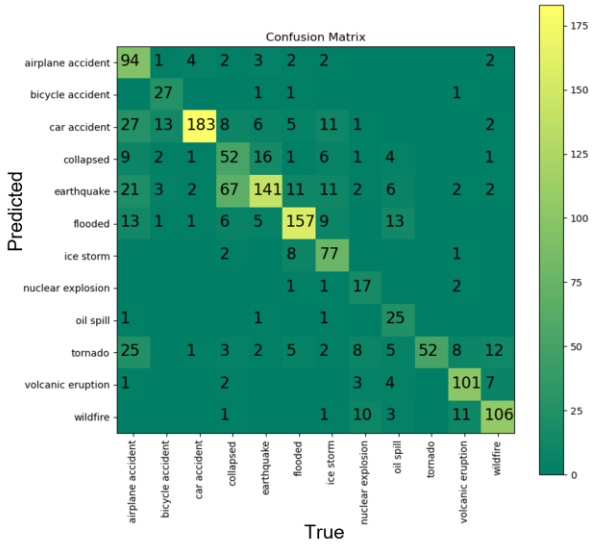

Figure 6: Confusion matrix: without data preprocessing or augmentation

One can observe from the relevant confusion matrix (Figure 6) that in underrepresented classes of our unbalanced dataset, like the 'bicycle accident' class, more than half of the samples get misclassified. However, classes that have abundancy in the number of samples, like the 'car accident' class, do not suffer from such an issue.

These results show that the unbalance of the dataset affects the performance of the classifier specially for those classes suffering from scarcity in the number of samples.

3.2 Experiment 2: after applying data preprocessing and augmentation.

In this experiment, we first balance our dataset to obtain the balanced dataset shown in Figure 5 by firstly doing data preprocessing to remove corrupted or low-quality images followed by data augmentation (details in Methodology section).

Figure 7: examples of correctly and incorrectly classified samples

Table 2, displays the average accuracy of 10-fold cross-validation, which reached 76.5%. Moreover, precision and recall attained 76%. The KNN classifier demonstrated its best performance when k was equal to 1, as indicated by the obtained results. Nonetheless, results obtained for values of k within the range of 6 to 50, yielded very similar outcomes. Figure 7 provides examples of both correctly and incorrectly classified samples.

Table 2: Experiment 2 results: after data preprocessing and augmentation

10-fold cross validation Average accuracy	Precision	Recall	F1 score
0.7648 +- 0.0166	0.76	076	0.75

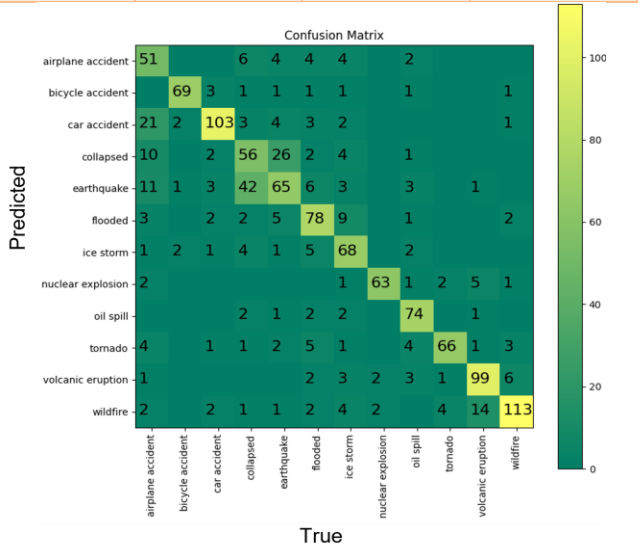

Figure 8: Confusion matrix: with data preprocessing and augmentation (balancing)

The confusion matrix shown in Figure 8 shows number of correctly classified samples (diagonal elements) and the number

of misclassified samples (off diagonal elements) per class (column).

3.3 Discussion and comparison between the results

Here a more detailed discussion is held, highlighting a comparison between the performance of the classification method before and after data balancing and preprocessing.

By comparing Table 1 and Table 2, we can see that the performance has been improved after balancing the dataset and performing data preprocessing. The average accuracy has increased from 59% up to 76.5% which means that the number of misclassifications has decreased. Also, precision, recall and F1 score has increased in case of applying the classifier after doing data balancing (augmentation) and preprocessing.

Another important observation is that by looking at the two confusion matrices in Figure 6 and Figure 8 we can see that the issue of underperformance for classes suffering from scarcity in the number of samples has been resolved. For example, if we look at the class 'nuclear explosion' which has a very low number of samples in the original dataset (around 200), we can see that in the first confusion matrix (Figure 6) more than **59% of the samples get misclassified (25 out of 42).** On the other hand when we look at the second confusion matrix Figure 8, we see that the percentage of misclassification in this class has **decreased to just 6% (4 out of 67),** which shows a big boost in the performance of the classifier for such classes thanks to data augmentation and preprocessing. This because the classifier in this case is not biased towards some (strong) classes over other week ones. Since the dataset is balanced, all classes have the same weight during the classification process by the KNN classifier.

3.3 Dimensionality reduction: PCA visualization

Here, another analysis was done to give more insights about the performance of our algorithm. We applied Principal Component Analysis (PCA) to our image test dataset. PCA works by transforming the high-dimensional dataset into a lower-dimensional space. This is achieved by finding the directions of maximum variance in the data and projecting the data onto these directions. These directions are called principal components and they form a new coordinate system for the data. We used only two principal components to visualize the images in a 2D space. The first principal component corresponds to the direction of maximum variation in the data whereas the second principal component corresponds to the direction of maximum variation orthogonal to the first. The main purpose of this visualization was to explore the relationships between the images and their corresponding classes. To achieve this, we plotted the images as circles with colors corresponding to their predicted class labels as shown in Figure 9.

An interesting observation is that, if we look at the last two classes in the confusion matrix in Figure 8 we see that these classes are well classified by our classifier. This means that they can be easily separated from other classes. This observation is reflected in the PCA visualization. By looking at Figure 9 (these two classes colors are highlighted), we can see that samples of these classes are distributed in the 2D plane relatively far from the other classes. This means that for a classification task, these two classes are likely to be classified correctly and less likely to be confused with other classes resulting in a misclassification.

We should note, however, that PCA analysis is not reliable in our case since PCA captures 'linear' variations while our data (images) are not linear. It was shown though that still some interesting observations can be obtained from the PCA visualization.

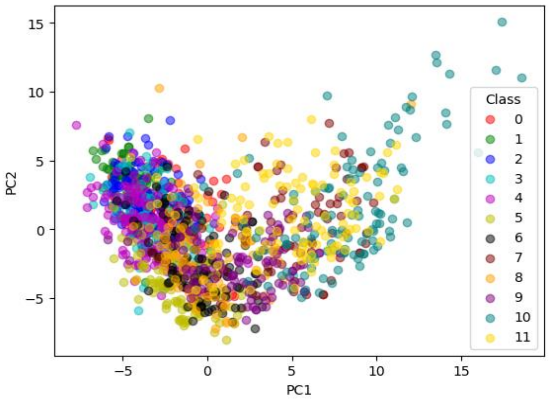

Figure 9: PCA applied on the test data. The last two class colors are highlighted. Predicted labels are used in coloring.

4 CONCLUSIONS

In conclusion, several important findings emerged from this project:

- Data balancing and preprocessing were shown to enhance greatly the performance of the classifier which confirms that this step is crucial before doing data operations.
- ResNet provided a means to effectively describe the dataset using low number of descriptors compared to the total number of image pixels.
- Despite the existence of other more sophisticated methods, KNN is a machine learning algorithm that has demonstrated satisfactory classification performance in our case.
- Using k-fold cross validation and accuracy, precision, recall and f1 score metrics provided a way to assess the performance of our algorithm together with using the confusion matrix.
- PCA visualization in feature space can give some visualization insights, yet not reliable in case the data are nonlinear.

References

[1] E. Weber, D. P. Papadopoulos, A. Lapedriza, F. Ofli, M. Imran and A. Torralba, "Incidents1M: a large-scale dataset of images with natural disasters, damage, and incidents," arXiv, 2022.

APPENDIX

A.1 Python main script

```
# The code is relatively heavy. Depending on your CPU it will need anything from 20 minutes to 1 hour
# Change lines 37, 38 to your own paths (path where dataset is stored, and where the produced figures will be
stored)
# Load all the needed packages for this project
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import torchvision
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from scipy.spatial.distance import cdist
import torch
import torchvision.transforms as transforms
import warnings
import cv2
import os
from sklearn.model_selection import KFold
from sklearn.utils import shuffle
from tqdm import tqdm
from PIL import Image
from statistics import mean
warnings.filterwarnings('ignore')

import torchvision.datasets as datasets
import torch.utils.data as data
import torch.nn as nn
import torchvision.models as models
import torchvision.transforms as transforms
from torch.utils.data import ConcatDataset
import random
from collections import defaultdict
#matplotlib.use('TkAgg')

## Read data
# Define the path to dataset
data_path = "C:/Users/s3084493/OneDrive - University of Twente/MSc Robotics/Quarter 3/Data
Science/Projects/CV_project/Datasets/Incidents-subset"
save_figure_path = "C:/Users/s3084493/OneDrive - University of Twente/MSc Robotics/Quarter 3/Data
Science/Projects/CV_project/Images"


## Remove corrupted images
# Get a list of all files in the folder
files = os.listdir(data_path)

# Loop over all folders in the directory
for folder_name in tqdm(os.listdir(data_path)):
    folder_path = os.path.join(data_path, folder_name)
    # Loop over all files in the folder
    for file_name in tqdm(os.listdir(folder_path)):
        file_path = os.path.join(folder_path, file_name)
        try:
            # Attempt to open the image file
            with Image.open(file_path) as img:
                pass
        except Exception:
            # If the file is not a valid image, delete it
            os.remove(file_path)



# Define the transformations to be applied to each image
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
```

```
    # The values [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225] are the mean and standard deviation of the
ImageNet dataset,
    # which is a large dataset of images used for training deep neural networks.

    # These values are commonly used for normalization when working with pre-trained models that were trained
    # on the ImageNet dataset. By normalizing your input images using these values, you can ensure that the data
has
    # a similar distribution to the data that the pre-trained model was trained on, which can help improve
performance.
])

# Load the dataset using ImageFolder (The variable 'transform' encapsulates the needed transformations of our
data)
dataset = datasets.ImageFolder(root=data_path, transform=transform)
removed_corrupted_count = 7345-len(dataset.samples)
print(f"Removed {removed_corrupted_count} corrupted images from the dataset.")



## Clean dataset
# function that determines whether an image is of low quality
def is_low_quality(image, threshold=0.8):
    if image is None:
        return True
    # calculate the image blur
    image = image.numpy()
    image = np.transpose(image, (1, 2, 0))
    image = ((image - image.min()) / (image.max()-image.min()))*255
    image = image.astype(np.uint8)
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    gray = cv2.convertScaleAbs(gray)
    fm = cv2.Laplacian(gray, cv2.CV_64F).var()
    if fm < threshold:
        return True
    return False

# Iterate through the dataset and remove low-quality images:
# set the threshold for image quality
threshold = 150

for i, (image, target) in enumerate(dataset):
    if is_low_quality(image, threshold):
        # show the image before deleting
        #plt.imshow(image.permute(1, 2, 0))
        #plt.savefig(save_figure_path + "/Deleted_image_example_" + str(i)+".png")
        # remove image
        del dataset.samples[i]
        del dataset.targets[i]

print(f"Removed {7345-len(dataset.samples)-removed_corrupted_count} low-quality images from the dataset.")



# Print the number of classes in the dataset
print("Number of classes:", len(dataset.classes))

## Number of Samples per class
# Create a dictionary to map the category index to its name
category_names = dataset.class_to_idx
# invert key and value of the dictionary
category_names = {v: k for k, v in category_names.items()}
print("Each index and the corresponding category: ", category_names)
# Get a list categorizing each image in the dataset
category_of_each_image = np.array(dataset.targets) # list of 50000 integers
# Calculate the number of samples per category.
sample_per_categoryidx = np.bincount(category_of_each_image)
# Print the number of samples per category.
print("Number of samples per category: ", sample_per_categoryidx)
# Check whether dataset is balanced

fig = plt.figure(figsize=(8, 6))  # Set the figure size to 8x6 inches
ax = fig.add_subplot(111)
```

```
# Plot the number of samples per category using a bar plot.
ax.bar(dataset.classes, sample_per_categoryidx)

# Add axis labels and title.
ax.set_xlabel("Class")
ax.set_ylabel("Number of samples")
ax.set_xticklabels(dataset.classes, rotation=90)  # Set the x-tick labels with rotation.
ax.set_title("Number of samples per class")

# Adjust the margins to make sure the labels fit inside the figure.
plt.tight_layout()

plt.ion()
plt.show()

# Save the plot in the data path with the desired file name.
plt.savefig(save_figure_path + "/Unbalanced_Dataset.png")

## Vizualize your data
# Create a list to store images for each category
category_images = [[] for _ in range(12)]
# Split the dataset images into 12 lists, one for each category.
for (image, category) in dataset:
    category_images[category].append(image)

# Create a 12x4 grid of subplots
fig, axes = plt.subplots(12, 4, figsize=(7, 12))
fig.suptitle("Random images from each class")
# Iterate over each category
for i, images in enumerate(category_images):
    # Set the title of the first subplot in the row to the category name
    axes[i, 0].set_title(category_names[i], x=-1, y=0.3, fontsize=8)
    # Iterate over each of the four subplots in the row
    for j in range(4):
        # Select a random image from the category's list of images
        image = images[np.random.randint(0, len(images))]
        # Show the image in the subplot and turn off axis labels
        image = image.permute(1, 2, 0)
        axes[i, j].imshow((image - image.min()) / (image.max() - image.min()))
        axes[i, j].set_axis_off()
# Show the plot
plt.show()
plt.savefig(save_figure_path + "/Initial_Dataset.png")




# DATA AUGMENTATION PERFORMED TO SAMPLES OF WEAK CLASSES.
# Define the transformations to be applied to each image
data_transforms = transforms.Compose([
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        #transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1),
    ])

# Compute the number of samples in each class
class_counts = {}
for _, class_label in dataset:
    class_counts[class_label] = class_counts.get(class_label, 0) + 1

# Compute the mean number of samples per class
mean_class_count = int(mean(class_counts.values()))

# Create new samples for the classes that have few samples
new_samples = []
for class_label, class_count in class_counts.items():
```

```
    if class_count < mean_class_count:
        # Compute the number of new samples to create
        num_new_samples = mean_class_count - class_count
        print(num_new_samples)

        # Find the indices of the existing samples in this class
        existing_sample_indices = [i for i, (_, c) in enumerate(dataset.samples) if c == class_label]

        # Randomly select existing samples to use for creating new samples
        selected_sample_indices = np.random.choice(existing_sample_indices, size=num_new_samples, replace=True)
        selected_sample_indices = np.unique(selected_sample_indices)

        # Create new samples by applying data augmentation to the selected samples

        for i in selected_sample_indices:
            image, label = dataset[i]
            image = data_transforms(image)
            image_path = dataset.samples[i][0]
            new_samples.append((image_path, label))


# Add new samples to dataset and update targets attribute
dataset.samples.extend(new_samples)
dataset.targets.extend([s[1] for s in new_samples])


## UNDERSAMPLE STRONG CLASSES to get a balanced dataset
labels = dataset.targets

labels = np.array(dataset.targets)

# Determine the number of samples to keep for each class
unique_labels, counts = np.unique(labels, return_counts=True)
num_classes = len(unique_labels)
num_samples = len(labels)
desired_samples_per_class = mean_class_count
num_samples_to_keep = desired_samples_per_class * num_classes

# Determine which indices to keep
indices_to_keep = []
for label in unique_labels:
    label_indices = np.where(labels == label)[0]
    num_label_samples = len(label_indices)
    if num_label_samples > desired_samples_per_class:
        # Randomly select samples to keep
        keep_indices = np.random.choice(label_indices, desired_samples_per_class, replace=False)
        indices_to_keep.extend(keep_indices)
    else:
        indices_to_keep.extend(label_indices)

balanced_dataset = torch.utils.data.Subset(dataset, indices_to_keep)


class_counts = {}
for _, class_label in balanced_dataset:
    class_counts[class_label] = class_counts.get(class_label, 0) + 1

# dataset[balanced_dataset.indices[0]][1]
sorted_counts = dict(sorted(class_counts.items()))
print("Number of samples per class after Data augmentation and undersampling of Strong classes:", sorted_counts)

plt.figure()
plt.bar(class_counts.keys(), class_counts.values())
plt.xlabel('Classes')
plt.ylabel('Number of Samples')
plt.title('Histogram of Number of Samples per Class')
plt.ylim(0, 1000)  # set the y-axis limit to 1000
plt.show()
plt.savefig(save_figure_path + "/Balanced_Dataset.png")

# Create an empty numpy array of size (len(balanced_dataset)x224x224x4
nump_dataframe = np.zeros((len(balanced_dataset), 224, 224, 4), dtype=np.float32)
```

```
# Iterate through the datasets to populate the numpy array
label_list = []
for i in range(len(balanced_dataset)):
    image, label = balanced_dataset[i]
    label_list.append(label)

    nump_dataframe[i, :, :, :3] = np.array(image.permute(1, 2, 0))
    nump_dataframe[i, :, :, 3] = label


## Get CNN descriptors
# Load a pretrained ResNet model from PyTorch
resnet = models.resnet18(pretrained=True)
# Remove the last fully-connected layer
modules = list(resnet.children())[:-1]
resnet = nn.Sequential(*modules)
# Set the model to evaluation mode
resnet.eval()
# Extract global features from the images in the dataset
resnet_descriptors = np.zeros((len(balanced_dataset), 512))
for i, image in enumerate(nump_dataframe[:, :, :, :3]):
    # Convert the NumPy array to a PyTorch tensor
    image_tensor = torch.from_numpy(image).float()
    # Transpose the tensor to the expected shape [batch_size, channels, height, width]
    image_tensor = image_tensor.permute(2, 0, 1).unsqueeze(0)
    # Feed the image through the model to get the output
    output = resnet(image_tensor)
    # Save the output as the descriptor for the image
    resnet_descriptors[i] = output.squeeze().detach().numpy()

# Add labels to resnet_descriptors
# convert labels to numpy array and reshape
labels_array = np.array(label_list).reshape(-1, 1)

# concatenate the labels array as an additional column to resnet_descriptors
resnet_descriptors_with_labels = np.concatenate((resnet_descriptors, labels_array), axis=1)

## Train

# Metrics
def accuracy_metric(actual, predicted):
    # compute accuracy
    accuracy = accuracy_score(actual, predicted)

    # compute precision, recall, and f-score for each class
    precision, recall, fscore, _ = precision_recall_fscore_support(actual, predicted, average=None)

    # compute macro-average precision, recall, and f-score
    macro_recall, macro_precision, macro_fscore, _ = precision_recall_fscore_support(actual, predicted,
average='macro')

    # return dictionary of evaluation metrics
    return {
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'fscore': fscore,
        'macro_precision': macro_precision,
        'macro_recall': macro_recall,
        'macro_fscore': macro_fscore
    }


# Define the train-test split ratio
train_ratio = 0.8
test_ratio = 1 - train_ratio

# Use your k-NN - play with the value of the parameters to see how the model performs
```

```
kvalue_list = [1, 2, 3, 4, 5, 6, 10, 15, 20, 25, 30, 35, 40, 50, 80]

# create a list to store the accuracies for each k value
accuracy_list = []

# split overall CNN descriptors
cnn_train_data, cnn_test_data = train_test_split(resnet_descriptors_with_labels, train_size=train_ratio)

# Separate the CNN values and labels
train_data = (cnn_train_data[:, :-1], cnn_train_data[:, -1].astype(int))
test_data = (cnn_test_data[:, :-1], cnn_test_data[:, -1].astype(int))

for k in kvalue_list:
     # define the kNN classifier
     knn = KNeighborsClassifier(n_neighbors=k)

     # train the kNN classifier on CNN training data
     knn.fit(train_data[0], train_data[1])

     # predict the labels of CNN test data
     cnn_pred_labels = knn.predict(test_data[0])

     # evaluate the performance of the model using accuracy_metric() function
     cnn_accuracy = accuracy_metric(test_data[1], cnn_pred_labels)
     accuracy_list.append((k, cnn_accuracy['accuracy']))
     print(f"Accuracy of kNN classifier on CNN descriptor with k={k}: {cnn_accuracy['accuracy']:.2f}")

# Best accuracy at k=1 (0.76)

# sort the accuracy list in descending order
accuracy_list = sorted(accuracy_list, key=lambda x: x[1], reverse=True)

# get the top 5 k values with the highest accuracy
top_k_values = [x[0] for x in accuracy_list[:5]]

print(f"The top 5 k values with the highest accuracy: {top_k_values}")

## K-Fold cross validation for different values of k-nearest neighbors

# The kNN that performed best in the previous exercises
k_nn_best = top_k_values

# define the k values to be tested
k_fold_list = [2, 5, 10]

# Separate the CNN values and labels
tuple_cross_val_data = (resnet_descriptors_with_labels[:, :-1], resnet_descriptors_with_labels[:, -1].astype(int))

# shuffle the data
cnn_c_val_descriptor, cnn_c_val_label = shuffle(tuple_cross_val_data[0], tuple_cross_val_data[1])

# perform k-fold cross validation with different number of folds and different number of neighbors
for neighbor_num in k_nn_best:
    for k in k_fold_list:
        kf = KFold(n_splits=k)
        acc_list = []
        for train_idx, test_idx in kf.split(cnn_c_val_descriptor):
            # split the data into training and testing sets
            cnn_train_data = cnn_c_val_descriptor[train_idx]
            cnn_test_data = cnn_c_val_descriptor[test_idx]
            train_labels = cnn_c_val_label[train_idx]
            test_labels = cnn_c_val_label[test_idx]

            # define the kNN classifier
            knn = KNeighborsClassifier(n_neighbors=neighbor_num)

            # train the kNN classifier on CNN training data
            knn.fit(cnn_train_data, train_labels)

            # predict the labels of the testing data using the trained kNN classifier
            cnn_pred_labels_test = knn.predict(cnn_test_data)
```

```
            # calculate the accuracy of the predictions
            acc_met = accuracy_metric(test_labels, cnn_pred_labels_test)
            acc = acc_met['accuracy']
            acc_list.append(acc)

        # summarize the results of k-fold cross validation
        print(f"{k}-fold cross validation with {neighbor_num} number of neighbors:")
        print("Accuracies per fold:", acc_list)
        avg_acc = round(np.mean(acc_list), 5)
        std_list = round(np.std(acc_list), 5)
        print("Average accuracy:", avg_acc, "+-", std_list)
        print("\n")
```

```
## Rerun experiment with best K closest neighbor SHOW metrics and confusion matrix
# split overall CNN descriptors
cnn_train_data, cnn_test_data = train_test_split(resnet_descriptors_with_labels, train_size=train_ratio)

# Separate the CNN values and labels
train_data = (cnn_train_data[:, :-1], cnn_train_data[:, -1].astype(int))
test_data = (cnn_test_data[:, :-1], cnn_test_data[:, -1].astype(int))

# define the kNN classifier
knn = KNeighborsClassifier(n_neighbors=k_nn_best[0])

# train the kNN classifier on CNN training data
knn.fit(train_data[0], train_data[1])

# predict the labels of CNN test data
cnn_pred_labels = knn.predict(test_data[0])

# Create a list to store the correctly classified images and another list to store the incorrectly classified
images
correct_images = []
incorrect_images = []

for i, pred_label in enumerate(cnn_pred_labels):
    if pred_label == test_data[1][i]:
        correct_images.append(test_data[0][i])
    else:
        incorrect_images.append(test_data[0][i])

correct_images = np.array(correct_images)
incorrect_images = np.array(incorrect_images)

## Find matching indices of correct_images and incorrect_images with resnet descriptors in order to find the
correct and incorrect images
# Compute pairwise distances between correct_images and resnet_descriptors
distances = cdist(correct_images, resnet_descriptors)
# Find the indices of the minimum distance for each image in correct_images
matching_correct_indices = distances.argmin(axis=1)

distances = cdist(incorrect_images, resnet_descriptors)
matching_incorrect_indices = distances.argmin(axis=1)

# Store correct and incorrect images in respective lists
correctly_matched_images = []
incorrectly_matched_images = []
for i in matching_correct_indices:
    correctly_matched_images.append(numpy_dataframe[i, :, :, :3])

for i in matching_incorrect_indices:
    incorrectly_matched_images.append(numpy_dataframe[i, :, :, :3])

correctly_matched_images = np.array(correctly_matched_images)
```

```
incorrectly_matched_images = np.array(incorrectly_matched_images)

# Display correctly and incorrectly classified images
num_examples = min(len(correctly_matched_images), len(incorrectly_matched_images), 10) # display up to 10 examples
fig, axs = plt.subplots(2, num_examples, figsize=(15,15))
title_set = False


for i in range(num_examples):
    axs[0][i].imshow((correctly_matched_images[i] - correctly_matched_images[i].min())/
(correctly_matched_images[i].max() - correctly_matched_images[i].min()))
    axs[0][i].axis('off')
    axs[1][i].imshow((incorrectly_matched_images[i] - incorrectly_matched_images[i].min())/
(incorrectly_matched_images[i].max() - incorrectly_matched_images[i].min()))
    axs[1][i].axis('off')
    if not title_set:
        axs[0][num_examples//2].set_title("Correctly classified", fontsize=20)
        axs[1][num_examples//2].set_title("Incorrectly classified", fontsize=20)
        title_set = True
plt.show()
plt.savefig(save_figure_path + "/Correctly and incorrectly classified Images.png")




# Function to show confusion matrix and metrics
def calulate_show_metrics(pred_labels, true_labels, lab=category_names):
    print(classification_report(pred_labels, true_labels,
                                target_names=[l for l in lab.values()]))

    conf_mat = confusion_matrix(pred_labels, true_labels)
    fig = plt.figure(figsize=(10, 10))
    width = np.shape(conf_mat)[1]
    height = np.shape(conf_mat)[0]

    res = plt.imshow(np.array(conf_mat), cmap=plt.cm.summer, interpolation='nearest')
    for i, row in enumerate(conf_mat):
        for j, c in enumerate(row):
            if c > 0:
                plt.text(j - .4, i + .1, c, fontsize=16)
    cb = fig.colorbar(res)
    plt.title('Confusion Matrix')
    _ = plt.xticks(range(len(lab)), [l for l in lab.values()], rotation=90)
    _ = plt.yticks(range(len(lab)), [l for l in lab.values()])

    # Adjust the margins of the plot
    plt.subplots_adjust(left=0.2, bottom=0.2)

print("Calculating Confusion matrix and metrics using K {k_nn_best[0]} closest neighbors (best):")
calulate_show_metrics(pred_labels=cnn_pred_labels, true_labels=test_data[1], lab=category_names)
plt.savefig(save_figure_path + "/Confusion_Matrix_k=1_train=0.8.png")



## PCA Analysis
from sklearn.decomposition import PCA
ratio = 0.8
k = 1
# Separate the CNN values and labels
train_data = (cnn_train_data[:, :-1], cnn_train_data[:, -1].astype(int))
test_data = (cnn_test_data[:, :-1], cnn_test_data[:, -1].astype(int))

# define the kNN classifier
knn = KNeighborsClassifier(n_neighbors=k)

# train the kNN classifier on CNN training data
knn.fit(train_data[0], train_data[1])

# predict the labels of CNN train data
cnn_pred_labels_test = knn.predict(test_data[0])
```

```
# Apply PCA on train CNN descriptor set
pca = PCA(n_components=2)
cnn_test_data_transform = test_data[0].copy()
cnn_pca = pca.fit_transform(cnn_test_data_transform)

# Plot samples using the first 2 principal components
colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k', 'maroon', 'orange', 'purple', 'teal', 'gold']

# create new figure and axis
fig, ax = plt.subplots()

for i in range(12):
    ax.scatter(cnn_pca[cnn_pred_labels_test==i, 0], cnn_pca[cnn_pred_labels_test==i, 1],
               c=colors[i], marker='o', label=i, alpha=0.5)


ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.legend(title='Class')
plt.show()
```