2024

Project work report

Recognizing hand signs using principal component analysis

By Mailin Brandt and Finian Landes

Supervisor: Robyn Steiner-Curtis

Gymnasium Liestal

Major Subject: Applied Mathematics/Physics

Contents

1. Introduction

The aim of our project was to distinguish five different hand signs using machine learning. At the beginning of the semester, we had to decide if we wanted to do our project in physics and rebuild a historical physics experiment or if we wanted to learn more about machine learning in applied mathematics. For Finian the decision on machine learning was clear from the beginning on, because he is interested in computer science and could imagine studying a similar topic. For Mailin the decision was harder. She was interested in both topics, but in the end, she decided to improve her programming skills. We started in class to program a principal component analysis (PCA) algorithm from scratch. This included multiple theory inputs and worksheets where we could step by step learn the functionality of the algorithm.

We then used this basic algorithm as a starting point for our own project and optimized it for our needs.

The idea for distinguishing different hand sings was set very fast. We did a short brainstorming session with all the ideas which we had in mind. We had a lot of different ideas but soon concluded that we find the topic of hand recognition the most interesting. Different ideas for working with hands came into our minds. To distinguish different hands was for us a bit boring so we thought of other ways and emojis came into the discussion. We thought about comparing our hands to emojis, but we turned down this idea. The emojis were then used as an inspiration for our decision on the different hand signs. Finally, we decided on five different hand signs.

Throughout the project we created our own data set and optimized our code.

Because we implemented an efficient way of creating our own data set, we had enough time and motivation to write an additional algorithm as comparator. After some evaluation we decided to implement the k-means clustering algorithm. We got a short theory input by Mrs. Steiner for this algorithm and then used those ideas to program it ourselves from scratch.

2. “Leitfrage”

We formulated our “Leitfrage” right at the beginning of programming our algorithm. After we decided on our topic, the questions of whether the algorithm could distinguish different hand signs of the same hand came up. Our final “Leitfrage” was „How can we distinguish

different hand signs using PCA and the same hand? “ After working on the algorithm, we saw that the algorithm can work on the same hand and decided to include more hands and tested it on an unknown hand. Due to a fast implementation of the PCA algorithm we extended the idea behind our “Leitfrage” and included a second algorithm, the K-Means clustering algorithm. Throughout the project we were able to answer the “Leitfrage” but also extend it as described above.

3. Theory and Results

3.1 Dataset

For our program we needed to create a dataset containing our five hand signs, shown by different hands. To create this in an efficient manner we built a camera rig from wooden boards, shown below:

Figure 1: Camera Rig used to take pictures of the hand signs

We then ensured that the light was the same for each person. Using a Leica V-Lux[1] (Type 114) Camera we then took squared images. This would simplify the process later when processing the images. Furthermore, our camera rig was set up exactly to fit one hand into the frame so we wouldn’t need to crop our images further. Worth mentioning here is, that the camera was really an overkill, as we rescaled the images drastically. So, a phone would have been sufficient as well, but some kind of tripod would still be recommended to guarantee standardized pictures.

To automate the process, we wrote a program which automatically named the images, so all the image names contained the characteristics of the hand sign. Our program also rescaled

[1] Leica V-Lux: https://leica-camera.com/de-CH/fotografie/kameras/v-lux/v-lux-5-schwarz (20.12.24)

the images to 256x256 pixels and converted them to grayscales. For our PCA and k-means clustering program to work, the images were read row by row to form vectors. All of this was done using os[2] and Pillow[3].

We chose our five hand signs arbitrarily but ensured that they were quite different from each other so recognition would be easier. We decided on using Easy, Metal, Peace, Thumbs up and Pistol as our standardized signs.

Figure 2: Processed images of the five hand signs. From left to right: Easy, Metal, Peace, Thumbs up, Pistol

3.2 Principal component analysis (PCA)

Principal component analysis (PCA) is an algorithm used to simplify complex datasets. It identifies the most important patterns and by doing so it helps to reduce complexity but maintain the essential information.[4]

3.2.1 Theory

PCA works by transforming the original dataset into a new coordinate where the axes (principal components) capture the maximum variance, the difference between a point and the mean. To do this, the data is first normalized by subtracting the mean of all points from each point. Using this data a covariance matrix is calculated. This matrix tells us whether the change in one feature is associated with the change in another feature. In our case this matrix

Figure 3: The first five eigenfaces

[2] Python Documentation, Art. “os”: https://docs.python.org/3/library/os.html (02.12.24)
[3] Jeffrey A. Clark and contributors, Pillow Documentation: https://pillow.readthedocs.io/en/stable/ (02.12.24)
[4] Wikipedia, Art. “Principal component analysis”: https://en.wikipedia.org/wiki/Principal_component_analysis (25.11.2024)

would tell us how the brightness of one pixel is related to the brightness of another pixel in another image. For example, the center of the hand which is in the same place for each hand sign might have a high covariance and the fingers which differ in all pictures have a low covariance. Using this matrix, we calculate the eigenvectors, which are called eigenfaces or eigenhands in our specific case. These eigenvectors are the directions of the axis which show the most important features of the data, ordered from the most relevant to the least relevant. This ordering allows us to disregard the last eigenvectors without losing important data.
To recognize the hand signs in the test dataset, we compared each test image, represented as a vector, to the images in the training set. We identified the closest match in the training set, and the test image was classified as showing the same hand sign as its closest training image.

3.2.2 Results

To prove our concept, we first created a small dataset only consisting of three pictures of three different hand signs of Mailin's hand. We tried our algorithm on these pictures with a random train and test set. The test set contained three randomly chosen pictures, which were then excluded from the train set. We then saw that our algorithm worked. We had a success rate between 66 and 100 percent, which was higher than just guessing. Therefore, we were able to conclude, that our algorithm worked with distinguishing different hand signs.

We then created a full dataset containing 150 pictures of Mailin's hand and decided on ten different signs. With this data set and a randomly created train and test set we came to a success rate around 90 percent. All those 150 pictures were then used as a train set. As a test set, we did three pictures per sign of Finian's hand. This resulted in a dramatic drop of the success rate down to about 50 percent.

We concluded that our dataset was not diverse enough with only Mailin's hand and therefore added more people's hands. To be more efficient we reduced the number of signs to five and only did ten pictures per sign. At the end we had four times ten pictures per sign and once 15 pictures per sign and therefore overall 275 pictures in our train set. When we used Mailin's and another person's hands as train set the success rate to recognize an unknown hand 's sign rose to 93 percent and with a third hand to 100 percent. Adding even more people 's pictures did not change the success rate any further.

Overall, we therefore concluded, that our principal component analysis algorithm worked perfectly with recognizing five different hand signs on an unknown hand.

When trying to further optimize our PCA algorithm, we investigated using only the most significant eigenfaces, which means the *n* first ones. Using only 50 instead of 275 we did not have any decrease in our success rate. So, with only around 20 % of the eigenfaces we still had a perfect score, this would make the algorithm way more efficient. With this small dataset, the efficiency is not really an issue, but it would matter, if we were to drastically increase our dataset size to a point where performance played a role.

3.3 K-Means clustering

K-Means clustering is, as the name implies, a clustering algorithm. It separates a dataset into the most compact *k* different clusters. Important to note here is, that K-means is nondeterministic, which means that for different starting conditions or so-called different initial centroids, the algorithm produces different clusters.

3.3.1 Theory

The Algorithm works by first choosing *k* centers, called centroids. For the clusters, these can be chosen arbitrarily or with some kind of criteria. For simplicity's sake we chose k points from our dataset to be our initial centroids. The algorithm then goes over each point and assigns it to the closest centroid. Then the centroids are updated by taking the mean of the points assigned to the old centroid. This gets repeated until some kind of converging criteria is fulfilled. In our case this criterion was, that the sum of all distances between the old and new centroids was close to zero.[5]

Figure 4: Five arbitrarily chosen centroids when dividing our dataset into 15 clusters

To score our k-means we built a function, which takes into account the number of images per sign per cluster. Then we calculated a score over all clusters, where a score of 100% would

[5] Wikipedia, Art. "K-means clustering": https://en.wikipedia.org/wiki/K-means_clustering (25.11.2024)

mean all clusters only contain one sign and the lower the score the more mixed the clusters are.

3.3.2 Results

Even though the results of the PCA were already perfect we went on to create a clustering algorithm. This algorithm was never intended to be used to classify the images but more to analyze patterns in similar pictures, classified as “similar” by the computer. To make the algorithm run faster, we used the already PCA-optimized data as points. At first our results were insufficient, so the signs were distributed evenly in each cluster. As we found out later, this was linked to our initial centroids, which had been random points in space. After modifying the initialization to the way mentioned above, choosing random points from the dataset, we got better results. Using five clusters the results were still quite mixed up, so the clusters did not make any sense for us. Increasing the number of clusters made the clusters better and then we found out that 15 to 20 clusters worked best for our dataset. Looking at these clusters many things became clearer for us. The algorithm made quite pure clusters for thumbs up and metal which makes sense as the signs are rather different from the others. Peace and Pistol on the other hand got clustered together often, from this we concluded, that the thumb is one of the more relevant features for the algorithm. Whether the index finger or pinky was shown, seemed to be less relevant. Shown below are three randomly chosen clusters from a run with a high score, dividing our data into 15 clusters.

In cluster one the main clustering criteria seems to be two fingers pointed out next to each other, the position of the hidden fingers does not seem to be well recognizable for the computer.

Figure 5: Cluster one

In cluster two you can see the phenomenon mentioned above, as it seems that the thumb is the important factor to order as well as the position of the second finger, seen in the last image, thumbs up, and in the two easy in the first row.

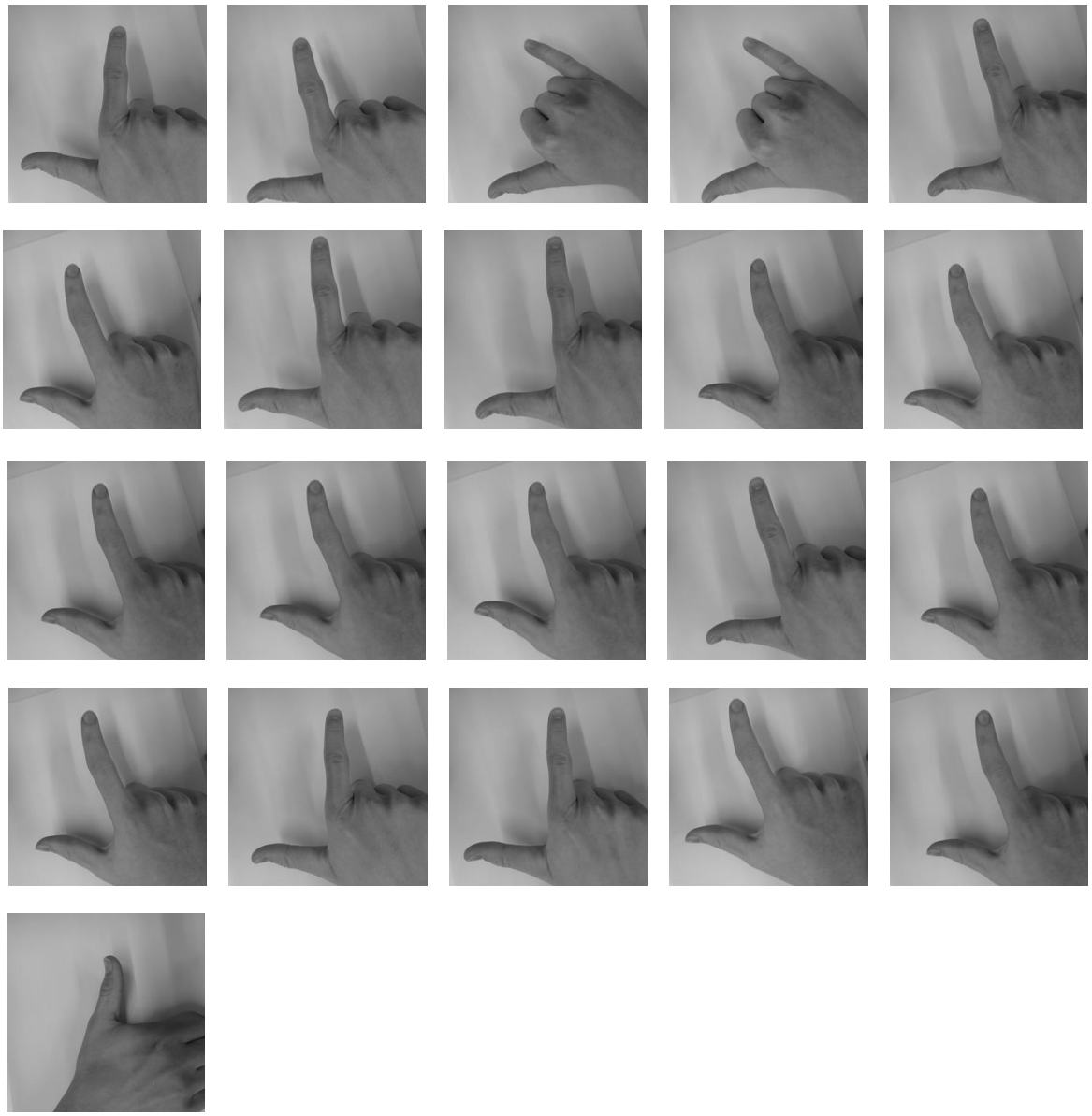

Figure 6: Cluster two

In cluster three the K-means algorithm seems to ignore the pinky which might be because the pinky is small, or the index finger and thumb are so close together that this information outweighs the pinky being there as well.

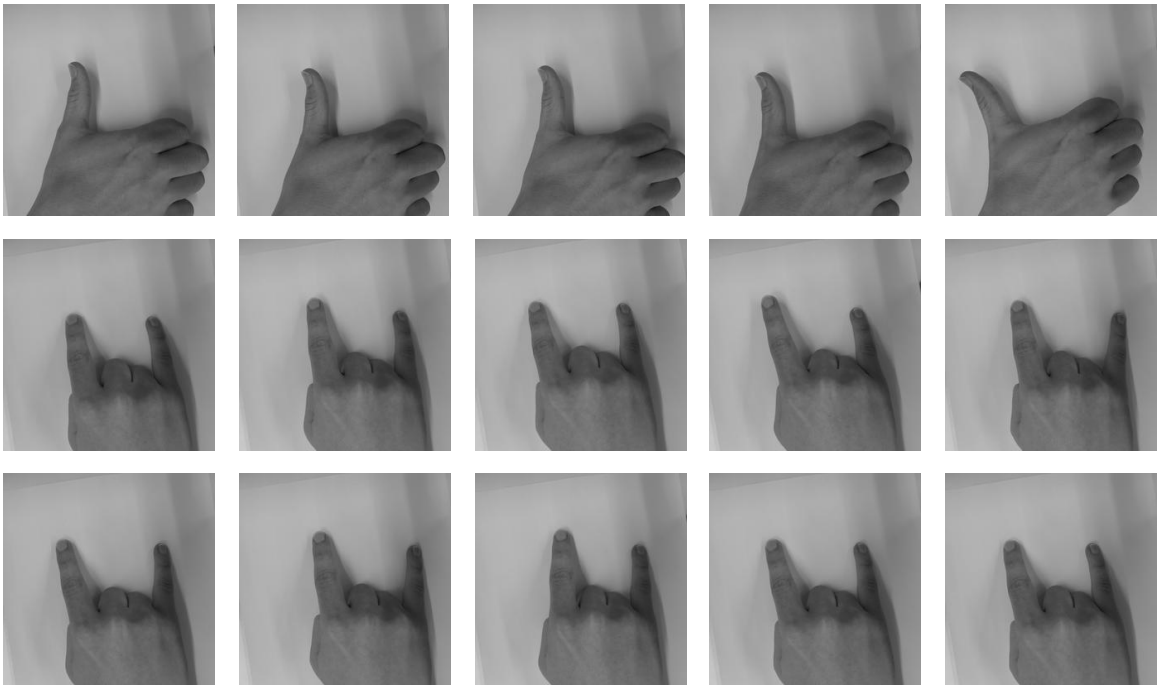

Figure 7: Cluster three

3.4 Reflection

Our project overall worked as planned. We had a nice and productive working environment. We sometimes struggled a bit with our code but managed to solve it quite fast. We are very pleased with the results of our PCA Algorithm. We did not expect it to work that well after such a short amount of time. The program to name and optimize our pictures saved us a lot of time and we would definitely recommend doing it this way again.

We are well aware of the fact that our dataset is not very diverse as it only contains white-colored hands. We do not know, if our programs also work on darker-colored hands, but would assume, that it might not work there, based on other studies presented to us by Mrs. Steiner. If we would have had more time on our project, we would have increased the diversity in our dataset with different colored hands. This would also apply to our test set. We only tested our algorithm with a person with the same age as the others in the train set. It would also be interesting to know, whether an older and/or younger person's hand would influence our results and if so by how much and why.

At the beginning of working with the K-means clustering algorithm, we were unsure, if the algorithm works at all, because we had inadequate results. This resulted in a rewriting of the algorithm in an empty file with two-dimensional data, so it would be plottable. By doing so, we found out, that our initialization process was deficient, but that our general algorithm worked. In the end the issue was with the data coming from the PCA algorithm as we used it unconverted to the right format.

The results of K-Means clustering were good, but not very satisfactory. Other clustering methods might result in a better outcome. The issue with our dataset was, that overlapping features in hand signs, such as the presence or position of a finger, sometimes seemed to be more important than other differences (e.g. whether the pinky is shown or not). Another way we could cluster our data would be by not assigning items to a single cluster. Rather we would assign points to all clusters, with a weight, which tells us how much the point belongs to the cluster. This would result in a so-called fuzzy clustering.[6] In our case we could then extend our K-Means clustering to fuzzy C-Means[7] clustering. This might result in a better performance for our use case.

[6] Wikipedia, Art. "Fuzzy Clustering": https://en.wikipedia.org/wiki/Fuzzy_clustering (13.12.2024)
[7] Medium, Art. " Fuzzy C-Means Clustering (FCM) Algorithm": https://medium.com/geekculture/fuzzy-c-means-clustering-fcm-algorithm-in-machine-learning-c2e51e586fff (13.12.2024)

4. References

4.1 List of references

Leica V-Lux:

https://leica-camera.com/de-CH/fotografie/kameras/v-lux/v-lux-5-schwarz (20.12.24)

Python Documentation, Art. “os”:

https://docs.python.org/3/library/os.html (02.12.24)

Jeffrey A. Clark and contributors, Pillow Documentation:

https://pillow.readthedocs.io/en/stable/ (02.12.24)

Wikipedia, Art. “Principal component analysis”:

https://en.wikipedia.org/wiki/Principal_component_analysis (25.11.2024)

Wikipedia, Art. “K-means clustering”:

https://en.wikipedia.org/wiki/K-means_clustering (25.11.2024)

Wikipedia, Art. “Fuzzy Clustering”:

https://en.wikipedia.org/wiki/Fuzzy_clustering (13.12.2024)

Medium, Art. “ Fuzzy C-Means Clustering (FCM) Algorithm”:

https://medium.com/geekculture/fuzzy-c-means-clustering-fcm-algorithm-in-machine-learning-c2e51e586fff (13.12.2024)

4.1 List of illustrations

5. Acknowledgements

First and foremost, we want to thank Mrs. Steiner-Curtis for her support and guidance in this project. Furthermore, we want to thank everyone from our class willing to let us photograph their hand for our dataset.

Finally, we want to thank our families for reviewing our text and providing us with a camera for creating our dataset.

6. Appendix

6.1 Image processing

```
1.  import os
2.  from PIL import Image, ImageOps
3.
4.  #Saves an image with a provided name at a given location
5.  def rename_save_img(img: Image, name: str, path: str) -> None:
6.      img.save(path + "//" + name + ".jpg")
7.
8.  #Rotates an image with a given angle
9.  #Returns the rotated image
10. def rotate(img: Image, alpha: int) -> Image:
11.     img = img.rotate(angle=alpha)
12.     return img
13.
14. #Scales down the image to a given width and height, only works on square
    pictures (Warning shown with LANCZOS method is irrelevant)
15. #Returns new image
16. def downscale(img: Image, size: int) -> Image:
17.     new_size: list = (size, size)
18.     return img.resize(new_size, Image.LANCZOS)
19.
20. #Converts image to Grayscale
21. #Returns new image
22. def black_white(img: Image) -> Image:
23.     return img.convert('L')
24.
25. #Mirrors an image
26. #Returns mirrored Image
27. def mirror(img: Image) -> Image:
28.     return ImageOps.mirror(img)
29.
30. _debug: bool = True
31. image_dir: str = "PCA of hand signs//Images//10_T_E_P_Pi_M_Remy"
32. image_dest: str = "PCA of hand signs//Processed Images//Test"
33.
34. order_names: list = ["thumbs_up", "easy", "peace", "pistol","metal"]#Abbr.
    Metal: M, Peace: P, Pistol: Pi, Easy: E, Thumbs up: T
35. n_per_cat: int = 10 #Images per category
36. replace: bool = False #If False checks whether name already exists, if it
    does it increments the number and adds the new image
37. size: int = 256 #resulting image in format size x size
38. file_names: list = sorted(os.listdir(image_dir))
39.
40. for i in range(len(file_names) // n_per_cat):
```

```
41.     image_num: list = [(i * n_per_cat), (i * n_per_cat) + n_per_cat]
    #Images [0,5)
42.     images: list = file_names[image_num[0]:image_num[1]]
43.     image_name: str = order_names[i]
44.     for j, name in enumerate(images):
45.         img_name: str = image_name + "_" + f"{j:02}"
46.         img: Image = Image.open(image_dir + "//" + name)
47.         img = rotate(img, 180)
48.         img = downscale(img, size)
49.         img = black_white(img)
50.         if not replace:
51.             n: int = j
52.             while True:
53.                 if os.path.isfile(image_dest + "//" + img_name + ".jpg"):
54.                     n += 1
55.                     img_name = image_name + "_" + f"{n:02}"
56.                 else: break
57.         if _debug:
58.             print(str((i * n_per_cat) + j + 1) + "/" +
    str(len(file_names)) +" Saved Image as: " + img_name)
59.         img = rename_save_img(img, img_name , image_dest)
```

6.2 PCA

```
1.  import numpy as np
2.  from PIL import Image
3.  import os
4.  import matplotlib.pyplot as plt
5.
6.  #Loads Images from dir, size has to be provided (image in format size x
    size)
7.  #returns images as vectors and image names as list
8.  def load_images(dir: str, size: int = 256, grayscale: bool = True) ->
    np.ndarray:
9.      image_names: list = sorted(os.listdir(dir))
10.     n: int = len(image_names)
11.     images: np.ndarray = np.zeros([n, size ** 2])
12.     for i, name in enumerate(image_names):
13.         img: Image = Image.open(dir + "//"+ name)
14.         images[i] = img.getdata(0)
15.     return images, np.array(image_names)
16.
17. #Generates a random train and test set with n_test test items
18. #returns train and test images and train and test names
19. def get_train_test_dataset(images: np.ndarray, image_names: list[str],
    n_test: int) -> tuple[np.ndarray, np.ndarray, list, list]:
20.     indices: list = np.random.choice(len(images), n_test, replace = False)
```

```
    test: np.ndarray = images[indices]
    names_test: list[str] = image_names[indices]
    train: np.ndarray = np.delete(images, indices, axis = 0)
    names_train: list[str] = np.delete(image_names, indices)
    return train, test, names_train, names_test

#Using the coeff_matrix, the eigenfaces and the meanface returns the index of the image which lies closest to point
#Returns index of closest image
def closest_neighbour(point: np.ndarray, meanface: np.ndarray , eigenfaces: np.ndarray, coeff_mat: np.ndarray) -> int:
    c1: np.ndarray = np.matmul(eigenfaces, point - meanface)
    distances: np.ndarray = np.sum((c1 - coeff_mat.T)**2, axis=1)
    return np.argmin(distances)

#Creates a matrix from the meanface, the eigenfaces and the passed points, which is used to increase efficiency of the PCA-Algorithm
#Returns a matrix in form n x n where n is the number of images
def coeff_matrix(meanface: np.ndarray, eigenfaces: np.ndarray, points: np.ndarray) -> np.ndarray:
    return np.matmul(eigenfaces, (points - meanface).T)

#Saves an image with a given filename stable for image in any format
#Returns None
def save_image(picture: np.ndarray, filename: str, location: str = "..\Processed Images\Eigenfaces", size: int = 256) -> None:
    min_val: float = np.min(picture)
    picture = picture - min_val
    picture = picture / np.max(picture)
    picture = (255 * picture).astype(np.uint8)
    n_picture = np.reshape(picture, (size, size)).astype(dtype = np.uint8)
    plt.imshow(n_picture, cmap = "gray")
    plt.show()
    im = Image.fromarray(n_picture, mode="L")
    im.save(location + "/" + filename + ".jpg")

#Gets the successes of a given test set
#Returns the percentage of correct predictions
def get_successes(test: np.ndarray, meanface: np.ndarray, eigenfaces: np.ndarray, coeff_mat: np.ndarray, names_train: list, names_test: list) -> float:
    successes: int = 0
    for j in range(len(test)):
        p: np.ndarray = test[j]
        i: int = closest_neighbour(p, meanface, eigenfaces, coeff_mat)
        pred_name: str = names_train[i][:-7] #Removes image name suffix _XX.jpg
        real_name: str = names_test[j][:-7]
```

```
61.         successes += (real_name == pred_name)
62.     return successes / len(test)
63.
64. image_dir_train: str = "..\Processed Images\Train"
65. image_dir_test: str = "..\Processed Images\Test"
66. n_test_images: int = 50
67. train, names_train = load_images(image_dir_train)
68. test, names_test = load_images(image_dir_test)
69.
70. #Comment out the upper test set, and get some test pictures from the train
    data
71. #train, test, names_train, names_test = get_train_test_dataset(train,
    names_train, n_test_images)
72. meanface: np.ndarray = np.mean(train, axis=0)
73. A: np.ndarray = train - meanface
74. eigenfaces, s, VT = np.linalg.svd(A.transpose(), full_matrices=False)
75. eigenfaces: np.ndarray = eigenfaces.transpose()
76. eigenfaces = eigenfaces[:50] #Use only the N first eigenfaces
77. coeff_mat: np.ndarray = coeff_matrix(meanface, eigenfaces, train)
78. s: float =  get_successes(test, meanface, eigenfaces, coeff_mat,
    names_train, names_test)
79. print ("Success rate: "+str(np.round(s * 100, 2)) +"%")
```

6.3 K-Means Test Program

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.
4.  def random_colors(k):
5.      np.random.seed(50)
6.      colors = []
7.      while len(colors) < k:
8.          r, g, b = np.random.randint(0, 255), np.random.randint(0, 255),
    np.random.randint(0, 255)
9.          if not (g > r * 1.2 and g > b * 1.2):
10.             colors.append((r, g, b))
11.     return np.array(colors) / 255.0
12.
13. def visualize(centroids, points, classification = [], k = 0, title = "") -
    >None:
14.     plt.title(title)
15.     if len(classification) == 0:
16.         plt.scatter(centroids[:, 0], centroids[:, 1], c='green',
    marker='o')
17.         plt.scatter(points[:, 0], points[:, 1], c='blue', marker='o')
18.     else:
19.         colors = random_colors(k)
```

```
        for i, point in enumerate(points):
            plt.scatter(point[0], point[1], color=colors[classification[i]], marker='o')
        plt.scatter(centroids[:, 0], centroids[:, 1], c='green', marker='o')
    plt.show()

def init_centroids(k: int, points: np.ndarray) -> np.ndarray:
    n: int = len(points) // k
    indicies: np.ndarray = np.arange(len(points))
    np.random.shuffle(indicies)
    init_centroids: np.ndarray = np.array([np.mean(points[indicies[n * i:n *(i + 1)]], axis = 0) for i in range(0, k)])
    return init_centroids

def k_means(points, centroids, k):
    classification = []
    new_centroids: np.ndarray = np.zeros((k, points.shape[1]))
    clusters: list = [[] for _ in range(k)]
    for point in points:
        distances = [np.linalg.norm(point- centroid) for centroid in centroids]
        classification.append(np.argmin(distances))
        clusters[classification[-1]].append(point)

    #visualize(centroids, points, classification,k, title = "New Clusters")
    for i in range(len(centroids)):
        if len(clusters[i]) == 0:
            new_centroids[i] = points[np.random.randint(0, points.shape[0])]
        else:
            new_centroids[i] = np.mean(clusters[i], axis = 0)
    #visualize(new_centroids, points, classification,k, title = "Moved Centoids")
    return classification, new_centroids

def is_not_converged(last_centroids: np.ndarray, current_centroids: np.ndarray, e: float = 1e-20) -> bool:
    dist: float = 0
    for  i in range(len(last_centroids)):
        dist += np.sqrt(np.sum((last_centroids[i] - current_centroids[i]) ** 2))
    print(dist)
    return dist > e

def outlying_distances(points: np.ndarray) -> list:
    mean_val: np.ndarray = np.mean(points, axis = 0)
```

```
    dist: list = [np.linalg.norm(point - mean_val) for point in points]
    return dist

k = 5
n_points_per_cl = 200
clusters = []
n = 20
centroids = np.zeros((k,2))
classification = []

n_clusters = 1
pos = [[n,n], [-n,n], [n,-n], [-n,-n]]

#Ensures that the random numbers are the same every time the algorithm is run
np.random.seed(2104013275)

for i in range(n_clusters):
    clusters.append(np.random.randn(n_points_per_cl, 2) + np.array(pos[i]))
points = np.vstack(clusters)
print(outlying_distances(points))
centroids = init_centroids(k, points)

last_centroids = np.zeros_like(centroids)
while is_not_converged(last_centroids, centroids):
    last_centroids = centroids
    classification, centroids = k_means(points,centroids,k)
```

6.4 PCA, K-Means

```
### Imports
import numpy as np
from PIL import Image
import os
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import shutil

#Loads Images from dir, size has to be provided (image in format size x size)
#returns images as vectors and image names as list
def load_images(dir: str, size: int = 256) -> np.ndarray:
    image_names: list = sorted(os.listdir(dir))
    n: int = len(image_names)
    images: np.ndarray = np.zeros([n, size ** 2])
    for i, name in enumerate(image_names):
```

```
22.         img: Image = Image.open(dir + "//"+ name)
23.         images[i] = img.getdata(0)
24.     return images, np.array(image_names)
25.
26. #Generates a random train and test set with n_test test items
27. #returns train and test images and train and test names
28. def get_train_test_dataset(images: np.ndarray, image_names: list[str],
    n_test: int) -> tuple[np.ndarray, np.ndarray, list, list]:
29.     indices: list = np.random.choice(len(images), n_test, replace = False)
30.     test: np.ndarray = images[indices]
31.     names_test: list[str] = image_names[indices]
32.     train: np.ndarray = np.delete(images, indices, axis = 0)
33.     names_train: list[str] = np.delete(image_names, indices)
34.     return train, test, names_train, names_test
35.
36. #Using the coeff_matrix, the eigenfaces and the meanface returns the index
    of the image which lies closest to point
37. #Returns index of closest image
38. def closest_neighbour(point: np.ndarray, meanface: np.ndarray ,
    eigenfaces: np.ndarray, coeff_mat: np.ndarray) -> int:
39.     c1: np.ndarray = np.matmul(eigenfaces, point - meanface)
40.     distances: np.ndarray = np.sum((c1 - coeff_mat.T)**2, axis=1)
41.     return np.argmin(distances)
42.
43. #Creates a matrix from the meanface, the eigenfaces and the passed points,
    which is used to increase efficiency of the PCA-Algorithm
44. #Returns a matrix in form n x n where n is the number of images
45. def coeff_matrix(meanface: np.ndarray, eigenfaces: np.ndarray, points:
    np.ndarray) -> np.ndarray:
46.     return np.matmul(eigenfaces, (points - meanface).T)
47.
48. #Saves an image with a given filename stable for image in any format eg.
    0-1, 0-255, -100-100
49. #Returns None
50. def save_image(picture: np.ndarray, filename: str, location: str =
    "..\Processed Images\Eigenfaces", size: int = 256) -> None:
51.     min_val: float = np.min(picture)
52.     picture = picture - min_val
53.     picture = picture / np.max(picture)
54.     picture = (255 * picture).astype(np.uint8)
55.     n_picture = np.reshape(picture, (size, size)).astype(dtype = np.uint8)
56.     plt.imshow(n_picture, cmap = "gray")
57.     plt.show()
58.     im = Image.fromarray(n_picture, mode="L")
59.     im.save(location + "/" + filename + ".jpg")
60.
61. #Initializes centroids for K-Means by choosing k random points
62. #Returns k centroids
```

```
63. def init_centroids(k: int, points: np.ndarray) -> np.ndarray:
64.     centroids: np.ndarray = points[np.random.choice(len(points), k,
    replace = False)]
65.     return centroids
66.
67. #Run K-means
68. #Returns new centroids and the classification of the points
69. def run_iteration(points: np.ndarray, centroids: np.ndarray, k: int) ->
    tuple[np.ndarray, list]:
70.     classification: list = []
71.     clusters: list[np.ndarray] = [[] for _ in range(k)]
72.     new_centroids: np.ndarray = np.zeros((k, points.shape[1]))
73.
74.     for point in points:
75.         distances: list[np.ndarray] = [np.linalg.norm(point - centroid)
    for centroid in centroids]
76.         classification.append(np.argmin(distances))
77.         clusters[classification[-1]].append(point)
78.     for i in range(len(centroids)):
79.         if len(clusters[i]) == 0:
80.             new_centroids[i] = points[np.random.randint(0,
    points.shape[0])]
81.         else:
82.             new_centroids[i] = np.mean(clusters[i], axis = 0)
83.     return classification, new_centroids
84.
85. #Checks whether k-means is converged by comparing the distances between
    the old and the new centroids
86. #Returns True when K-means is NOT converged
87. def is_not_converged(last_centroids: np.ndarray, current_centroids:
    np.ndarray, e: float = 1e-20) -> bool:
88.     dist: float = 0
89.     for  i in range(len(last_centroids)):
90.         dist += np.sqrt(np.sum((last_centroids[i] - current_centroids[i])
    ** 2))
91.     return dist > e
92.
93. #Clusters the names of the images using the classification of the points
94. #Returns list with the names of images in each cluster, where each cluster
    is depicted by a sublist
95. def cluster_names(classification: list, names: list, k: int) -> list:
96.     classified_names: list = [[] for _ in range(k)]
97.     for i, name in enumerate(names):
98.         classified_names[classification[i]].append(name)
99.     return classified_names
100.
101.  #Computes coefficients from a point a mean and a point array
102.  #Returns those coefficients as np.array
```

```
103.  def compute_coefficients(point: np.ndarray, points: np.ndarray, mean:
    np.ndarray) -> np.ndarray:
104.      return np.matmul(points, point - mean)
105.
106.  #Calculates the distance between a point and a mean by additionally
    using a points array to make a transformation in space to make the
    euclidean distance more meaningful
107.  #Returns the distance as float
108.  def distance(point: np.ndarray, q: np.ndarray, mean: np.ndarray,
    points: np.ndarray) -> float:
109.      c1: np.ndarray = compute_coefficients(point, points, mean)
110.      c2: np.ndarray = compute_coefficients(q, points, mean)
111.      return np.sqrt(np.sum((c1 - c2)**2))
112.
113.  #Calculates the distance between each point and a given center point
114.  #Returns the distances of each point to the center point
115.  def outlying_distances(center: np.ndarray, points: np.ndarray) -> list:
116.      dist: list = [distance(point, np.zeros_like(point), center, points)
    for point in points]
117.      return dist
118.
119.  #Sorts points by taking their distance to a given center and then
    ordering them in ascending order
120.  #Returns the Distances and the sorted points aswell as the sorted names
121.  def sort_points_names_by_dist(center: np.ndarray, points: np.ndarray,
    names: list) -> tuple[np.ndarray, np.ndarray, list]:
122.      distances: list = np.array(outlying_distances(center, points))
123.      new_names: np.ndarray = np.array(names)
124.      sorted_indicies: list = np.argsort(distances)
125.      distances: list = list(distances[sorted_indicies])
126.      new_names: list = list(new_names[sorted_indicies])
127.      points: np.ndarray = points[sorted_indicies]
128.      return distances, points, new_names
129.
130.  #Creats a count of how often each kind appears in each cluster
131.  #Returns a list with dicts containig the count of items and also a
    score where 100% means that all clusters only contain one kind and 0%
    means that all kinds are distributed evenly
132.  def k_means_score(cluster_names: list) -> tuple[list, float]:
133.      cluster_names: list = [[name[:-7] for name in sublist] for sublist
    in cluster_names]
134.      all_names: list = [item for sublist in cluster_names for item in
    sublist]
135.      without_duplicates: list = list(dict.fromkeys(all_names))
136.      result: list = []
137.      for sublist in cluster_names:
138.          count_dict: dict = {name: sublist.count(name) for name in
    without_duplicates}
```

```
139.            result.append(count_dict)
140.        total_items: int = len(all_names)
141.        max_counts: list = [max(cluster.values()) for cluster in result]
142.        actual_score: int = sum(max_counts)
143.        normalized_score: float = round(actual_score / total_items, 2)
144.        return result, normalized_score
145.
146.    #Get succeses in prediticting the right name for a test point by
    looking at its closest distance to a centroid
147.    #Returns the percentage of successes of guessing the point in test
    correct
148.    def get_successes(centroids: np.ndarray, test: np.ndarray,
    distribution: list, test_name: list) -> float:
149.        successes: int = 0
150.        for i, point in enumerate(test):
151.            i_centroid: int = np.argmin([np.linalg.norm(point - centroid)
    for centroid in centroids])
152.            pred_name: str = max(distribution[i_centroid],
    key=distribution[i_centroid].get)
153.            real_name: str = test_name[i][:-7]
154.            certainty: float = distribution[i_centroid][pred_name] /
    sum(distribution[i_centroid].values())
155.            #print(pred_name, real_name, round(certainty, 2) * 100)
156.            successes += (pred_name == real_name)
157.        return round(successes / len(test), 2)
158.
159.    #Run K-Means with sklearn to test performace
160.    #Returns classification and the centroids
161.    def sklearn_kmeans(points: np.ndarray, k: int, init = "k-means++") ->
    tuple[list, np.ndarray]:
162.        kmeans: KMeans = KMeans(n_clusters=k, init=init, random_state=42)
163.        classification_sklearn: list = kmeans.fit_predict(points)
164.        centroids_sklearn: np.ndarray = kmeans.cluster_centers_
165.        return classification_sklearn, centroids_sklearn
166.
167.    #From a list with sublists containing image names, copies those images
    into numareted folders, Due to the copying of the images rather than
    saving, if not cleared before the images will just get added to the
    folders if same destination is used twice
168.    #Returns None
169.    def order_save_images(cluster_n: list, top_folder_path: str =
    "..\Processed Images\Results", folder_prefix: str = "Cluster_",
    current_image_folder_path: str = "..\Processed Images\Train") -> None:
170.        for i, cluster in enumerate(cluster_n):
171.            folder_name: str = folder_prefix + str(i)
172.            folder_path: str = top_folder_path + "/" + folder_name
173.            os.makedirs(folder_path, exist_ok = True)
174.            for image_name in cluster:
```

```
175.                shutil.copy(current_image_folder_path +"/"+ image_name,
    folder_path+ "/" + image_name)
176.
177.    image_dir_train: str = "..\Processed Images\Train"
178.    image_dir_test: str = "..\Processed Images\Test"
179.    n_test_images: int = 50
180.    train, names_train = load_images(image_dir_train)
181.    test, names_test = load_images(image_dir_test)
182.    #train, test, names_train, names_test = get_train_test_dataset(images,
    image_names, n_test_images)
183.
184.    k: int = 15
185.    dimensions: np.ndarray = train.shape[1]
186.    meanface: np.ndarray = np.mean(train, axis=0)
187.    A: np.ndarray = train - meanface
188.    eigenfaces, s, VT = np.linalg.svd(A.transpose(), full_matrices=False)
189.    eigenfaces: np.ndarray = eigenfaces.transpose()
190.    coeff_mat: np.ndarray = coeff_matrix(meanface, eigenfaces, train)
191.
192.    points: np.ndarray = coeff_mat.T
193.    classification: list = []
194.    socre = 0
195.
196.    centroids: np.ndarray = init_centroids(k, points)
197.    init_cent: np.ndarray = centroids.copy()
198.    last_centroids: np.ndarray = np.zeros_like(centroids)
199.
200.    while is_not_converged(last_centroids, centroids):
201.        last_centroids = centroids.copy()
202.        classification, centroids = run_iteration(points, centroids, k)
203.
204.    cluster_n: list = cluster_names(classification, names_train, k)
205.    distribution, score = k_means_score(cluster_n)
206.
207.
208.    print("K-Means score: " + str(score * 100) + "%")
209.    print(distribution)
210.
211.    #for i in range(5):
212.        #save_image(meanface + np.matmul(centroids[i], eigenfaces),
    "k_means_" + str(i + 1))
213.
214.    #order_save_images(cluster_n)
215.    #print("Successes K-Means: " + str(get_successes(np.matmul(centroids,
    eigenfaces), test, distribution, names_test) * 100) + "%")
216.
217.    #Comment out to verify our k means result by using the k means
    implemented with sklearn
```

```
#classification_sklearn, centroids_sklearn = sklearn_kmeans(points, k)
#cluster_n_sklearn: list = cluster_names(classification_sklearn,
    names_train, k)
#distribution_sklearn, score_sklearn = k_means_score(cluster_n_sklearn)
#print("K-Means score sklearn: " + str(score_sklearn * 100) + "%")

#Safe the images in the clusters into folders
#order_save_images(cluster_n)
```