Can ML Tell Allison and Ingrid Apart?

Problem Definition

Ever since the first day of Minerva, people have been struggling to tell me and Allison apart. Fair enough, we are both awesome strong independent women, but are we really that similar in appearance? Students, staff, strangers, and even Ben Nelson has mistaken us for each other, and we absolutely LOVE IT. We have made it into a brand at this point, and we keep people on their toes with matching clothes, dyeing our hair the same way, and other shenanigans.

So, the big question is: can computers tell us apart better than humans? Google Photos is definitely having a hard time sometimes, asking if we are the "same or different person?", but that pops up often enough with other people as well. So do we look the same to a computer?

Solution Specification

Originally I wanted to make something like Google Photos, where you can have several people in your pictures and it groups the pictures by people. However, this proved to be a bit more difficult than what my procrastinating self had time for, so I split into just images of me and Allison separately. Doing it this way, only allowing one face per image, I avoided having random backgrounds (e.g. trees, shirt creases etc.) be recognized as faces and the labelling and classification was easier.

(The "images/both" folder is left over from my first tries with multiple faces, but I decided to keep some of the pictures in case you want to see some of our fun matching outfits.)

To train the model to classify the images as either me or Allison, we first need to find the faces in the images to avoid the background affecting the classification too much. After extracting the faces, we classify them.

To find the faces, we use Haar cascading detection with OpenCV. Haar cascading detection is an object detection method that trains on many positive (contains object) and negative (does not contain object) images to learn what features define the object. Kernels are used in all possible locations and sizes of the training images to calculate features, and then Adaboost is used to find the features that are most relevant for detecting the object (OpenCV, 2022). These are the features that are used to quickly find faces in new images.

Technically, a CNN would probably do a better job at detecting faces than Haar cascading. However, OpenCV includes pre-trained models for face detection with Haar cascading, which makes it much more convenient to use in this case since the dataset is so small that we cannot really train a good CNN. Even if I had more data, using pre-trained Haar cascades is a lot less computationally expensive and time-consuming than training and using a CNN.

Having isolated the faces, we can use any classifier to classify them. In this project we use KNN, as we never actually covered this simple model in class and it works for small datasets. KNN takes the training data, in this case image arrays, and stores it as feature vectors with class labels. When introducing new unlabeled data points, the data is classified by looking at the k nearest labeled feature vectors and choosing the majority class (Starmer, 2017).

(All images in this assignment are included with the consent of Allison and Paul.)

Implementation

In [ ]:
```
# run this cell if dependencies are not installed
# common packages such as numpy are assumed to be installed already
%pip install opencv-python
%pip install Pillow

from IPython.display import clear_output
clear_output()
```

In [ ]:
```
import cv2
import os
import numpy as np
from PIL import Image

import matplotlib.pyplot as plt
```

Time to do like the hacker bois in the movies: we find faces and put boxes around them. Not just because it makes us feel cool, but because we want to single out the faces in the images as mentioned in the solution specification. This is shown for an example image with two faces below, but in the actual dataset and classification we will only work with images that contains one face. The color is different from the original image because of the way OpenCV orders the color layers, but this makes no difference for classification as this color change happens the same way to all images.

In [ ]:
```
# load the image
example = os.path.join('.', 'images/both/IMG_3184.JPG')
imgtest = cv2.imread(example, cv2.IMREAD_COLOR)
image_array = np.array(imgtest, "uint8")

 # for detecting faces
facecascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

# get the faces detected in the image
faces = facecascade.detectMultiScale(imgtest,
    scaleFactor=1.2, minNeighbors=5)

for (x_, y_, w, h) in faces:
    face_detect = cv2.rectangle(imgtest,
                (x_, y_),
                (x_+w, y_+h),
                (255, 0, 255), 2)
```

```
plt.imshow(face_detect)
plt.show()
```

To recognize the faces, we used the pre-trained frontal face Haar cascade model from OpenCV. The model was downloaded from OpenCV's Github, here (using the file `haarcascade_frontalface_default.xml`).

We can now use this to extract the faces from our pictures that we will use for the classification (in the folders "allison" and "ingrid").

In [ ]:
```
## adapted from https://www.codemag.com/Article/2205081/Implementing-Face-Recognition-Using-Deep-Learning-and-Support-Vector-

def get_faces(imgs: str, dir: str):
    """
    Takes a folder of images, finds the face in it if there is
    exactly one face, and saves the face images to a specified directory.
    The function returns nothing, but creates or adds face images to
    specified directory.

    Parameters:
    -------------
    imgs: str
        Directory with input images (relative to current file).
    dir: str
        Output directory for face images (relative to current file).
    """
    image_width = 224
    image_height = 224

    # Haar cascade for detecting faces
    facecascade = cv2.CascadeClassifier('haarcascade_frontalface_default.xml')

    # set the directory containing the images
    images_dir = os.path.join(".", imgs)

    k = 0
    # iterates through all the files in each subdirectories
    for root, _, files in os.walk(images_dir):
        for file in files:
            # this implementation only considers jpg images
            if file.endswith("JPG") or file.endswith("jpg"):
                path = os.path.join(root, file)
            else:
                continue

            # get the label name (name of the person)
            label = os.path.basename(root).replace(" ", ".").lower()

            # load the image as an array
            imgtest = cv2.imread(path, cv2.IMREAD_COLOR)
            image_array = np.array(imgtest, "uint8")

            # get the faces detected in the image
            faces = facecascade.detectMultiScale(imgtest,
                scaleFactor=1.2, minNeighbors=5)

            # if not exactly 1 face is detected, skip this photo
            if len(faces) != 1:
                print(f'Photo skipped: {file} w/ {len(faces)} "faces"')
                continue

            # if the specified output directory does not already exist, create it
            if not os.path.exists(f'./{dir}/'):
                os.mkdir(f'./{dir}/')

            for (x_, y_, w, h) in faces:
                # make array of region of interest (the face)
                roi = image_array[y_: y_ + h, x_: x_ + w]

                # resize the detected face to 224x224
```

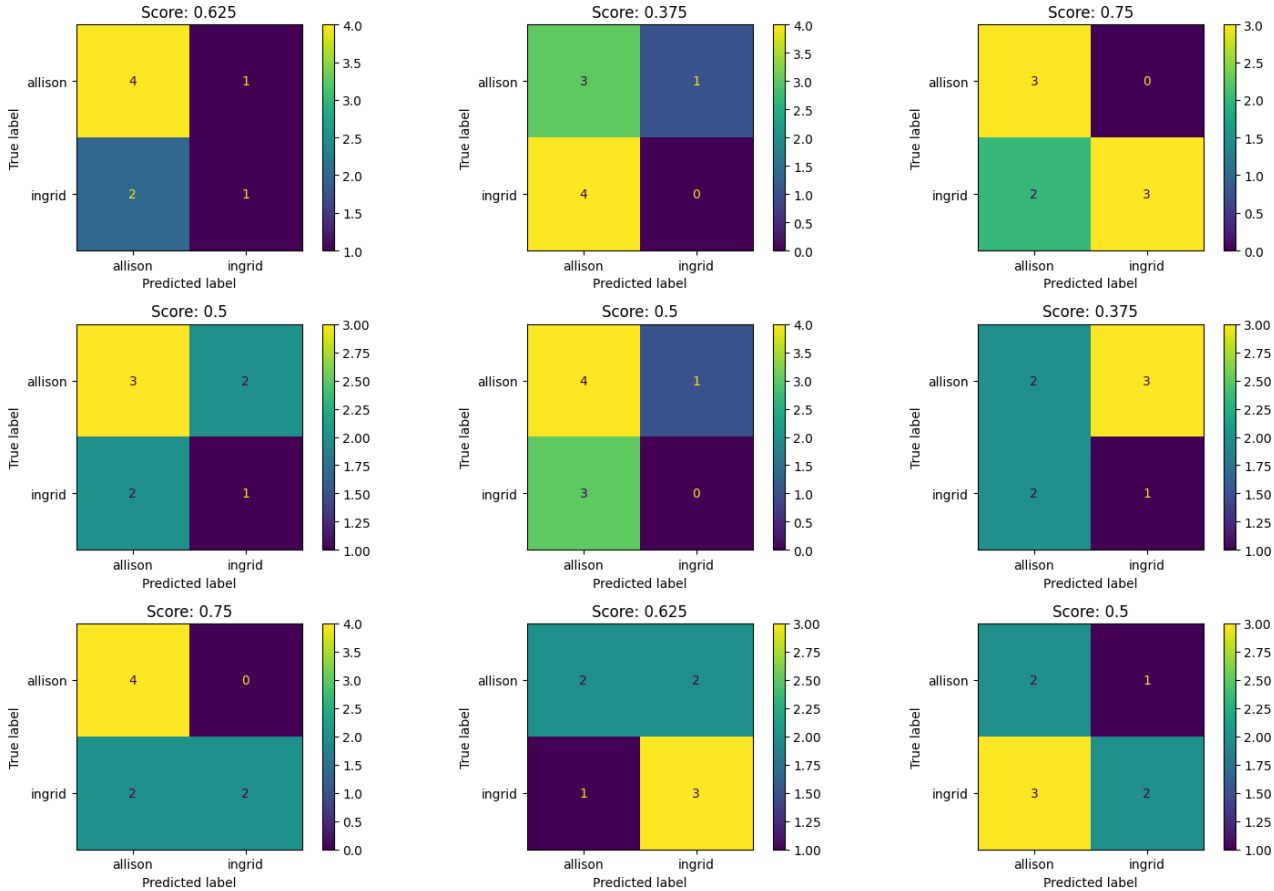
```
                size = (image_width, image_height)
                resized_image = cv2.resize(roi, size)
                resized_array = np.array(resized_image, "uint8")

                # save image as
                im = Image.fromarray(resized_array)
                im.save(f'./{dir}/{label}_{k}.jpg')

            k += 1


get_faces("images/ingrid", "data_algridingson")
get_faces("images/allison", "data_algridingson")
```

```
Photo skipped: IMG_14.jpg w/ 2 "faces"
Photo skipped: IMG_8.jpg w/ 3 "faces"
Photo skipped: IMG_111.jpg w/ 2 "faces"
Photo skipped: IMG_123.jpg w/ 0 "faces"
Photo skipped: IMG_124.jpg w/ 0 "faces"
```

Now we can classify our faces using KNN. Since the dataset is small, we make a function that takes several random states for splitting the data and returns the results for all of them. The metric returned is the accuracy (proportion of correct classifications) and a confusion matrix. These will show how well the model predicts the test data.

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import ConfusionMatrixDisplay
import re

def knn(img_dir: str, rand_st: list):
    """
    Trains a KNN model on part of the input images and tests the model
    on the remaining data. Prints accuracy score and confusion matrix for
    random split into test and training data.

    Parameters:
    img_dir: str
        Directory with images to be classified
    rand_st: list
        list of random states for train_test_split
    """

    # make arrays of the face images, lst, and a corresponding array of labels
    lst = []
    labels = []
    for root, _, files in os.walk(img_dir):
        for file in files:
            path = os.path.join(root, file)
            # make image into array
            imgtest = cv2.imread(path, cv2.IMREAD_COLOR)
            image_array = np.array(imgtest, "uint8").flatten()

            lst.append(image_array)

            # find label from filename and append to labels
            name = re.match(".*?(?=_)", file).group(0)
            labels.append(name)

    # initialize plots
    n = np.sqrt(len(rand_st))
    if int(n) == n:
        m = int(n)
    else:
        m= int(n)+1
    fig, axes = plt.subplots(nrows=int(n), ncols=m, figsize=(15,10))

    # for each random state in rand_st, split the data,
    # make a model and get metrics
    for randst, ax in zip(rand_st, axes.flatten()):
        # split data using random state
        x_train, x_test, y_train, y_test = train_test_split(lst, labels,
                    test_size = 0.15,
                    random_state=randst)

        # make model
        model = KNeighborsClassifier(n_neighbors=3)
        model.fit(x_train, y_train)

        # get score (accuracy) and confusion matrix
        sc = model.score(x_test, y_test)
        y_pred = model.predict(x_test)
        ConfusionMatrixDisplay.from_predictions(y_test, y_pred, ax=ax)
        ax.title.set_text(f"Score: {sc}")

    plt.tight_layout()
    plt.show()
```

In [ ]:
```
# try with 4 random states if this takes too long to run
rand_states = [1, 2, 3, 25, 50, 100, 123, 234, 345]
algrid = os.path.join(".","data_algridingson")

knn(algrid, rand_states)
```

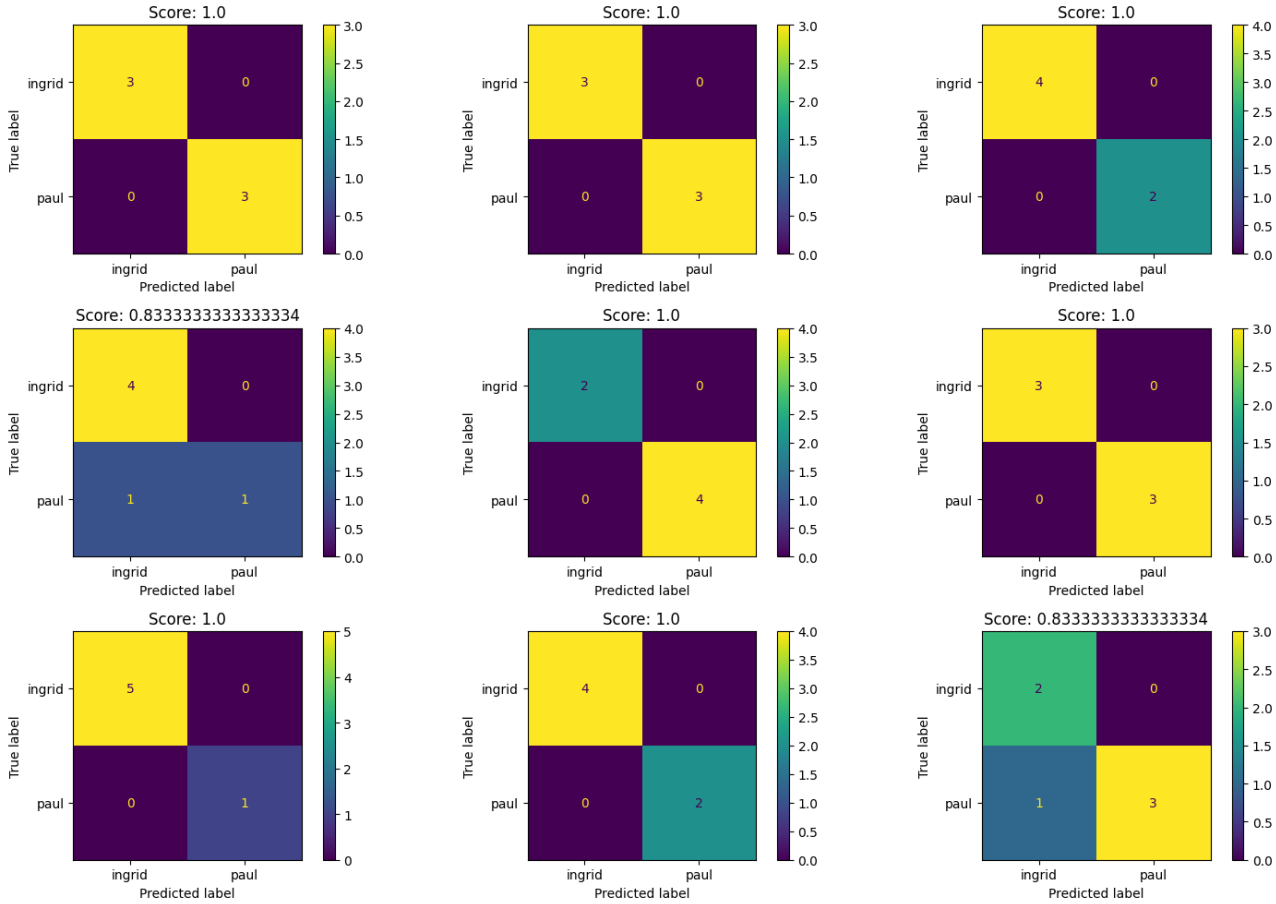


Sooooo, the classification is "meh" at best - sometimes it even has more misclassifications than correct classifications. But, is this because Allison and I are hard to tell apart, or is it just because the dataset is so small that the model is not great? Lets test if we get the same results when comparing me and Paul instead:

In [ ]:
```
get_faces("images/ingrid", "data_paul")
get_faces("images/paul", "data_paul")
```

```
Photo skipped: IMG_14.jpg w/ 2 "faces"
Photo skipped: IMG_8.jpg w/ 3 "faces"
Photo skipped: IMG_507.jpg w/ 2 "faces"
Photo skipped: IMG_515.jpg w/ 0 "faces"
Photo skipped: IMG_501.jpg w/ 2 "faces"
```

In [ ]:
```
paul = os.path.join(".","data_paul")

knn(paul, rand_states)
```



And look at that! Even with the same size (ish) of the dataset, the classifier seems to have no issue telling me and Paul apart. Maybe Allison and I are difficult to tell apart after all!

Possible Extensions

To make the conclusion more solid and evidence-based, the dataset should be extended to include many more pictures. An alternative to trying several different random states for the split into test and training data could be using crossvalidation and looking at the best, worst, and/or average models. The classifier could also be changed from KNN to any other classifier to see if the results are repeated using other models.

If allowing several faces per images, k-means (without labels) might be a better approach, as labeling can get difficult when many faces (or things that are not faces at all) are detected.

References

Lee, W. (2022). Implementing Face Recognition Using Deep Learning and Support Vector Machines. Retrieved December 13, 2022, from https://www.codemag.com/Article/2205081/Implementing-Face-Recognition-Using-Deep-Learning-and-Support-Vector-Machines

OpenCV. (2022). Cascade classifier. OpenCV. Retrieved December 13, 2022, from https://docs.opencv.org/3.4/db/d28/tutorial_cascade_classifier.html

Starmer, J. (2017). StatQuest: K-nearest neighbors, Clearly Explained. Retrieved December 14, 2022, from https://www.youtube.com/watch?v=HVXime0nQeI