AdaBoost

March 5, 2023

0.0.1 Quick introduction to jupyter notebooks

- Each cell in this notebook contains either code or text.
- You can run a cell by pressing Ctrl-Enter, or run and advance to the next cell with Shift-Enter.
- Code cells will print their output, including images, below the cell. Running it again deletes the previous output, so be careful if you want to save some results.
- You don't have to rerun all cells to test changes, just rerun the cell you have made changes to. Some exceptions might apply, for example if you overwrite variables from previous cells, but in general this will work.
- If all else fails, use the "Kernel" menu and select "Restart Kernel and Clear All Output". You can also use this menu to run all cells.
- A useful debug tool is the console. You can right-click anywhere in the notebook and select "New console for notebook". This opens a python console which shares the environment with the notebook, which let's you easily print variables or test commands.

0.0.2 Setup

```python
[272]: %reload_ext autoreload
%autoreload 2

import numpy as np
from matplotlib import pyplot as plt
from scipy import io as sio

from utils import GenerateHaarFeatureMasks, GenerateTrainTestData,
  ↪ExtractHaarFeatures
from utils import PlotErrorGraphs, PlotClassifications,
  ↪PlotSelectedHaarFeatures, PlotHaarFeatureDemonstration, PlotSolvayHeatmap
plt.rcParams['figure.facecolor']='white'

# Load data
faces = sio.loadmat('Data/faces.mat')['faces'].astype("float")
nonfaces = sio.loadmat('Data/nonfaces.mat')['nonfaces'].astype("float")
```

0.0.3 *! IMPORTANT NOTE !*

Your implementation should only use the numpy module (already imported as np). The numpy module provides all the functionality you need for this assignment and makes easier debuging your

code. No other modules, e.g. scikit-learn or scipy among others, are allowed and solutions using modules other than the numpy module will be sent for re-submission.

0.0.4 1 Introduction

In this assignment you will explore a branch of supervised learning called *Ensemble lerning*, specifically using the algorithm known as *AdaBoost*. This algorithm trains multiple simple *weak classifiers* that are later combined into a *strong classifier* which is significantly more powerful than the individual *weak classifiers*. The *weak classifier* you will work with is a simple type of binary classifier called decision stump. These classify scalar inputs based on a single comparisson with a threshold.
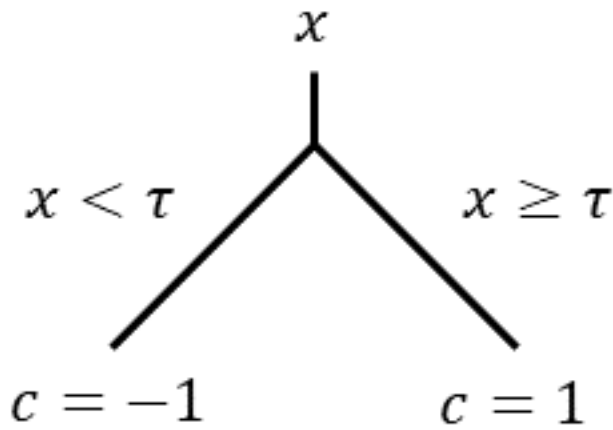

In order to apply this classifier to images, we require a way to describe the features of the images using a single number that the decision stump can compare to the threshold. In a seminal article by Viola and Jones, called Robust real-time object detection, this was demonstrated using so called Haar-features, which is what we will use in this assignment.

A Haar-feature is a simple yet surprisingly effective way to quantify edges and gradients in images, by computing the weighted sum of different areas of the image. We will begin by exploring how these Haar-features work, in order to understand the fundamental building blocks of the algorithm you will implement later.

Run the following code cell to generate a random Haar-feature, select a random face and nonface image, and compute the feature values by applying the Haar-feature to the images.

```
[273]: # Generate random Haar-feature and select a random face and a random non-face
       randomHaarFeatureMask = GenerateHaarFeatureMasks(1)[:,:,0]
       randomFace = faces[:,:,np.random.randint(faces.shape[2])]
       randomNonFace = nonfaces[:,:,np.random.randint(nonfaces.shape[2])]

       PlotHaarFeatureDemonstration(haar=randomHaarFeatureMask, face=randomFace,␣
        ↪nonface=randomNonFace)
```
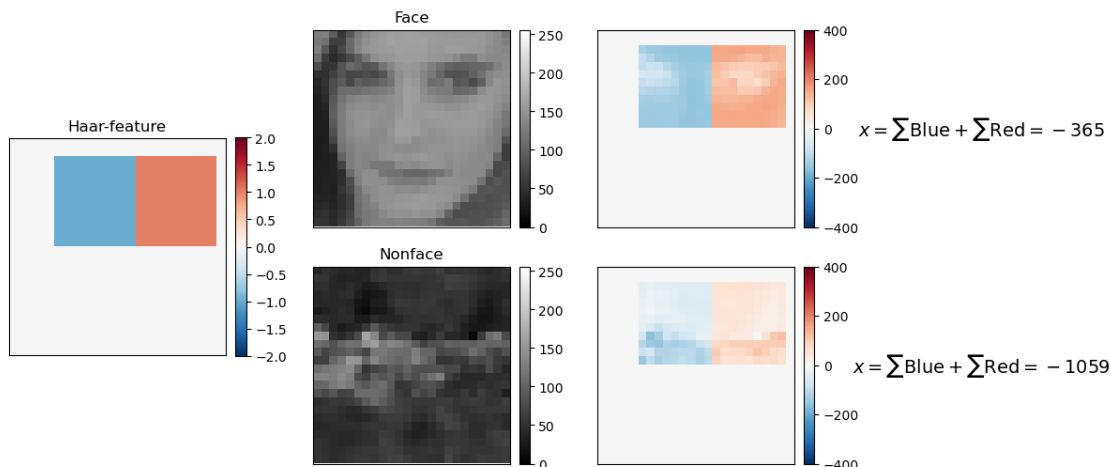
The Haar-features we generate in this assignment consist of either two or three areas, each with a scaling factor of -1, 1, or 2. The pixels outside the Haar-feature has a scaling of 0. The images are pixel-wise multiplied with the Haar-feature, after which the entire image is summed to a single scalar value. This weighted sum of pixels is the feature value of the image, which can be used in the weak classifier.

Of course, since the Haar-feature in this picture is chosen entierly at random, we should not expect that it makes sense as a good feature extractor. Finding a good set of Haar-features is your task, but we will get back to that soon. Before that, take a look at the decision stump classifier again. Now that we know how to extract a single feature value from an image, we can see how a desision stump can be used to classify the images as faces or nonfaces. We can write the decision stump as a function:

$$f(x \mid \tau) = \begin{cases} +1 & \text{if } x \geq \tau \\ -1 & \text{if } x < \tau \end{cases}$$

As you have already seen in the lecture on boosting and ensemble learning, this can also be extended to a more general definition by including a polarity parameter:

$$f(x \mid \tau, p) = \begin{cases} +1 & \text{if } px \geq p\tau \\ -1 & \text{if } px < p\tau \end{cases}$$

By selecting p equal to +1 or -1, we can flip the sign of the predicted output. Since this is a binary classification, this also changes the error of the classifier from e to $1 - e$. This is necessary if we want to guarantee that we can find the optimal classifier.

1.1 Generating the training data Since a single random Haar-feature is unlikely to be a good feature extractor, we need to generate a large number of them and find the ones that are useful. But extracting the feature values from the images during training is very inefficient, so instead we precompute the feature value for each Haar-feature applied on each image. This will therefore

3

result in a feature value matrix $X^{[R,C]}$ (i.e. R rows and C columns), where R is the number of Haar-feature and C is the number of images in the dataset, as illustrated in this picture:

Note that the training loop will only work with this matrix, never with the original images or Haar-features.

1.2 The AdaBoost algorithm AdaBoost is an ensemble learning algorithm that successively trains many weak classifiers that are informed about the weaknesses of the previous classifiers. This is done by keeping track of a weight value d for each training image, representing how "difficult" that image has been to classify during training so far. After a weak classifier has been trained, the predictions of that classifier is used to update the weights of all images. If an image was classified correctly the weight is exponentially decreased, and the next weak classifier will therefore put less emphasis on that sample since it is already taken care of by the previous weak classifiers. The weights of misclassifed images are instead exponentially increased to make them more important for the next weak classifier.

In more detail, if a weak classifier results in an error ε, we compute the update factor

$$\alpha = \frac{1}{2}\ln\left(\frac{1-\varepsilon}{\varepsilon}\right)$$

and then update the weights according to

$$d_{t+1} = \begin{cases} d_t e^{-\alpha} & \text{if correctly classified} \\ d_t e^{\alpha} & \text{if misclassified} \end{cases}$$

We also normalize d_{t+1} such that the total weight of all samples is 1. The next weak classifier is then trained using the new weights d_{t+1}, which will put the focus on missclassified images and therefore result in a new combination of optimal parameters. This is how AdaBoost gradually covers more and more of the difficult cases in the data, even thought the same training images and Haar-feature are used for each weak classifier.

4

When the training of all weak classifiers is finished we use them in a weighted voting scheme to build the *strong classifier.*

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$$

Note that we use the update factor α for each weak classifier as a weight in the voting. This is because weak classifiers with lower total error should have more voting power to increase the overall performance of the strong classifier.

Question 1: The error of a single weak classifier is always in the range 0 to 0.5 (since we otherwise flip the polarity). Consider the two edge cases: 1. If you get an optimal $\varepsilon = 0$ this single weak classifier correctly predicted the entire training set. What does this indicate about your hyperparameters (number of Haar-features, number of training images, number of weak classifiers)? 2. If you get an optimal $\varepsilon = 0.5$ the training will get stuck. Explain why this is the case.

Answer: [1. There are very few number of training images, it does not say anything about the number of Haar-features or number of weak classifiers 2. If epsilon is 0.5, the formula for alpha will give alpha=0 which in turn stalls the updating of the weights. The weights gets updated by a factor which is always 1 if epsilon is 0.5. So it never updates the weights.]

1.3 Pseudo-code With all this in mind, we can summarize the entire algorithm in the following pseudo-code:

TrainWeakClassifier: >for every combination of (feature, threshold, polarity): » Run WeakClassifier » Measure error » if new smallest error: »> Save parameters > > return optimal parameters

TrainStrongClassifer: > Initialize weights for each training sample > for number of weak classifiers: » TrainWeakClassifier » Measure error and get predictions » Compute update parameter alpha » Update training sample weights > >return optimal parameters

---

0.0.5   2. Implementing the weak classifier

You will now start by implementing the decision stump weak classifier, the weighted error function, and the training loop for a single weak classifier.

2.1 Weak classifier forward function The forward function classifies the data according to the threshold and polarity.

```
[274]:  # Implement decision stump classifier
        # X - Feature values (vector)
        # t - Threshold (scalar)
        # p - Polarity (1 or -1)
        # Return a vector of predicted classes (1 or -1), same shape as X.
```

```
def WeakClassifier(X,t,p):
    preds = []
    for f in X:
        if f == t:
            preds.append(1)
        elif f > t:
            preds.append(p)
        elif f < t:
            preds.append(-p)

    return preds
```

2.2 Weighted classification error This function computes the weighted classification error based on predicted and true classes, as well as the weight vector.

[275]:
```
# Implement error function
# Y  - Target classes (vector)
# Yp - Predicted classes (vector)
# D  - Weights (vector)
# Return a scalar for the total weighted error of all predictions.

def WeakClassifierError(Y,Yp,D):
    epsilon_error = 0
    for i in range(len(Y)):
        if Y[i] != Yp[i]:
            epsilon_error += D[i]

    return epsilon_error
```

2.3 Training function for a single weak classifier Here you should implement the training of a single weak classifier, using the two functions you just implemented. Based on the training data, sample weights, and target labels, find the optimal way to separate the two classes. You should do this using brute force search, in other words, try every (reasonable) combination of parameters, measure the errors, and pick the best. Return all parameters that define the optimal weak classifier, as well as the error of said weak classifier.

Tip: Brute force search is not an efficient algorithm, so any optimization you can do is welcome. When you have a working version of this function, consider if there is any way you can decrease the amount of calls to WeakClassifier, especially by reducing the number of thresholds. Of course, you must still guarantee that you can find the optimal parameters, so for example naively discarding every other threshold will not work.

[317]:
```
# Implement function for training one decision stump
# X - Training data (matrix)
# Y - Target classes (vector)
# D - Error weights (vector)
# Return optimal feature, threshold, polarity, and the corresponding error
```

6

```python
import random

def TrainWeakClassifier(X,Y,D):

    fOpt = None
    tOpt = None
    pOpt = None
    eMin = np.inf
    YPredOpt = None
    # --------------------------------------------
    # === Your code here ========================
    # --------------------------------------------
    x_min = min(X.flatten())
    x_max = max(X.flatten())

    p = 1
    YPred = None
    for i, row in enumerate(X):
        thresholds = np.unique(row)
        tresholds = thresholds[i:]
        thresholds = sorted(thresholds)
        if len(thresholds) > 10:
            thresholdLength = len(thresholds)//10
        else:
            thresholdLength = len(thresholds)
        for t in range(0, thresholdLength):
            medianThreshold = thresholds[t * 10 + 5]
            YPred = WeakClassifier(row, medianThreshold, p)
            epsilon_error = WeakClassifierError(Y, YPred, D)
            if epsilon_error > 0.5:
                epsilon_error = 1 - epsilon_error
                p *= -1
            if epsilon_error < eMin:
                eMin = epsilon_error
                fOpt = i
                tOpt = medianThreshold
                pOpt = p
                YPredOpt = YPred
    # ==========================================
    return fOpt, tOpt, pOpt, eMin, YPredOpt
```

Test your implementation by running the following minimal test case. You should find that the optimal feature is 1 (remember that python is 0-indexed!), the optimal threshold is between 1 and 3 (depending on your implementation), and the optimal polarity is -1. The minimum error should be 0.222 repeating.

```
[316]: X = np.array([[2, 2, 2, 2, 2, 3], [3, 3, 3, 1, 1, 4], [4, 2, 4, 2, 4, 2]])
       Y = np.array([-1,-1, -1, 1, 1, 1])
       D = np.array([1,1,1,2,2,2]) / 9

       fOpt, tOpt, pOpt, eMin, YPredOpt= TrainWeakClassifier(X,Y,D)
       print(f"Optimal feature   : {fOpt}")
       print(f"Optimal threshold : {tOpt}")
       print(f"Optimal polarity  : {pOpt}")
       print(f"Minumum error     : {eMin}")
```

```
Optimal feature   : 1
Optimal threshold : 3
Optimal polarity  : -1
Minumum error     : 0.2222222222222222
```

0.0.6 3. Implementing the strong classifier

You should now implement the main AdaBoost algorithm, using the training function for weak classifiers you just implemented. Return the optimal parameters for all weak classifiers. We have given you some boilerplate code here to help you get started, and to provide some useful outputs during the training.

```
[296]: from time import perf_counter as tic
       from datetime import timedelta

       # Implement AdaBoost training
       # X - Training data (matrix)
       # Y - Target classes (vector)
       # N - Number of weak classifiers to train (scalar)
       # Return the optimal features, thresholds, polarities, and alphas describing␣
        ↪the N weak classifiers


       def TrainStrongClassifier(X,Y,N=10):

           fOpt = np.zeros(N,"int")
           tOpt = np.zeros(N)
           pOpt = np.zeros(N)
           aOpt = np.zeros(N)


           # ----------------------------------------------
           # === Your code here =======================
           # ----------------------------------------------


           # Initialize sample weights here
           D = [1/len(Y)] * len(Y)
```

8

```
    # =============================================

    timeStart = tic()
    for n in range(N):
        # ---------------------------------------------
        # === Your code here ========================
        # ---------------------------------------------

        fOpt[n], tOpt[n], pOpt[n], eMin, YPredOpt = TrainWeakClassifier(X, Y, D)
        aOpt[n] = 1/2 * np.log((1-eMin)/ eMin)
        for i in range(len(Y)):
            D[i] = D[i] * np.exp(-aOpt[n] * Y[i] * YPredOpt[i])
            D[i] = D[i] / np.sum(D)
            #WeakClassifier(X[i], tOpt[n], pOpt[n])[i]


        # =============================================
        # This prints the training progress in a nice format
        timeLeft = round((tic()-timeStart)*(N/(n+1) - 1))
        etaStr = str(timedelta(seconds=timeLeft))
        print(f"{n+1} of {N}: ETA {etaStr}    ", end="\r")

    return fOpt, tOpt, pOpt, aOpt
```

Finally, you should implement the StrongClassifier function. This function takes in the parameters returned from the training function and classifies the input X. If input Y is also provided, the error of the strong classifier should also be computed and returned.

Additionally, it is interesting to evaluate the strong classifier not only on the final ensemble model, but how the performance improved during the training, as more and more weak classifiers were added. As such, the error returned from this function should be a vector with the same lenght as the number of weak classifiers. The first value in the vector should only use the first weak classifier. The second value should use the first two classifiers in the ensemble, and so on. In general, the n:th entry is the error when using the first n weak classifiers as a strong classifier. This way we can plot the performance as a function of number of weak classifiers, which is important to investigate, for example, overfitting.

Tip: There is a function in numpy called `cumsum`*, which stands for cumulative sum. It is not mandatory to use this, but the code for computing the error vector can be very short if you do.*

```
[340]:  # Implement strong classifier
        # Params - Parameters of the weak classifiers (output from AdaBoost training
         ↪function)
        # X - Data to classify (matrix)
        # Y - Target labels (vector). Optional input, see text description above.
        # Return predicted classes (vector) and optionally the error (vector).
        from operator import add

        def StrongClassifier(Params, X, Y=None):
```

9

```
    H = None
    Err = None


    # ---------------------------------------------
    # === Your code here =========================
    # ---------------------------------------------

    H = np.zeros((len(Params[0]), len(X[0])))
    Err = np.zeros(len(Params[0]))
    for i in range(len(Params[0])):

        if i == 0:
            H[i] = [Params[3][i] * j for j in WeakClassifier(X[Params[0][i]],␣
↪Params[1][i], Params[2][i])]
        if i != 0:
            H[i] = [H[i-1][j] + Params[3][i] * value for j, value in␣
↪enumerate(WeakClassifier(X[Params[0][i]], Params[1][i], Params[2][i]))]

    H = np.sign(H)
    if Y != None and Y.any():
        for i in range(len(Params[0])):
            temp_error = 0
            for j in range(len(Y)):
                if Y[j] != H[i][j]:
                    temp_error += 1
            Err[i] = temp_error / len(Y)

    # =============================================



    return H, Err
```

0.0.7 4. Training and evaluating the model

Now you will run the AdaBoost training and evaluate the results. We have provided all required code for plotting the results, but if there are additional results you wish to show you are welcome to make your own plots as well.

The following cell generates a set of random Haar-features, then precomputes the feature values and splits the data into a training set and a test set. Change the three hyperparamters; number of Haar-features, number of training samples, and number of weak classifiers to obtain the target performance 7% error or lower. Of course the error should be stable below the target as well.

```
[321]: # Generate Haar feature masks
       haarFeatureMasks = GenerateHaarFeatureMasks(nbrHaarFeatures = 100)
```

```
# Extract feature values and split into training and test sets
trainImages, testImages, XTrain, YTrain, XTest, YTest =␣
  ↪GenerateTrainTestData(faces, nonfaces, haarFeatureMasks, nbrTrainImages =␣
  ↪3000)

# Train the classifier
optParams = TrainStrongClassifier(XTrain, YTrain, N = 40)
```

40 of 40: ETA 0:00:00

Now evaluate your classifier on the training data and test data.

[322]:
```
HTrain, ErrTrain = StrongClassifier(optParams, XTrain, Y = YTrain)
HTest , ErrTest  = StrongClassifier(optParams, XTest , Y = YTest)
ErrTrain *= 100.
ErrTest  *= 100

print(f"Training error: {ErrTrain[-1]:.1f}%")
print(f"Test error: {ErrTest[-1]:.1f}%")
```
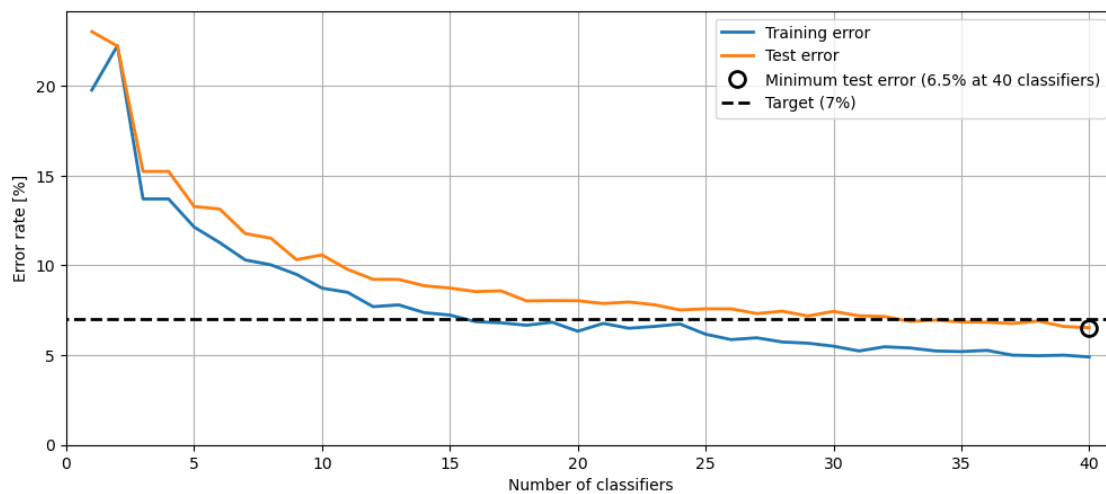
Training error: 4.9%
Test error: 6.5%

Plot the error of the strong classifier as a function of the number of weak classifiers.

[323]: `PlotErrorGraphs(ErrTrain, ErrTest)`

Question 2: Suppose you want to deploy your model in a real-world application, for example in a smartphone camera app that can highlight faces. Based on the above result, how many weak classifiers would you choose to include in the model? Motivate your answer.

Answer: [We would choose atleast over 15 for acceptable performance in real world application. 15 weak classifiers is at around 8% error rate and using up to more than double the weak classifiers only lowers the error rate by a single unit of %]

Question 3: Discuss your choise of hyperparameters * number of training images * number of Haar-features * number of weak classifiers

and how each affect the model performance. What are the advantages and disadvantages to low and high values for each parameter?

Answer: [We used 3000 training images, the advantage of increasing the number of training images is that the model will be more generalized. The disadvantage is that training the model takes longer. The number of Haar features used were 100. The advantage of increasing this amount is that it increases the diversity in weak classifiers and it can get better accuracy for more complex datasets. The disadvantage is that the mdoel can get overfitted to the dataset. The amount of weak-classifiers we used were 40. The advantage of the increasing the amount of weak-classifiers is better accuracy, too many can make the model overfit instead.]

4.1 Investigate the trained model Plot some examples of faces and nonfaces that were classified correctly and that were misclassified. To do this, we requre you to find the index in the test data of these four cases: * Faces classified as faces (F_F) * Faces classified as nonfaces (F_NF) * Nonfaces classified as faces (NF_F) * Nonfaces classified as nonfaces (NF_NF)

There are many ways to do this. It is possible to write each in a single line using numpy functions, although this is not required.

```
[1]: print(HTest.shape)
     print(YTest.shape)

     F_F   = (YTest == 1) & (HTest == 1)

     F_NF  = (YTest == 1) & (HTest == -1)

     NF_F  = (YTest == -1) & (HTest == 1)

     NF_NF = (YTest == -1) & (HTest == -1)


     PlotClassifications(testImages, F_F, F_NF, NF_NF, NF_F, N=3, selectRandom=True)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_1352\3828946551.py in <module>
----> 1 print(HTest.shape)
      2 print(YTest.shape)
      3
      4 F_F   = (YTest == 1) & (HTest == 1)
      5
```

```
NameError: name 'HTest' is not defined
```

Question 4: Run the above cell a few times and discuss some potential issues in the data that might explain these misclassified images. Can you think of any pre-processing that might improve the results?

Answer: [There might be and imbalance in classes, for example there could be more training data of actual faces. Then the model would be more biased towards detecting faces everywhere. To fix this you can make sure to balance the dataset at start. It could be poor image quality on certain data samples, to fix this you could enhance the image with some kind of filtering and edge detection technique before training on the dataset.]

Run the following code to plot (some) of your selected Haar-features. Feel free to change the paramters `shuffle` and `N`.

[338]:
```
# Inputs:
# - haarFeatureMasks: The list of all generated Haar features
# - selectedIdx: The feature numbers of Haar features selected in the training
# - shuffle (False/True): Select in random order or as they were selected in
  ↪training
# - N: The number of Haar features to plot

PlotSelectedHaarFeatures(haarFeatureMasks, selectedIdx = optParams[0], shuffle
  ↪= False, N = 12)
```

Chosen Haar features
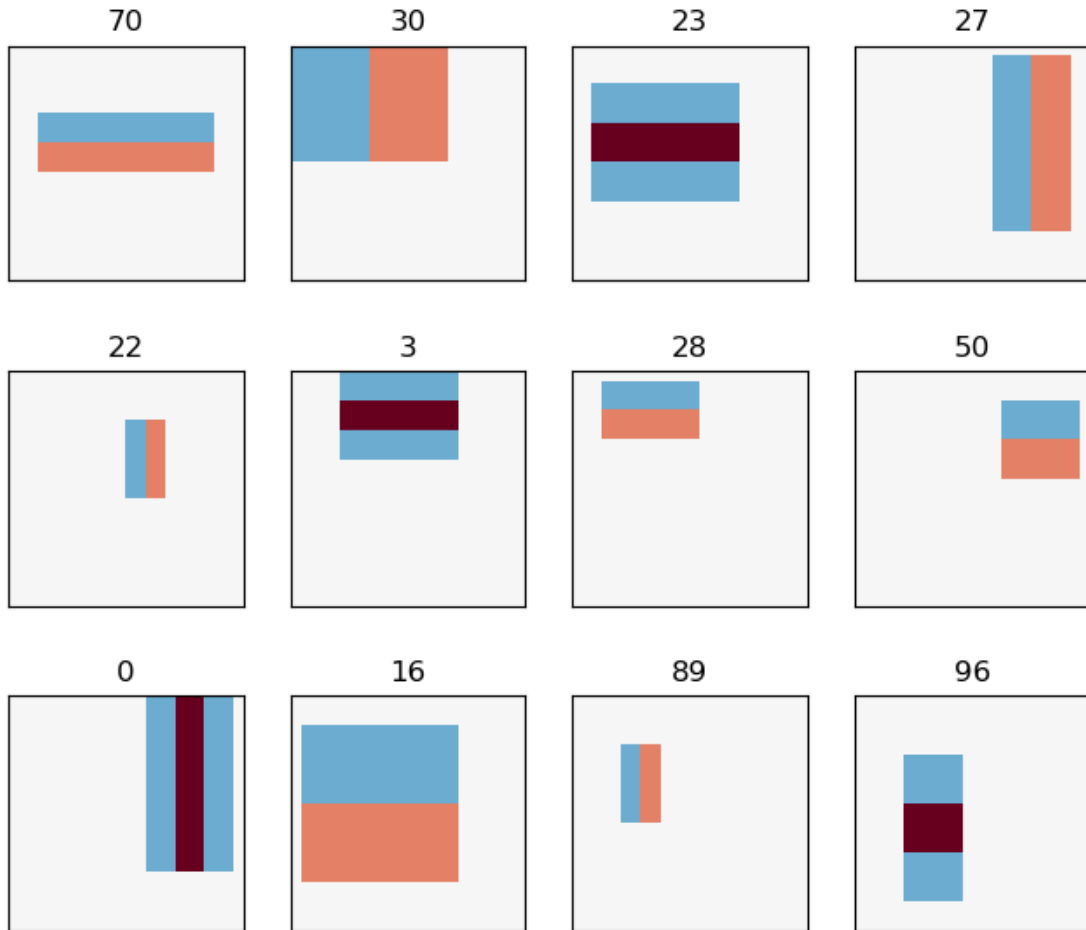
Question 5: Choose one or two of these Haar-features and speculate what feature of a face this might detect.

Answer: [Haar feature 3 could be used to detect the top of the head or perhaps the hairline. Number 70 could be detecting the forehead.]

0.0.8 5. Applying the model on real data

Evaluating on the test data is, of course, a valid method to measure the objective performance of the model. However, if doesn't really give a feel for how useful the model would actually be in a real world application. This final test applies the model to every 24x24 pixel subimage in this famous photograph of the 1927 Solvay Conference on Physics, and highlights the most likely parts of the image to contain faces.

Let's see how many of these famous faces your model can find!

```
[341]:  solvay = sio.loadmat('Data/solvay.mat')['X'].astype("float")
        xSolvay = ExtractHaarFeatures(solvay, haarFeatureMasks)

        cSolvay = StrongClassifier(optParams, xSolvay)

        PlotSolvayHeatmap(cSolvay)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_18576\2086157781.py in <module>
      4 cSolvay = StrongClassifier(optParams, xSolvay)
      5
----> 6 PlotSolvayHeatmap(cSolvay)

C:\NNLabs\LabCode\A3_Boosting\utils.py in PlotSolvayHeatmap(cSolvay)
    303       n   = np.zeros_like(img)
    304       for i,c in enumerate(cSolvay):
--> 305           hm[coords[i,0]:(coords[i,0]+24), coords[i,1]:(coords[i,1]+24)]
    ↪+= c
    306           n[coords[i,0]:(coords[i,0]+24), coords[i,1]:(coords[i,1]+24)] +
    ↪1
    307       hm /= n

ValueError: operands could not be broadcast together with shapes (24,24)
    ↪(40,159829) (24,24)
```

Question 6: Give a final discussion based on all the results. Do you think the model works well? Can you think of any way to improve the results?

Answer: [Your answer here]