Name1: Ali Hassan Ali Ahmed ElSharawy - **ID1:** 19016013

Name2: Ziad Mohamed Abuelkher - **ID2:** 19015733

Name3: Youssef Magdy Helmy - **ID3:** 19016937

Assignment 1: Face Recognition

Problem Statement:

We intend to perform face recognition. Face recognition means that for a given image you can tell the subject id. Our database of subject is very simple. It has 40 subjects. Below we will show the needed steps to achieve the goal of the assignment.

---

1. Download the Dataset and Understand the Format

Imports

In [ ]:
```
import os
import pandas as pd
import numpy as np
from sklearn.neighbors import KNeighborsClassifier
import cv2
import matplotlib.pyplot as plt
import sklearn as sk
from google.colab import files
!pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
from google.colab import drive
import sklearn as sk
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

connect to google drive to load data

In [ ]:
```
drive.mount('/content/drive')
```

check that we are in the right location otherwise , go to location of our data

In [ ]:
```
!pwd
!ls
```

face_recognition folder is where we stored the faces dataset

In [ ]:
```
os.chdir('drive/MyDrive/face_recognition')
!ls
!pwd
```

Result:

1PzB-dTdr2muezPn6LgPi1PK-hAsxdKNp s12 s17 s21 s26 s30 s35 s4 s8 face_recognition.zip s13 s18 s22 s27 s31 s36 s40 s9 s1 s14 s19 s23 s28 s32 s37 s5 view s10 s15 s2 s24 s29 s33 s38 s6 s11 s16 s20 s25 s3 s34 s39 s7 /content/drive/MyDrive/face_recognition

now we are at location of all faces, the next cell is done to get familiar of our data and display images

In [ ]:
```
dir = "s30"     #  feel free to change it from 1 to 40
os.chdir(dir)
string = "9"   #  feel free to change it from 1 to 10
x = cv2.imread(string+".pgm",-1)
nparr = x.reshape(1, 10304)
y = x
print("image reshape : ",x.shape)
print("\n")
print("data as matrix : ",x)
print("\n")
print("data type : ",type(x))
os.chdir('..')
x = list(x.flat)
print("\nimage after get flatten : ")
print(x)
print("\n")
print("class  : ",dir[1:3])
plt.imshow(y)
```

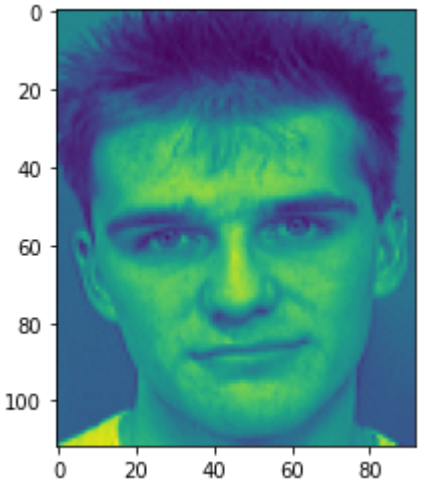
image reshape : (112, 92)

data as matrix : [[108 108 106 ... 103 106 105] [110 104 110 ... 104 102 109] [108 106 109 ... 103 105 104] ... [77 79 74 ... 73 71 69] [75 90 128 ... 70 71 70] [132 176 193 ... 69 69 70]]

data type : <class 'numpy.ndarray'>

image after get flatten : [108, 108, 106, 107, 106, 108, 108, 106, 107, 106, ...

class : 30



<matplotlib.image.AxesImage at 0x7f1da29085e0>

2. Generate the Data Matrix and the Label vector

In [ ]:
```
# !ls
# os.chdir("..")
# !pwd
```

Runtime of importing data: 3m 25s

load data from drive to local variables : data, label

In [ ]:
```
data = []
label = []
for i in range(1,41):
  os.chdir("s"+str(i))
  for j in range(1,11):
    img = cv2.imread(str(j) + ".pgm",-1)
    data.append(list(img.flat))
    label.append(i)
  os.chdir("..")
```

We are working with Numpy Arrays, so the data and labels lists are transformed to np.ndarray

In [ ]:
```
data = np.array(data)
label = np.array(label)
```

In [ ]:
```
print(data.shape)
```

```
(400, 10304)
```

(400, 10304)

3. Split the Dataset into Training and Test sets

In [ ]:
```
training_set = data[1::2] # even rows
training_labels = label[1::2]
testing_set = data[::2] # odd rows
testing_labels = label[::2]
```

---

4. Classification using PCA

A) Use the pseudo code below for computing the projection matrix U,

Helping Functions

In [ ]:
```
def compute_eigens(matrix):
  values, vectors = np.linalg.eigh(matrix)
  # putting them in descending order
  values = values[::-1]
  vectors = vectors[:, ::-1]
  return values, vectors
```

In [ ]:
```
def compute_eigens_number(alpha, eigen_values):
  total_sum = sum(eigen_values)
  for i in range(len(eigen_values)):
    explained_variance = sum(eigen_values[:i+1])/total_sum
    if(explained_variance >= alpha):
      return i+1
```

Main Algorithm

In [ ]:
```
def PCA_impl_repetetive(Data):
  mean_vector = np.mean(Data, axis = 0)
  centered_data = Data - mean_vector
  cov_matrix = np.cov(centered_data, bias=True, rowvar=False)
  eigen_values, eigen_vectors = compute_eigens(cov_matrix)
  return eigen_values, eigen_vectors
```

Comment: Above we can see the repetitive part in the pca algorithm, where we find the eigen vectors and eigen values.

Next, the implementation varies following the different values of alpha. So the common part is run once, then the different part will be run 4 times (for each value of alpha)

In [ ]:
```
def PCA_impl(alpha, eigen_values, eigen_vectors):
  fraction_of_total_variance = compute_eigens_number(alpha, eigen_values)
  projection_matrix = eigen_vectors[:, :fraction_of_total_variance]
  return projection_matrix
```

Comments:

rowvar=false means that the features are on the columns, not the rows (!default).

eigh() is used instead of eig() because:

1- covariance matrix is always symmetric.

2- eigh() returns sorted eigen values, while eig() doesn't.

3- faster than eig() with big matrices.

4- eigh neglect imaginary part

Define the alpha = {0.8,0.85,0.9,0.95}.

Runtime of generating eigen value & vectors: 5m

In [ ]:
```
alpha = [0.8, 0.85, 0.9, 0.95]
values, vectors = PCA_impl_repetetive(training_set)
projection_matrices = []
for i in range(len(alpha)):
  projection_matrices.append(PCA_impl(alpha[i], values, vectors))
```

In [ ]:
```
for i in range(len(projection_matrices)):
  print(projection_matrices[i].shape)
```

(10304, 37)

(10304, 53)

(10304, 77)

(10304, 116)

As we can see, eigen vectors chosen for alpha = 0.8 are 37, for alpha = 0.85 are 53, ...

B) Project the training set, and test sets separately using the same projection matrix.

In [ ]:
```
def reduce_dimension(projection_matrix, data):
  return np.matmul(data, projection_matrix)
```

In [ ]:
```
def calculate_projected_matrices(projection_mat, train_set, test_set):
  return reduce_dimension(projection_mat, train_set), reduce_dimension(projection_mat,
```

In [ ]:
```
projected_training0, projected_testing0 = calculate_projected_matrices(projection_matr
projected_training1, projected_testing1 = calculate_projected_matrices(projection_matr
projected_training2, projected_testing2 = calculate_projected_matrices(projection_matr
projected_training3, projected_testing3 = calculate_projected_matrices(projection_matr

print(f"projected train 0 = {projected_training0.shape}, projected test 0 = {projected
print(f"projected train 1 = {projected_training1.shape}, projected test 1 = {projected
```

```
print(f"projected train 2 = {projected_training2.shape}, projected test 2 = {projected
print(f"projected train 3 = {projected_training3.shape}, projected test 3 = {projected
```

Just making sure of the correctness of dimensions:

projected train 0 = (200, 37), projected test 0 = (200, 37)

projected train 1 = (200, 53), projected test 1 = (200, 53)

projected train 2 = (200, 77), projected test 2 = (200, 77)

projected train 3 = (200, 116), projected test 3 = (200, 116)

C) Use a simple classifier (first Nearest Neighbor to determine the class labels)

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=1)
```

In [ ]:
```
knn.fit(training_set, testing_labels)
print(knn.predict(testing_set))
```

[16 2 1 1 1 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10
10 38 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 15 15 15 16 16 16 16
16 17 17 17 17 17 18 18 18 18 18 19 19 19 19 8 20 20 3 20 20 21 21 21 21 21 22 22 22 22 22 23
23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 26 26 27 27 27 27 27 28 28 28 28 28 29 29
29 29 29 30 30 30 30 30 31 31 31 30 31 32 32 32 2 32 33 33 33 33 33 34 34 34 34 34 40 35 35
35 35 36 36 36 36 36 37 37 37 37 37 38 38 38 38 38 39 39 39 39 39 40 40 5 5 5]

D) Report Accuracy for every value of alpha separately.

In [ ]:
```
def knn1PCA(projected_training, training_labels, projected_testing, testing_labels):
  knn.fit(projected_training, training_labels)
  return knn.score(projected_testing, testing_labels)
```

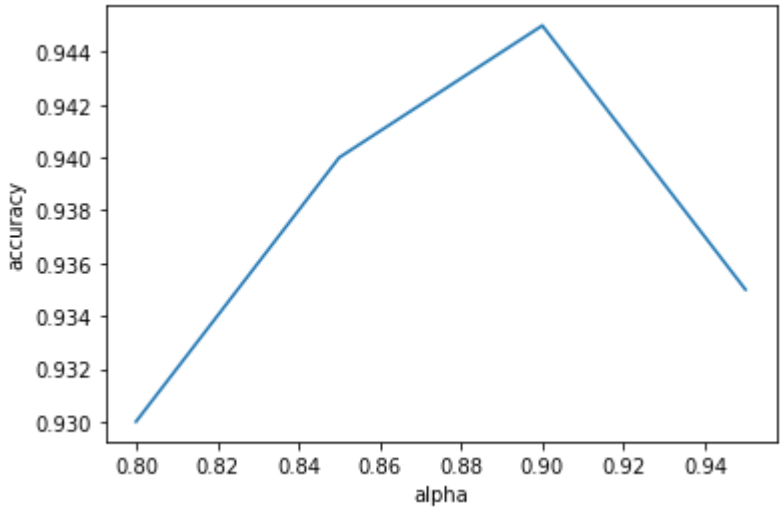
In [ ]:
```
accuracies = []
accuracies.append(knn1PCA(projected_training0, training_labels, projected_testing0, te
accuracies.append(knn1PCA(projected_training1, training_labels, projected_testing1, te
accuracies.append(knn1PCA(projected_training2, training_labels, projected_testing2, te
accuracies.append(knn1PCA(projected_training3, training_labels, projected_testing3, te

for i in range(len(accuracies)):
  print(f"alpha= {alpha[i]} -->  accuracy{i} = {accuracies[i]}")

plt.plot(np.array(alpha),accuracies)
plt.xlabel('alpha')
plt.ylabel('accuracy')
```

alpha= 0.8 --> accuracy0 = 0.93

alpha= 0.85 --> accuracy1 = 0.94

alpha= 0.9 --> accuracy2 = 0.945

alpha= 0.95 --> accuracy3 = 0.935



E) Can you find a relation between alpha and classification accuracy?

Increasing alpha should increase accuracy, but it is not a must after a certain threshold, which is here equal to 0.90 ..

Afterall, differences in accuracy are not significant at all in the range of these values for alpha (0.8 -> 0.95)

6'- Classifier Tuning:

A) Set the number of neighbors in the K-NN classifier to 1,3,5,7

B) Tie Breaking: built-in, explained in the classifier tuning section

C) Plot (or tabulate) the performance measure (accuracy) against the K value.

Apply K-NN with i neighbours for each alpha>

In [ ]:
```
def KnnPCA(projected_training, training_labels, projected_testing, testing_labels, alp
  accuracy = []
  for i in range(1,8,2):
    knn = KNeighborsClassifier(n_neighbors= i)
    knn.fit(projected_training, training_labels)
    # print(knn.predict(projected_testing0))
    currentAcc = knn.score(projected_testing, testing_labels)
    # print(currentAcc)
    accuracy.append(currentAcc)

  print(f"Alpha = {alpha}:")
  j = 0
  for i in range(1,8,2):
    print(f"for {i} neigbour -> accuracy = {round(accuracy[j],3)}")
    j+=1
  plt.plot(np.array([1,3,5,7]),accuracy)
  plt.title(f'PCA accuracy')
```

```
  plt.xlabel('neighbour')
  plt.ylabel('accuracy')
  # plt.xticks()
  # plt.yticks()
```

In [ ]:
```
KnnPCA(projected_training0, training_labels, projected_testing0, testing_labels, 0.80)
KnnPCA(projected_training1, training_labels, projected_testing1, testing_labels, 0.85)
KnnPCA(projected_training2, training_labels, projected_testing2, testing_labels, 0.90)
KnnPCA(projected_training3, training_labels, projected_testing3, testing_labels, 0.95)
```

Alpha = 0.8:

for 1 neigbour -> accuracy = 0.93 for 3 neigbour -> accuracy = 0.855 for 5 neigbour -> accuracy = 0.805 for 7 neigbour -> accuracy = 0.78

Alpha = 0.85:

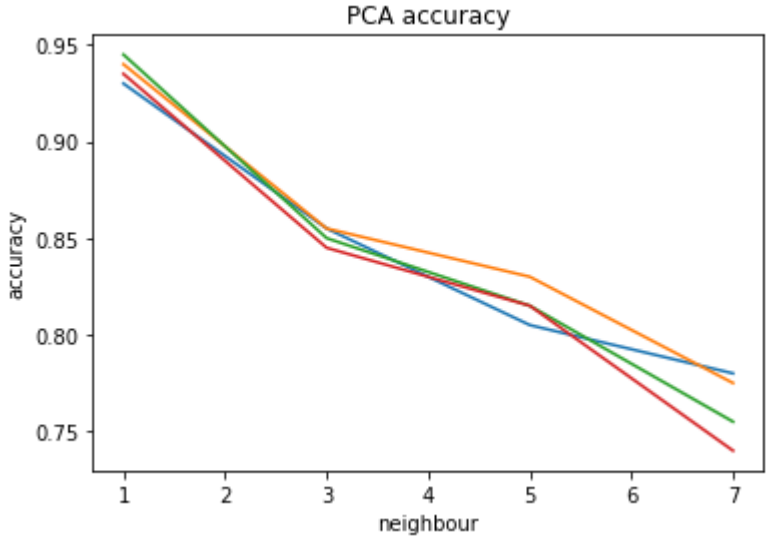
for 1 neigbour -> accuracy = 0.94 for 3 neigbour -> accuracy = 0.855 for 5 neigbour -> accuracy = 0.83 for 7 neigbour -> accuracy = 0.775

Alpha = 0.9:

for 1 neigbour -> accuracy = 0.945 for 3 neigbour -> accuracy = 0.85 for 5 neigbour -> accuracy = 0.815 for 7 neigbour -> accuracy = 0.755

Alpha = 0.95:

for 1 neigbour -> accuracy = 0.935 for 3 neigbour -> accuracy = 0.845 for 5 neigbour -> accuracy = 0.815 for 7 neigbour -> accuracy = 0.74



---

5. Classification Using LDA

LDA main function used to get projection matrix to project our data with new dimensions which is eigen vectors corresponding to greatest 40 eigen values.

In [ ]:
```
def LDA(data_matrix, labels, new_dim):

  #cast data matrix to float
  data_matrix = data_matrix.astype('float64')

  # classify our data matrix according to its corresponding class
  # class_data_matrix is 3d array array[class[images of class]]
  class_data_matrix, unique_labels = classify_data(data_matrix, labels)
  class_number = len(unique_labels)
  feature_number = data_matrix.shape[1]

  # calc mean vector for images in each class.
  class_mean_vectors = calc_mean_vectors(data_matrix, class_data_matrix, class_number,

  # calc overall mean vector for all images in train set.
  overall_mean_vector = calc_overall_sample_mean(data_matrix, feature_number)

  sb = calc_sb(class_data_matrix, class_mean_vectors, overall_mean_vector, class_numbe

  centered_class_data_matrix = center_data_matrix(class_data_matrix, class_mean_vector
  print('centered data matrix:')
  print(centered_class_data_matrix.shape)
  print(centered_class_data_matrix)

  s = calc_s(centered_class_data_matrix, class_mean_vectors, feature_number)

  lambd, vectors = np.linalg.eigh(np.dot(np.linalg.inv(s),sb))
  print('Eigens')
  print('Eigen values:')
  print(lambd.shape)
  print(lambd)
  print('Eigen vectors:')
  print(vectors.shape)
  print(vectors)

  # pick 39 eigen vectors corresponding to max 39 eigen values
  vectors = vectors.T
  projection_matrix = vectors[-new_dim:]
  print('Projection shape:')
  print(projection_matrix.shape)
  print('Projection Matrix:')
  print(projection_matrix)


  return projection_matrix, class_mean_vectors
```

We classify faces according to its class so all images of same face are grouped together.
class i -> all images of face i:

in this function , we assume that data_matrix and labels should be sorted as classes and labels flows to funtion in descending order

In [ ]:
```
# classify data
def classify_data(data_matrix,labels):
  unique_labels, s = np.unique(labels, return_index=True)
  class_data_matrix = np.asarray(np.split(data_matrix,s[1:]))
  print('classified data matrix:')
  print(class_data_matrix.shape)
  print(class_data_matrix)
  print('unique labels:')
  print(unique_labels)
  return class_data_matrix, unique_labels
```

We calculate mean vector for each class.

In [ ]:
```
def calc_mean_vectors(data_matrix, class_data_matrix, class_number, feature_number):
  class_mean_vectors = np.zeros((class_number, feature_number))
  for i in range(class_number):
    class_mean_vectors[i] = np.mean(class_data_matrix[i], axis = 0)
  print('Mean Vectors for each class:')
  print(class_mean_vectors.shape)
  print(class_mean_vectors)
  return class_mean_vectors
```

We calculate the overall mean vector for the whole data set samples.

In [ ]:
```
def calc_overall_sample_mean(data_matrix, feature_number):
  overall_mean_vector = np.mean(data_matrix, axis = 0).reshape(1,feature_number)
  print('overall samples mean vector:')
  print(overall_mean_vector.shape)
  print(overall_mean_vector)
  return overall_mean_vector
```

We calculate classes scatter matrix sb.
$Sb=\Sigma nk(\mu k-\mu)(\mu k-\mu)T$.

In [ ]:
```
def calc_sb(class_data_matrix, class_mean_vectors, overall_mean_vector, class_number,
  sb = np.zeros((feature_number, feature_number))
  for i in range(class_number):
    sb += ( len(class_data_matrix[i]) * np.dot((class_mean_vectors[i] - overall_mean_v
  print('SB:')
  print(sb.shape)
  print(sb)
  return sb
```

We centerize our data matrix. Zi = Di - μi.

In [ ]:
```
def center_data_matrix(class_data_matrix,class_mean_vectors):
  for i in range(len(class_data_matrix)):
    class_data_matrix[i] -= class_mean_vectors[i]
  return class_data_matrix
```

we calculate scatter matrix S = ΣSi where Si = (Zi.T)(Zi).

In [ ]:
```
def calc_s(centered_class_data_matrix, class_mean_vectors, feature_number):
  s = np.zeros((feature_number, feature_number))
  for i in range(len(centered_class_data_matrix)):
    s += (np.dot(centered_class_data_matrix[i].T, centered_class_data_matrix[i]))
  print('S:')
  print(s)
  print(s.shape)
  return s
```

K-NN Classifier for LDA

Get projection matrix to transfor to our new dimension.

In [ ]:
```
projection_matrix, mean_vectors = LDA(training_set, training_labels,39)
print(projection_matrix.shape)
print(projection_matrix)
```

Next pictures are snippets of the results of the above function (LDA)

```
classified data matrix:
(40, 5, 10304)
[[[ 60.  60.  62. ...  32.  34.  34.]
  [ 63.  53.  35. ...  41.  10.  24.]
  [ 43.  50.  41. ... 158. 153. 169.]
  [ 44.  43.  32. ...  43.  43.  37.]
  [ 34.  34.  33. ...  37.  40.  33.]]

 [[ 37.  35.  35. ...  26.  28.  28.]
  [ 34.  36.  35. ...  32.  25.  25.]
  [ 34.  35.  35. ...  31.  24.  24.]
  [ 34.  39.  35. ... 135. 138. 148.]
  [ 37.  34.  38. ... 135. 143. 133.]]

 [[100.  99. 108. ...  44.  42.  44.]
  [105. 102. 108. ...  42.  35.  37.]
  [110. 106. 109. ...  49.  53.  53.]
  [107. 104. 109. ...  58.  56.  57.]
  [104. 109. 103. ...  57.  56.  59.]]

  ...
```

```
unique labels:
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40]
Mean Vectors for each class:
(40, 10304)
[[ 48.8  48.   40.6 ...  62.2  56.   59.4]
 [ 35.2  35.8  35.6 ...  71.8  71.6  71.6]
 [105.2 104.  107.4 ...  50.   48.4  50. ]
 ...
 [109.8 108.6 109.6 ...  68.2  68.   65. ]
 [ 84.6  83.4  85.2 ... 115.8 119.8 117.4]
 [126.  123.6 125.8 ...  47.8  46.   49. ]]
overall samples mean vector:
(1, 10304)
[[85.12  84.89  85.165 ... 77.24  74.335 73.37 ]]
SB:
(10304, 10304)
[[235278.32  232717.44  233560.64  ... -43174.56  -21703.44  -15646.88 ]
 [232717.44  230668.78  231142.03  ... -40923.52  -20191.43  -14642.46 ]
 [233560.64  231142.03  232523.155 ... -39926.72  -18941.855 -13273.41 ]
 ...
 [-43174.56  -40923.52  -39926.72  ... 360976.48  315665.12  312321.04 ]
 [-21703.44  -20191.43  -18941.855 ... 315665.12  294892.955 289157.81 ]
 [-15646.88  -14642.46  -13273.41  ... 312321.04  289157.81  289627.82 ]]
```

```
centered data matrix:
(40, 5, 10304)
[[[ 11.2  12.   21.4 ... -30.2 -22.  -25.4]
  [ 14.2   5.   -5.6 ... -21.2 -46.  -35.4]
  [ -5.8   2.    0.4 ...  95.8  97.  109.6]
  [ -4.8  -5.   -8.6 ... -19.2 -13.  -22.4]
  [-14.8 -14.   -7.6 ... -25.2 -16.  -26.4]]

 [[  1.8  -0.8  -0.6 ... -45.8 -43.6 -43.6]
  [ -1.2   0.2  -0.6 ... -39.8 -46.6 -46.6]
  [ -1.2  -0.8  -0.6 ... -40.8 -47.6 -47.6]
  [ -1.2   3.2  -0.6 ...  63.2  66.4  76.4]
  [  1.8  -1.8   2.4 ...  63.2  71.4  61.4]]

 [[ -5.2  -5.    0.6 ...  -6.   -6.4  -6. ]
  [ -0.2  -2.    0.6 ...  -8.  -13.4 -13. ]
  [  4.8   2.    1.6 ...  -1.    4.6   3. ]
  [  1.8   0.    1.6 ...   8.    7.6   7. ]
  [ -1.2   5.   -4.4 ...   7.    7.6   9. ]]

 ...
```

```
S:
[[ 28496.8  27128.2  27372.4 ...  -3977.2  -3511.6    539. ]
 [ 27128.2  28448.8  27518.6 ...  -2697.2  -2349.2   1781.6]
 [ 27372.4  27518.6  28698.4 ...  -3472.2   -923.2   3416.2]
 ...
 [ -3977.2  -2697.2  -3472.2 ... 140034.   83425.8  64222.2]
 [ -3511.6  -2349.2   -923.2 ...  83425.8  99585.6  82754.4]
 [   539.    1781.6   3416.2 ...  64222.2  82754.4 104148.8]]
(10304, 10304)
```

```
Eigens
Eigen values:
(10304,)
[-1.13363577e+19 -5.67599743e+18 -3.60393996e+18 ...  3.64265939e+18
  5.62258133e+18  1.13809605e+19]
Eigen vectors:
(10304, 10304)
[[ 8.48040330e-03 -3.60603806e-03  2.10800516e-02 ... -2.11763471e-02
   3.40539171e-03 -8.32129618e-03]
 [ 8.37921947e-03 -3.19003732e-03  2.08637196e-02 ... -2.06794918e-02
   2.95499383e-03 -8.18635779e-03]
 [ 8.17594710e-03 -3.94777442e-03  2.08026492e-02 ... -2.09376584e-02
   3.75698090e-03 -8.02581305e-03]
 ...
 [ 9.48521935e-03 -5.38139223e-03 -4.67369325e-03 ... -3.72057109e-03
  -5.31315225e-03  9.53644578e-03]
 [ 6.94781897e-03 -1.06247449e-02 -4.96597235e-03 ... -4.95545765e-03
  -1.05509058e-02  7.02968690e-03]
 [-2.41868614e-03  9.39328027e-05 -2.33249788e-05 ... -5.59884875e-04
   1.17005530e-04 -2.46233210e-03]]
Projection shape:
(39, 10304)
Projection Matrix:
[[-0.00699197 -0.00655511 -0.00722016 ... -0.00035135 -0.00467179
  -0.00857671]
 [ 0.03496077  0.03178627  0.03518832 ...  0.00170638  0.00287306
  -0.00456613]
 [-0.00378704 -0.00184453 -0.0044919  ... -0.00630928 -0.00392258
  -0.01036979]
```

```
(39, 10304)
```

Project train & test data sets.

In [ ]:
```
projected_train_data = np.dot(training_set, projection_matrix.T)
projected_test_data = np.dot(testing_set, projection_matrix.T)
```

Apply K-NN with i neighbours.

In [ ]:
```
accuracy = []
for i in range(1,12,2):
  knn = KNeighborsClassifier(n_neighbors= i)
  knn.fit(projected_train_data, training_labels)
  print(knn.predict(projected_test_data))
  acc = knn.score(projected_test_data, testing_labels)
  print(acc)
  accuracy.append(acc)
```

[16 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 15 15 15 16 16 16 16 16 17 17 17 17 17 18 18 18 18 18 19 19 19 19 11 20 20 4 20 20 21 21 21 21 21 22 22 22 22 22 23 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 26 26 27 27 27 27 27 28 28 28 28 28 29 29 29 29 39 30 30 30 30 30 31 31 31 31 31 32 32 32 32 32 33 33 33 33 33 34 34 34 34 34 40 15 35 35 35 7 36 7 36 36 37 37 37 37 37 38 38 38 38 23 39 39 39 39 39 40 40 40 5 5]

0.945

[16 1 1 1 1 2 2 2 2 2 3 3 25 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 15 15 15 15 16 1 16 16 4 17 4 17 17 18 18 18 18 18 19 19 19 19 11 20 20 4 20 20 21 21 21 21 21 22 22 22 22 22 38 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 25 26 27 27 27 27 27 28 28 28 28 28 29 29 29 29 29 30 30 30 30 30 21 31 31 21 21 32 32 32 32 32 33 33 33 33 33 34 34 34 34 34 40 15 18 35 13 7 36 7 36 36 28 28 37 28 28 38 38 38 38 38 39 39 39 39 39 40 40 5 5 5]

0.87

[16 1 1 1 1 2 2 2 2 2 3 3 25 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 14 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 15 15 15 16 16 1 16 16 4 17 16 17 17 18 18 18 18 18 19 19 19 19 27 20 20 4 20 23 21 21 21 21 21 22 22 22 22 22 38 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 26 26 27 27 27 11 27 28 28 19 19 28 29 29 29 29 29 30 30 30 30 30 21 31 31 21 21 32 32 32 15 32 33 33 33 33 33 34 34 34 34 34 40 15 5 25 13 7 36 7 36 36 28 28 37 28 28 38 38 38 38 38 39 39 39 29 39 40 40 18 5 5]

0.84

[16 1 1 1 1 2 2 2 2 2 3 3 25 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 14 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 14 15 15 15 35 15 15 16 1 16 16 4 17 4 17 17 18 18 18 18 18 19 19 19 19 27 20 20 4 20 23 21 21 21 21 21 22 22 22 22 22 38 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 13 26 19 19 27 11 27 28 28 37 19 28 29 29 29 29 29 30 30 30 30 30 21 34 34 21 21 32 32 32 2 32 33 33 20 33 20 34 34 34 34 34 40 15 18 25 13 7 36 7 36 36 28 28 37 26 28 38 38 38 38 38 22 39 39 39 39 40 26 18 5 5]

0.79

[16 1 1 1 1 2 2 2 2 2 3 3 25 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 11 11 11 12 12 12 12 12 13 13 13 13 18 14 14 14 14 11 15 2 15 35 15 24 16 1 16 16 16 17 4 17 17 18 18 18 18 18 19 19 19 19 27 20 20 4 20 23 21 21 21 21 21 22 22 22 22 22 38 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 13 26 19 19 2 11 27 37 37 37 19 19 29 29 29 29 29 30 30 30 30 30 21 34 34 21 21 32 32 32 2 32 33 33 33 33 20 34 34 34 34 34 40 15 13 25 13 7 7 7 36 36 26 13 37 26 13 38 38 38 38 38 39 22 39 39 39 40 5 18 5 5]

0.76

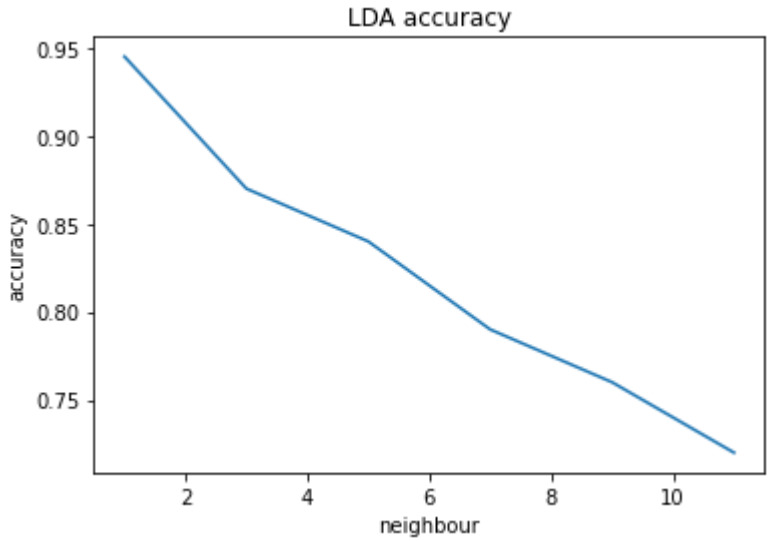
[16 2 1 1 1 2 2 2 2 2 3 3 25 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 10 10 10 10 10 11 11 19 11 11 12 12 12 12 12 13 13 13 13 13 14 14 14 14 11 15 2 15 35 15 24 16 1 16 16 4 17 4 17 17 18 18 18 18 18 19 19 19 19 27 20 20 4 20 23 21 21 21 21 21 22 22 22 22 22 38 23 23 23 23 24 24 24 24 24 25 25 25 25 25 26 26 26 13 26 2 19 2 11 27 37 37 19 19 27 29 29 29 29 29 21 30 30 30 21 21 34 34 21 21 32 32 32 2 2 33 33 33 33 20 34 34 34 34 22 40 15 13 25 13 7 7 7 36 17 26 13 14 26 13 38 38 38 38 38 22 22 39 39 39 40 5 18 5 18]

0.72

Plot LDA accuracy with different neighbours classifier:

In [ ]:
```
for i in range(len(accuracy)):
  print(f"for {i} neigbour -> accuracy = {accuracy[i]}")
plt.plot(np.array([1,3,5,7,9,11]),accuracy)
plt.title('LDA accuracy')
plt.xlabel('neighbour')
plt.ylabel('accuracy')
# plt.xticks()
# plt.yticks()
```

for 0 neigbour -> accuracy = 0.945 for 1 neigbour -> accuracy = 0.87 for 2 neigbour -> accuracy = 0.84 for 3 neigbour -> accuracy = 0.79 for 4 neigbour -> accuracy = 0.76 for 5 neigbour -> accuracy = 0.72



Comments: Here, accuracy is inverse proportional with the neighous number

---

6. Classifier Tuning

Done above, herewe just comment.

A) Set the number of neighbors in the K-NN classifier to 1,3,5,7

For PCA, this was done for these 4 values for each value of alpha, so the total runs are 4*4 = 16.

For LDA, this was done for K-NN = 1, 3, 5, 7, 9, 11

B) Tie breaking at your preferred strategy.

Sickit K-NN classifier resolves ties by choosing the class that appears first in the set of neighbors.

C) Plot (or tabulate) the performance measure (accuracy) against the K value. This is to be done for PCA and LDA as well.

All code and results of this section are done at the end of each algorithm's section: PCA in section 4, LDA in section 5.

---

7. Compare vs Non-Face Images

load face data

In [ ]:
```
!pwd
os.chdir("..")
!pwd
```

```
/content/drive/MyDrive/face_recognition
/content/drive/MyDrive
```

In [ ]:
```
os.chdir("face_recognition")
```

In [ ]:
```
data_face = []
for i in range(1,41):
  os.chdir("s"+str(i))
  for j in range(1,11):
    img = cv2.imread(str(j) + ".pgm",-1)
    data_face.append(list(img.flat))
  os.chdir("..")
```

In [ ]:
```
data_face = np.array(data_face)
```

In [ ]:
```
print(data_face.shape)
```

```
(400, 10304)
```

(400, 10304)

In [ ]:
```
!pwd
!ls
os.chdir("..")
```

```
/content/drive/MyDrive/face_recognition
1PzB-dTdr2muezPn6LgPi1PK-hAsxdKNp  s12  s17  s21  s26  s30  s35  s4   s8
face_recognition.zip               s13  s18  s22  s27  s31  s36  s40  s9
s1                                 s14  s19  s23  s28  s32  s37  s5   view
s10                                s15  s2   s24  s29  s33  s38  s6
s11                                s16  s20  s25  s3   s34  s39  s7
```

In [ ]:
```
from pathlib import Path
entries = Path('nonface_images/')
data_nonface = []
for entry in entries.iterdir():
  img = cv2.imread("nonface_images/"+str(entry.name),0)
  img = cv2.resize(img, (92,112))
  data_nonface.append(list(img.flat))
```

In [ ]:
```
data_nonface = np.array(data_nonface)
```

In [ ]:
```
index = 154        # feel free to change from 0 to 484
plt.imshow(data_nonface[index].reshape(112,92))
```

In [ ]:
```
print(np.mean(data_nonface,axis = 0))
```

```
[103.41443299 104.5628866 106.76701031 ... 89.83917526 89.18969072 89.53814433]
```

prepare data and labels of our new classification problem

In [ ]:
```
def get_trian_and_test(data_face, data_nonface, number):
  print(data_face.shape)
  print(data_nonface.shape)
```

```
  data_face_label = np.ones((len(data_face),1))

  X_face_train, X_face_test, y_face_train, y_face_test = train_test_split(data_face, d

  data_nonface_label = np.zeros((len(data_nonface),1))
  X_nonface_train, X_nonface_test, y_nonface_train, y_nonface_test = train_test_split(

  X_train = np.concatenate(( X_nonface_train, X_face_train))
  X_test = np.concatenate((X_nonface_test, X_face_test))

  y_train = np.concatenate((y_nonface_train, y_face_train))
  y_test = np.concatenate(( y_nonface_test, y_face_test))

  print(data_face_label.shape)
  print(data_nonface_label.shape)
  print(y.shape)
  return X_train, X_test, y_train, y_test
```

train and test data

In [ ]:
```
def nonface_classifier(X_train, X_test, y_train, y_test):
  scaler = StandardScaler()
  projection_data_matrix, _ = LDA(X_train,y_train, 1)
  projected_X_train = np.dot(X_train, projection_data_matrix.T)
  projected_X_test = np.dot(X_test, projection_data_matrix.T)
  print("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  acc = []
  neighbours = [1,2,4,7,10,12]
  for i in neighbours:
    knn = KNeighborsClassifier(n_neighbors= i)
    knn.fit(projected_X_train, y_train)
    acc.append(knn.score(projected_X_test, y_test))
  print("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  max_value = max(acc)
  index = acc.index(max_value)

  knn = KNeighborsClassifier(n_neighbors= neighbours[index])
  knn.fit(projected_X_train, y_train)
  print("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

  # X_test = scaler.transform(projected_X_test)
  predictions = knn.predict(projected_X_test)
  print("\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  good_prediction = []
  bad_prediction = []
  for input, prediction, label in zip(X_test, predictions, y_test):
    if prediction != label and len(bad_prediction)!=5:
      bad_prediction.append(input)
    elif prediction == label and len(good_prediction)!=5:
      good_prediction.append(input)
    if len(good_prediction) == 5 and len(bad_prediction) == 5:
      break

  return max_value,good_prediction,bad_prediction
```

tune with number of non faces to values : 80, 160, 240, 320, 400, 485

In [ ]:
```
variant_number = [80, 160, 240, 320, 400, 485]
accuracy = []
good_cases = []
bad_cases = []
for i in variant_number:
  X_train, X_test, y_train, y_test = get_trian_and_test(data_face, data_nonface, i)
  a, good, bad = nonface_classifier(X_train, X_test, y_train, y_test)
  accuracy.append(a)
  good_cases.append(good)
  bad_cases.append(bad)
plt.plot(variant_number,accuracy)
```

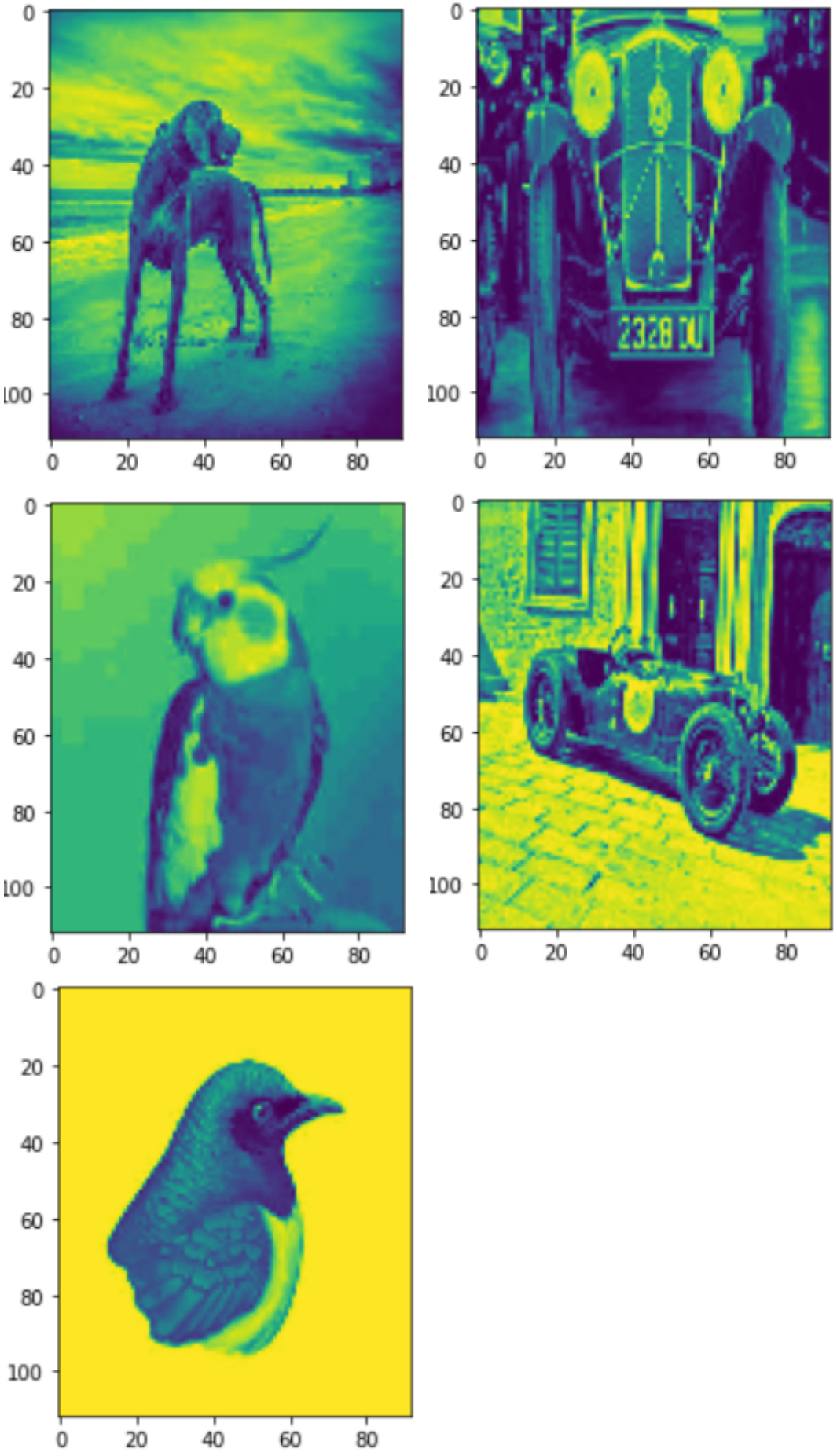
In [ ]:
```
print(accuracy)
plt.plot(variant_number, accuracy)
```

[0.9182389937106918, 0.8702702702702703, 0.8773584905660378, 0.7310924369747899, 0.7272727272727273, 0.7815699658703071]

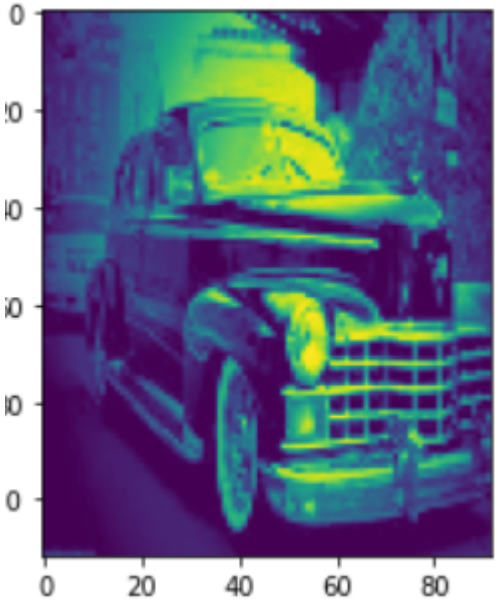
In [ ]:
```
good_cases = np.array(good_cases)
bad_cases = np.array(bad_cases)
```

number of non-face images = 80

success cases

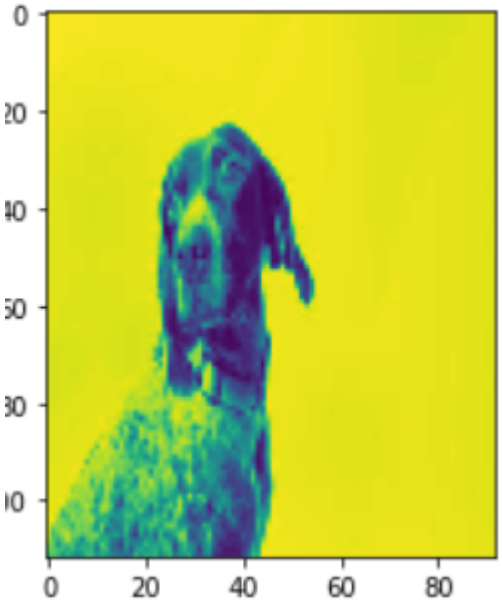
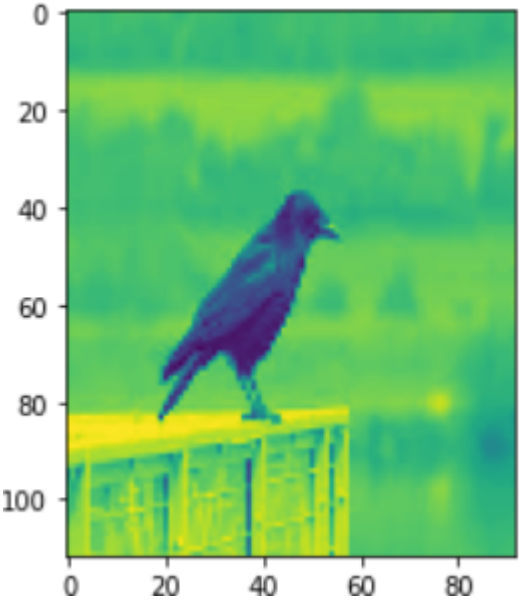
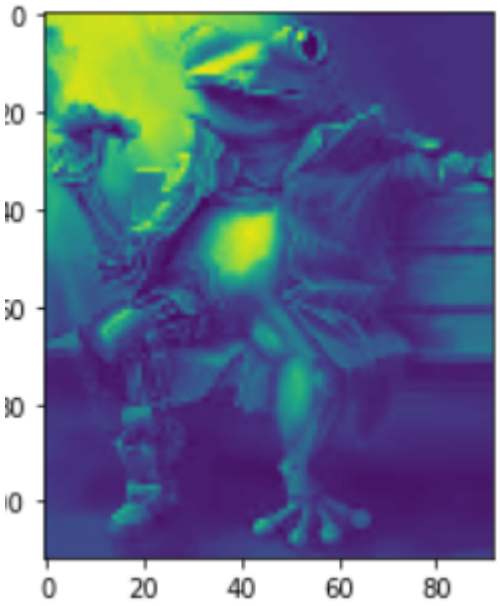
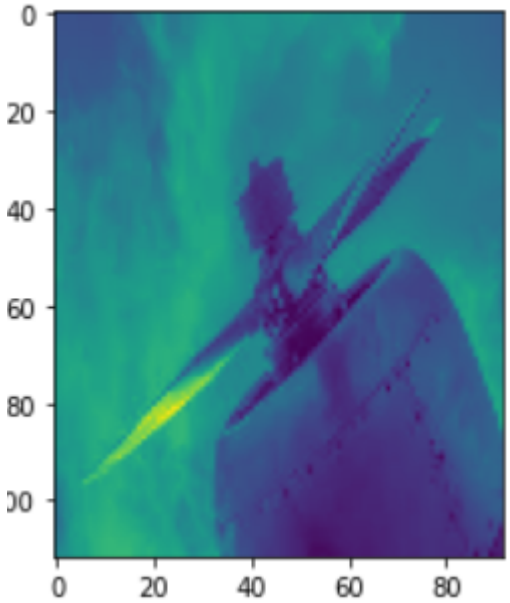
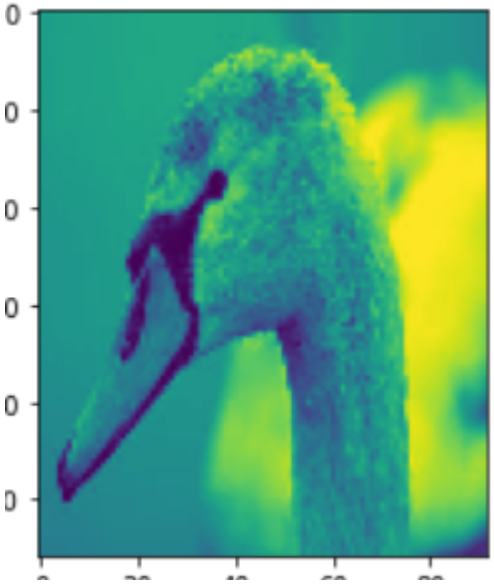


failure cases

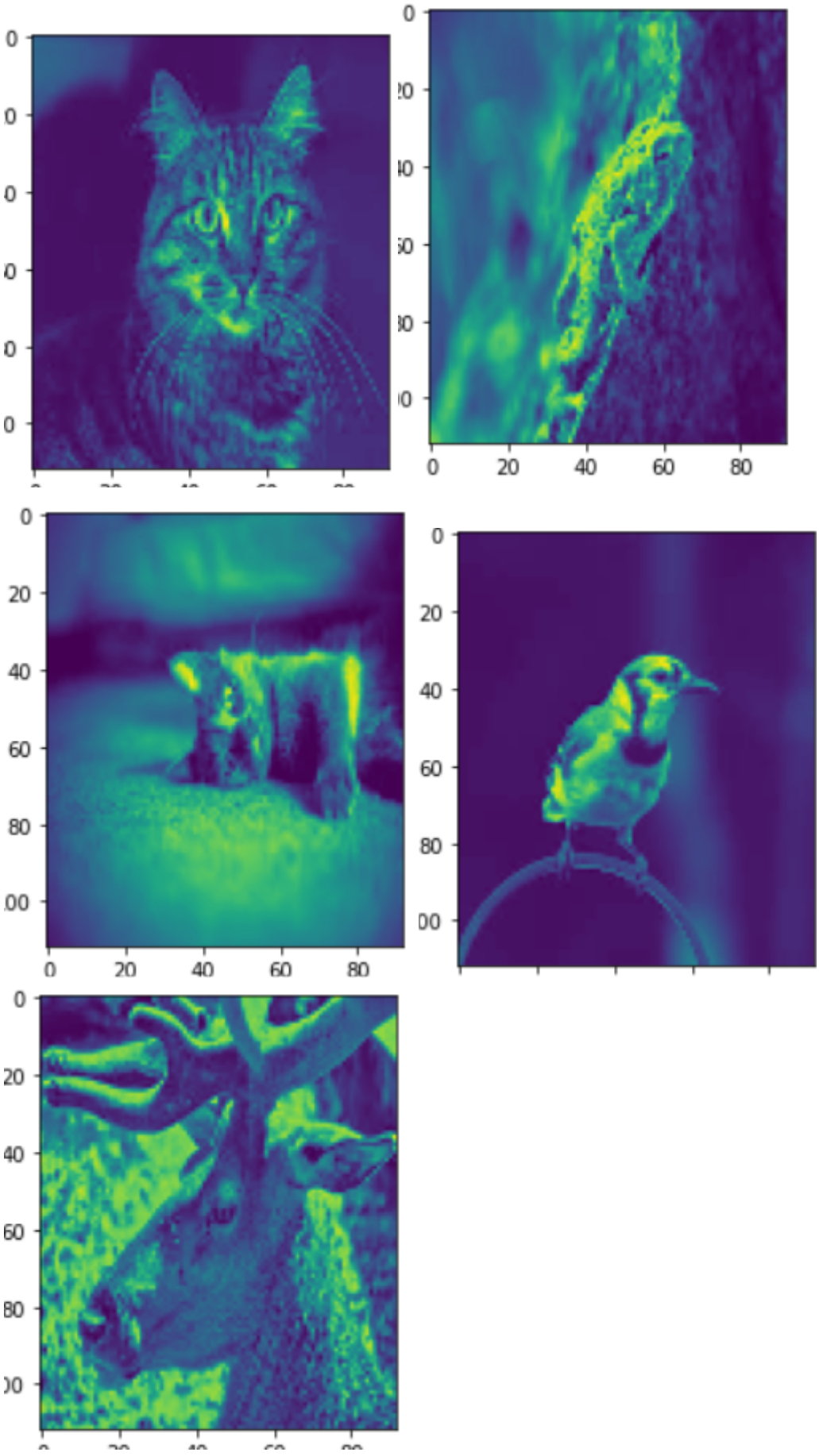


number of non-face images = 160

success cases

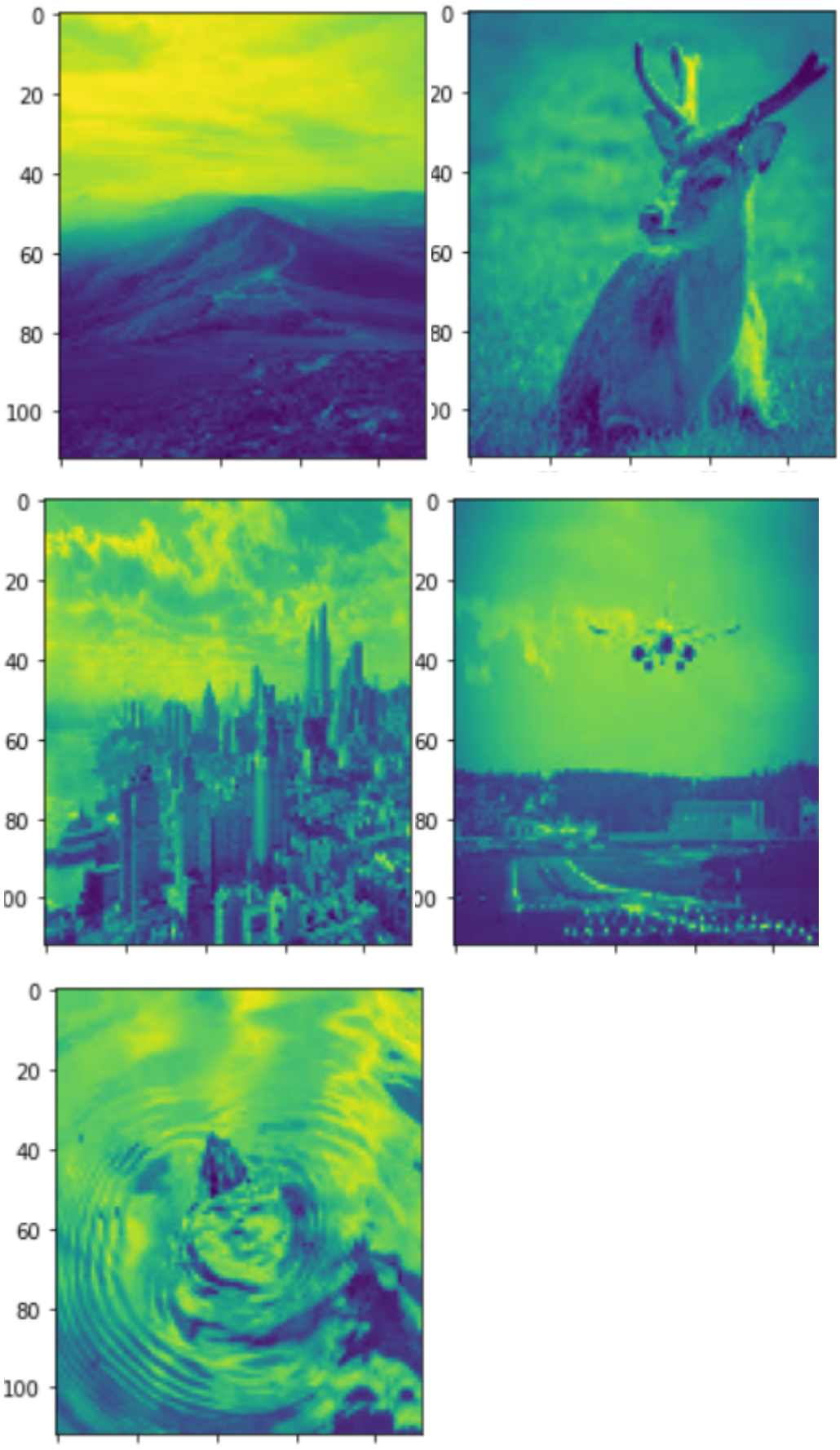






failure cases

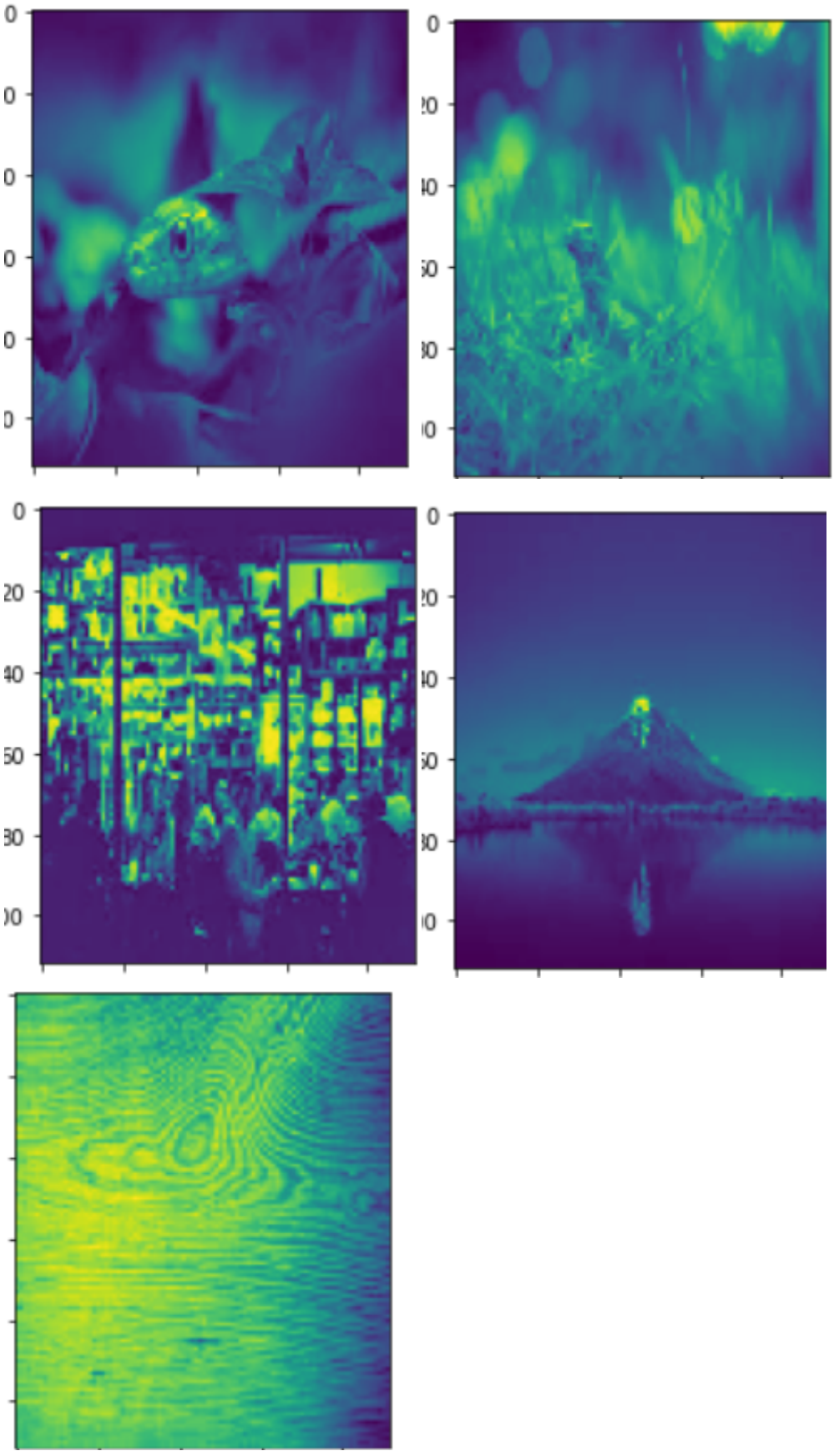


number of non-face images = 240

success cases

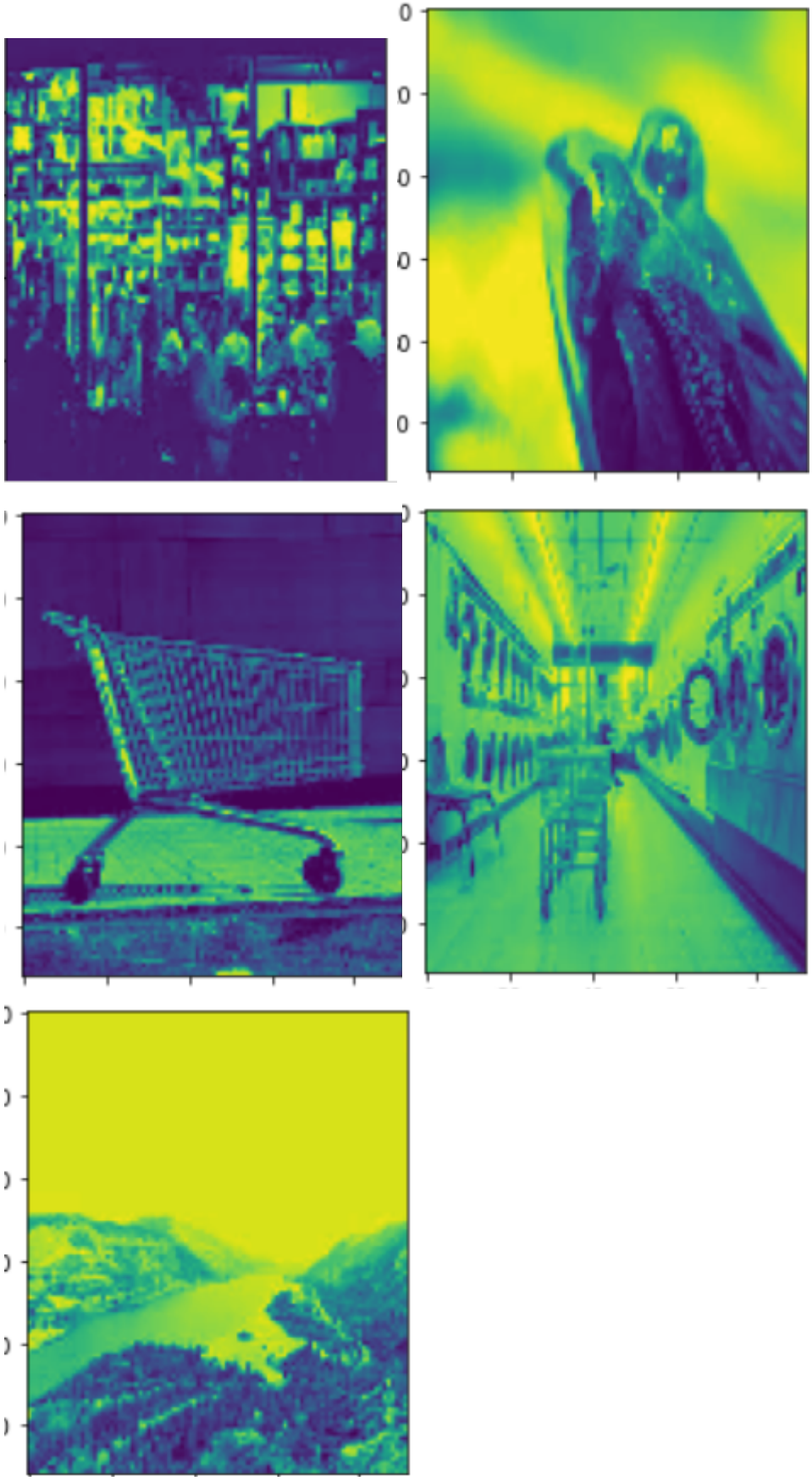


faluire cases

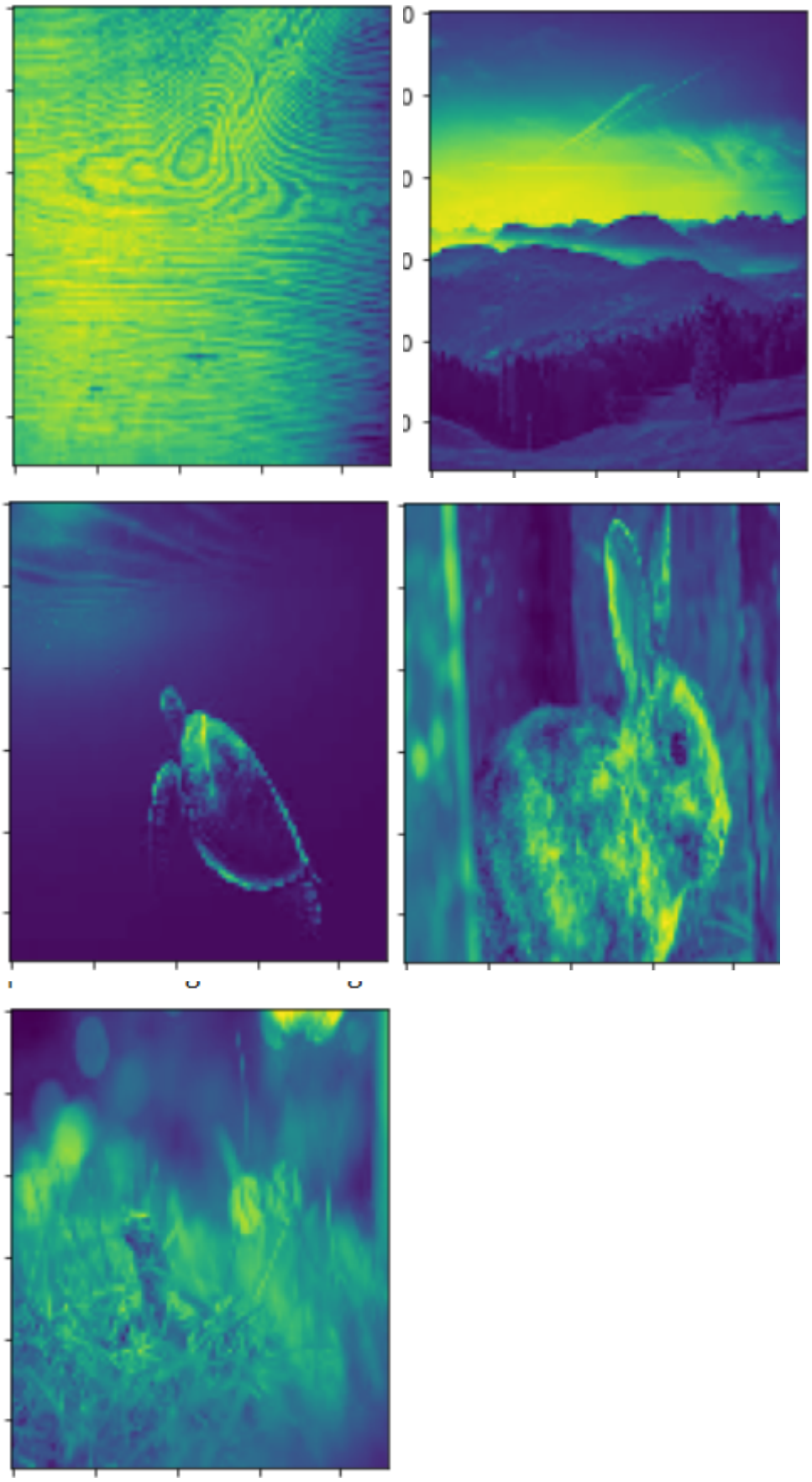


number of non-face images = 320

success cases

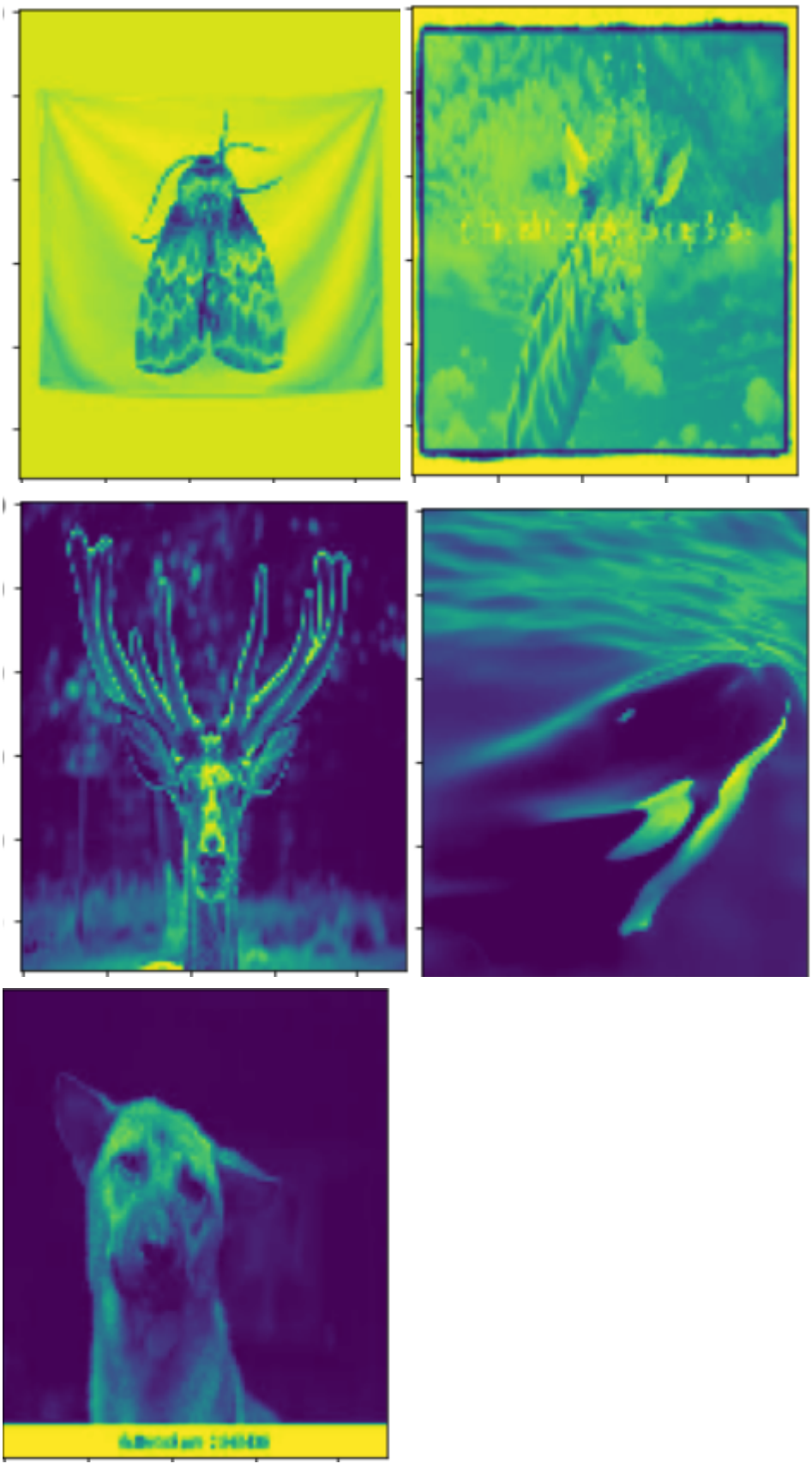


failure cases

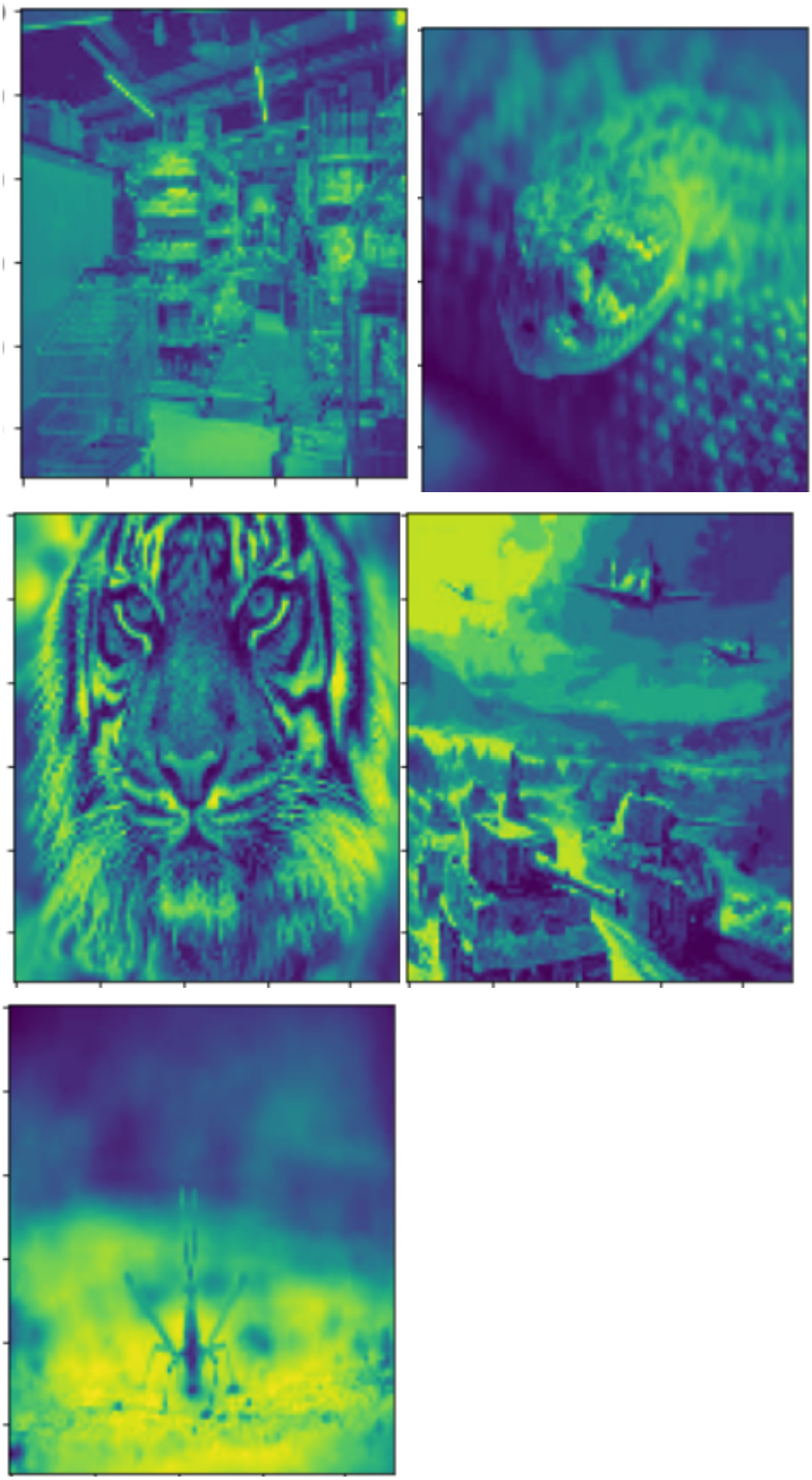


number of non-face images = 400

success cases

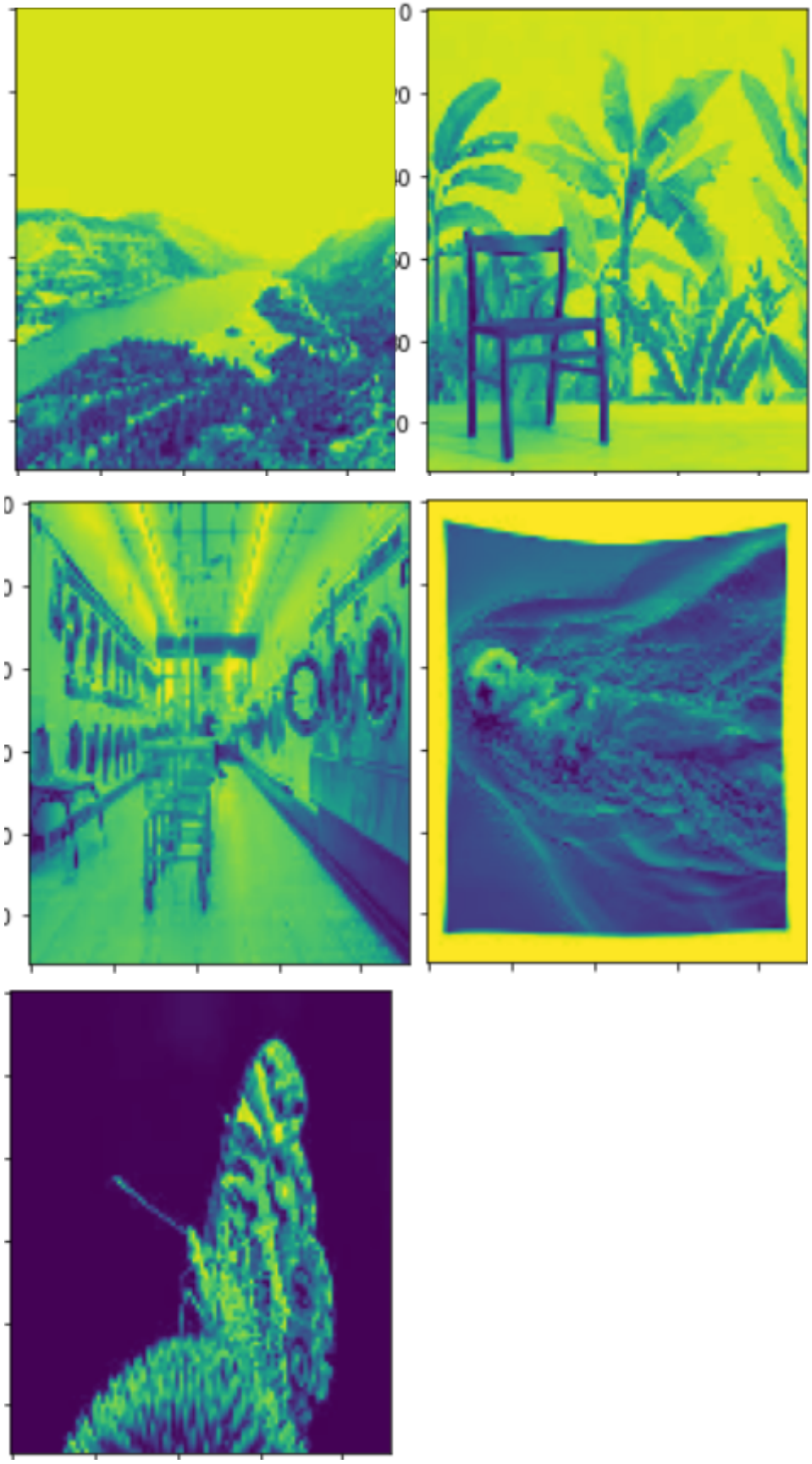


failure cases

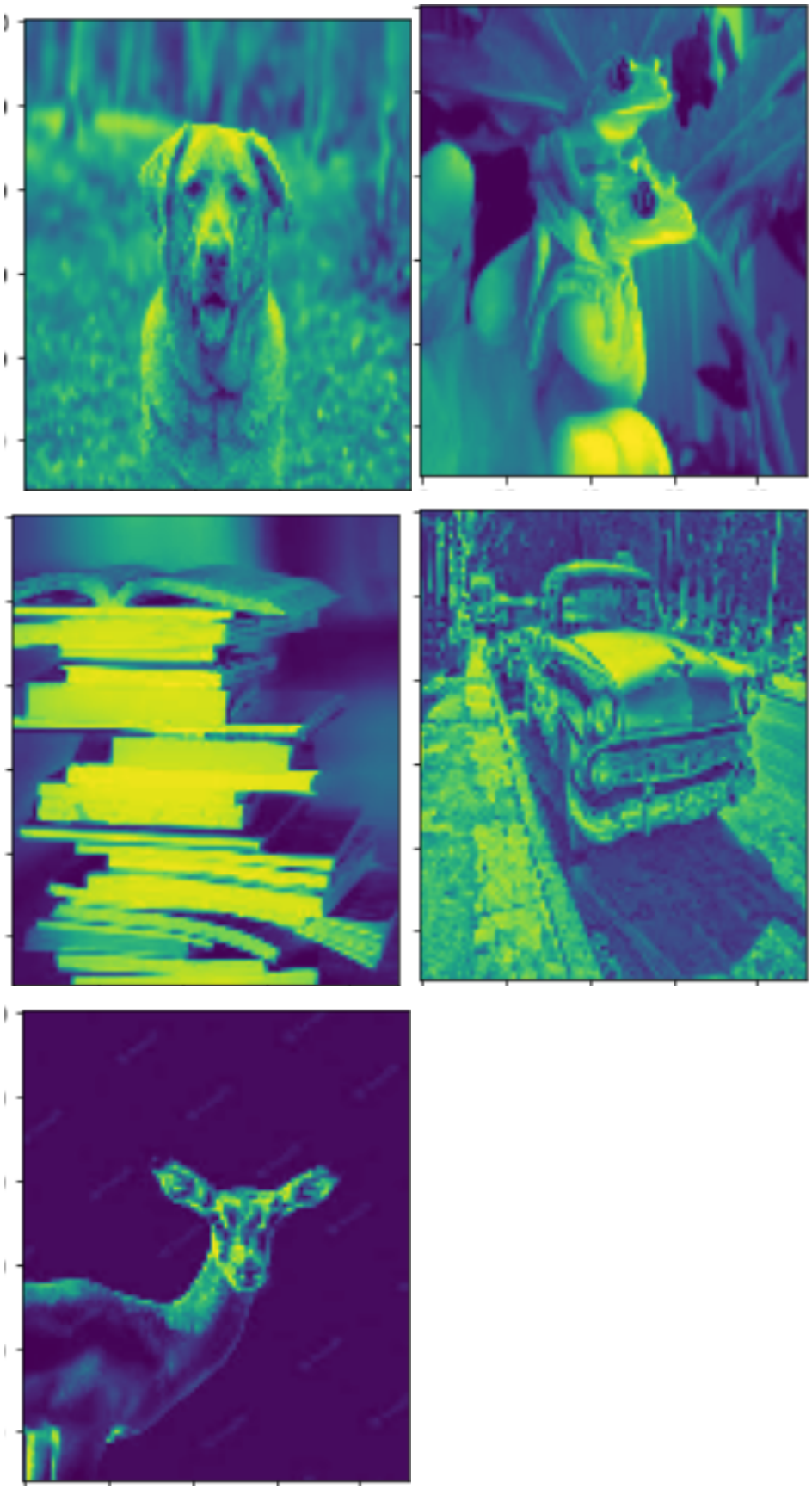


number of non-face images = 485

success cases

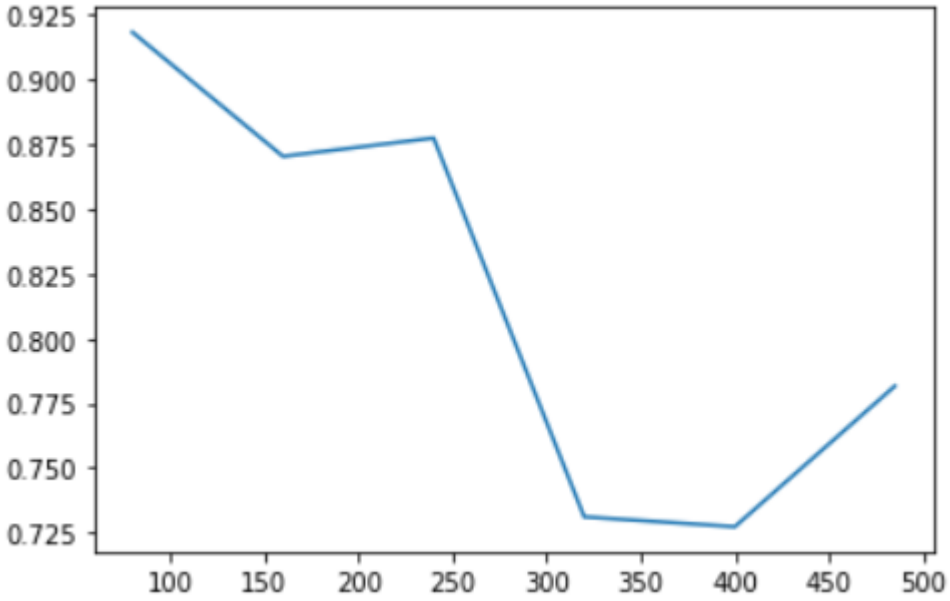


failure cases



In [ ]:
```
plt.imshow(bad_cases[5][4].reshape(112,92))
#plt.imshow(good_cases[5][4].reshape(112,92))
```

tune with number of non faces to values : 80, 160, 240, 320, 400, 485



How many dominant eigenvectors will you use for the LDA solution?

- we use 1 dominant eignvector as we classify between 2 classes

Criticize the accuracy measure for large numbers of non-faces images in the training data: - when we use large number of non-face that equivalent to number of faces, accuracy was not good enough but for small number of non-face , accuracy is better.

```
- we also note that we maximize number of non-faces to be larger than face images, accuracy increases
```

---

8. Bonus

a) Use different Training and Test splits. Change the number of instances per subject to be 7 and keep 3 instances per subject for testing. compare the results you have with the ones you got earlier with 50% split.

In [ ]:
```
reshaped_data = data.reshape((-1, 10, data.shape[1]))
# Extracting the first 7 rows from each group of 10
bonus_training_set = reshaped_data[:, :7, :].reshape((-1, data.shape[1]))
# Extracting the remaining 3 rows from each group of 10
bonus_testing_set = reshaped_data[:, 7:, :].reshape((-1, data.shape[1]))

reshaped_labels = label.reshape((40, 10))

bonus_training_labels = reshaped_labels[:, :7].flatten()
bonus_testing_labels = reshaped_labels[:, 7:].flatten()


print(bonus_training_set.shape)
print(bonus_training_labels.shape)
```

```
print(bonus_testing_set.shape)
print(bonus_testing_labels.shape)
```

(280, 10304) (280,) (120, 10304) (120,)

PCA

In [ ]:
```
alpha = [0.8, 0.85, 0.9, 0.95]
values, vectors = PCA_impl_repetetive(bonus_training_set)
bonus_projection_matrices = []
for i in range(len(alpha)):
  bonus_projection_matrices.append(PCA_impl(alpha[i], values, vectors))
```

In [ ]:
```
bonus_projected_training0, bonus_projected_testing0 = calculate_projected_matrices(bor
bonus_projected_training1, bonus_projected_testing1 = calculate_projected_matrices(bor
bonus_projected_training2, bonus_projected_testing2 = calculate_projected_matrices(bor
bonus_projected_training3, bonus_projected_testing3 = calculate_projected_matrices(bor
```

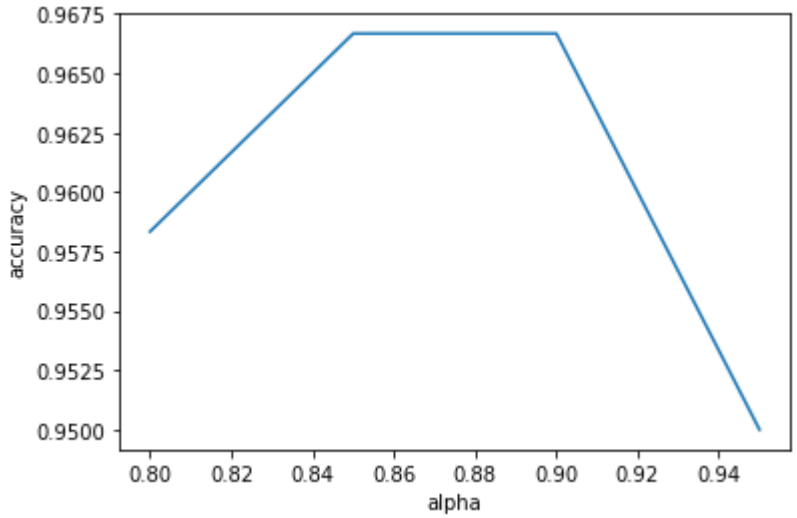
In [ ]:
```
accuracies = []
accuracies.append(knn1PCA(bonus_projected_training0, bonus_training_labels, bonus_proj
accuracies.append(knn1PCA(bonus_projected_training1, bonus_training_labels, bonus_proj
accuracies.append(knn1PCA(bonus_projected_training2, bonus_training_labels, bonus_proj
accuracies.append(knn1PCA(bonus_projected_training3, bonus_training_labels, bonus_proj

for i in range(len(accuracies)):
  print(f"alpha= {alpha[i]} -->  accuracy{i} = {round(accuracies[i],3)}")

plt.plot(np.array(alpha),accuracies)
plt.xlabel('alpha')
plt.ylabel('accuracy')
```

alpha= 0.8 --> accuracy0 = 0.958

alpha= 0.85 --> accuracy1 = 0.967

alpha= 0.9 --> accuracy2 = 0.967

alpha= 0.95 --> accuracy3 = 0.95



In [ ]:
```
KnnPCA(bonus_projected_training0, bonus_training_labels, bonus_projected_testing0, bor
KnnPCA(bonus_projected_training1, bonus_training_labels, bonus_projected_testing1, bor
KnnPCA(bonus_projected_training2, bonus_training_labels, bonus_projected_testing2, bor
KnnPCA(bonus_projected_training3, bonus_training_labels, bonus_projected_testing3, bor
```

Alpha = 0.8:

for 1 neigbour -> accuracy = 0.958

for 3 neigbour -> accuracy = 0.925

for 5 neigbour -> accuracy = 0.908

for 7 neigbour -> accuracy = 0.842

Alpha = 0.85:

for 1 neigbour -> accuracy = 0.967

for 3 neigbour -> accuracy = 0.933

for 5 neigbour -> accuracy = 0.908

for 7 neigbour -> accuracy = 0.842

Alpha = 0.9:

for 1 neigbour -> accuracy = 0.967

for 3 neigbour -> accuracy = 0.917

for 5 neigbour -> accuracy = 0.892

for 7 neigbour -> accuracy = 0.85

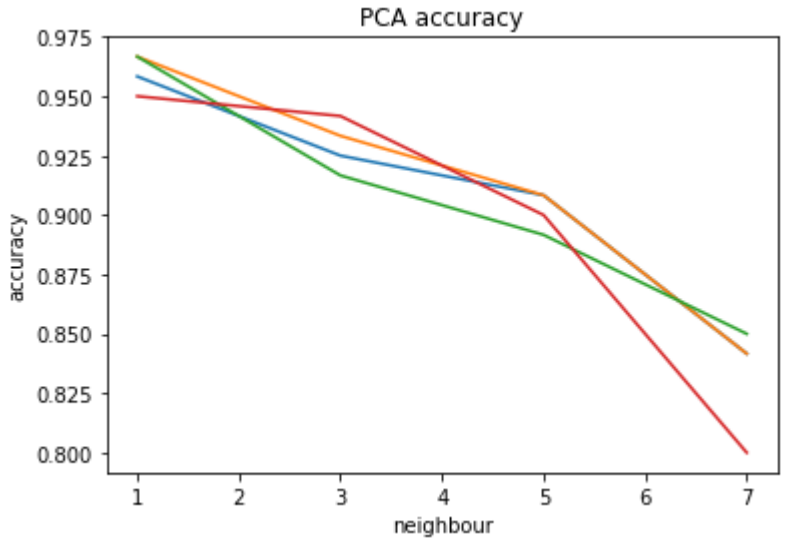
Alpha = 0.95:

for 1 neigbour -> accuracy = 0.95

for 3 neigbour -> accuracy = 0.942

for 5 neigbour -> accuracy = 0.9

for 7 neigbour -> accuracy = 0.8



LDA

In [ ]:
```
bonusLDA_projection_matrix, mean_vectors = LDA(bonus_training_set, bonus_training_labe

bonusLDA_projected_train_data = np.dot(bonus_training_set, bonusLDA_projection_matrix.
bonusLDA_projected_test_data = np.dot(bonus_testing_set, bonusLDA_projection_matrix.T)
```

In [ ]:
```
accuracy = []
for i in range(1,12,2):
  knn = KNeighborsClassifier(n_neighbors= i)
  knn.fit(bonusLDA_projected_train_data, bonus_training_labels)
  # print(knn.predict(projected_test_data))
  acc = knn.score(bonusLDA_projected_test_data, bonus_testing_labels)
  print(round(acc,3))
  accuracy.append(acc)
```

0.942

0.925

0.867

0.85

0.792

0.717

In [ ]:
```
for i in range(len(accuracy)):
  print(f"for {i*2+1} neigbour -> accuracy = {accuracy[i]}")
plt.plot(np.array([1,3,5,7,9,11]),accuracy)
plt.title('LDA accuracy')
plt.xlabel('neighbour')
plt.ylabel('accuracy')
```

for 1 neigbour -> accuracy = 0.9416666666666667

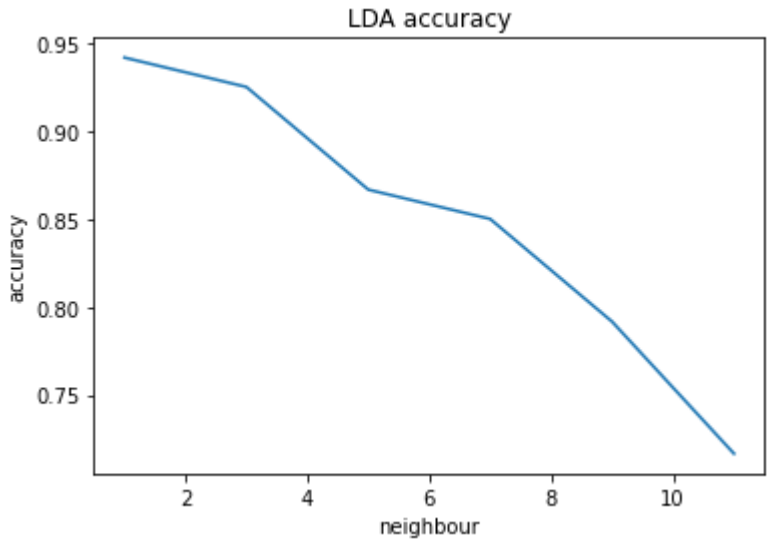
for 3 neigbour -> accuracy = 0.925

for 5 neigbour -> accuracy = 0.8666666666666667

for 7 neigbour -> accuracy = 0.85

for 9 neigbour -> accuracy = 0.7916666666666666

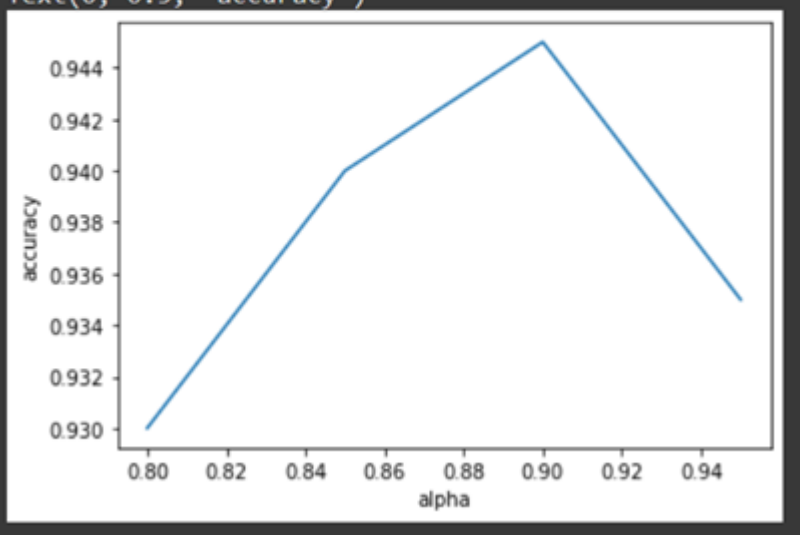
for 11 neigbour -> accuracy = 0.7166666666666667



Comparison

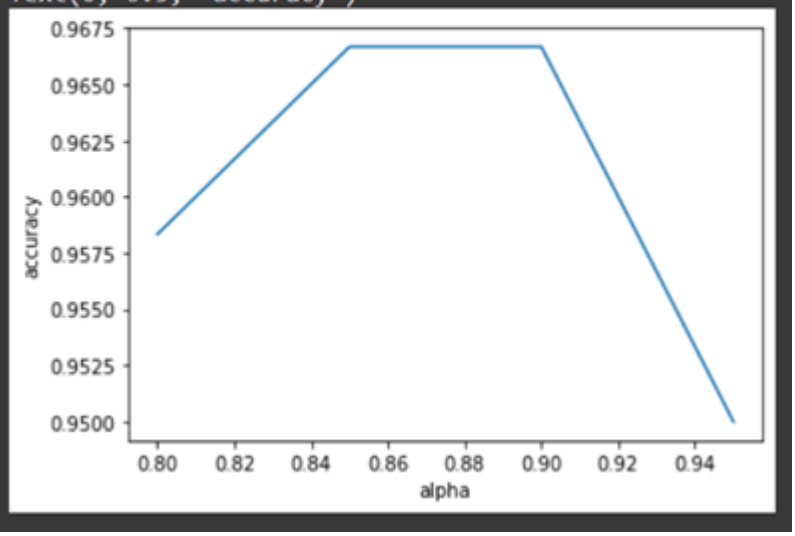
PCA For training = 50%, testing = 50%

```
alpha= 0.8 -->  accuracy0 = 0.93
alpha= 0.85 -->  accuracy1 = 0.94
alpha= 0.9 -->  accuracy2 = 0.945
alpha= 0.95 -->  accuracy3 = 0.935
Text(0, 0.5, 'accuracy')
```



For training = 70%, testing = 30%

```
alpha= 0.8 -->  accuracy0 = 0.958
alpha= 0.85 -->  accuracy1 = 0.967
alpha= 0.9 -->  accuracy2 = 0.967
alpha= 0.95 -->  accuracy3 = 0.95
Text(0, 0.5, 'accuracy')
```



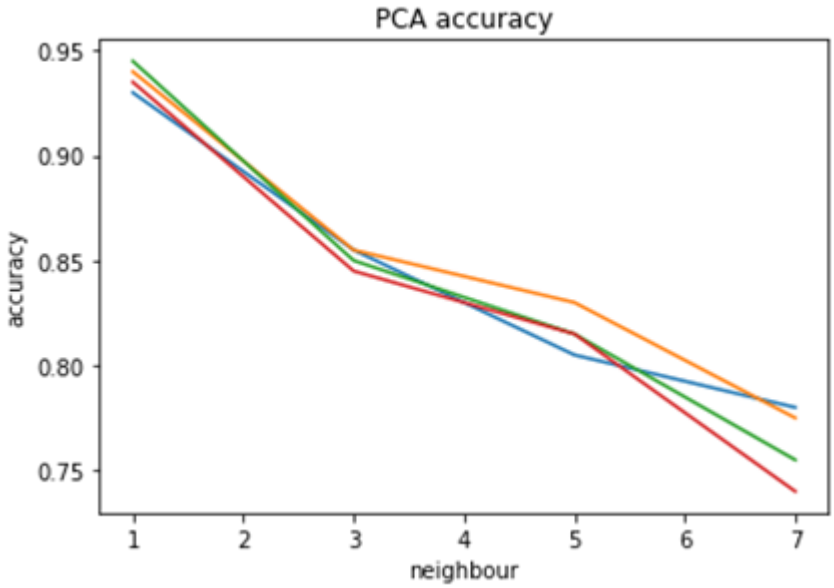
Using different Knn values (50 - 50):

```
Alpha = 0.8:
for 1 neigbour -> accuracy = 0.93
for 3 neigbour -> accuracy = 0.855
for 5 neigbour -> accuracy = 0.805
for 7 neigbour -> accuracy = 0.78
Alpha = 0.85:
for 1 neigbour -> accuracy = 0.94
for 3 neigbour -> accuracy = 0.855
for 5 neigbour -> accuracy = 0.83
for 7 neigbour -> accuracy = 0.775
Alpha = 0.9:
for 1 neigbour -> accuracy = 0.945
for 3 neigbour -> accuracy = 0.85
for 5 neigbour -> accuracy = 0.815
for 7 neigbour -> accuracy = 0.755
Alpha = 0.95:
for 1 neigbour -> accuracy = 0.935
for 3 neigbour -> accuracy = 0.845
for 5 neigbour -> accuracy = 0.815
for 7 neigbour -> accuracy = 0.74
```

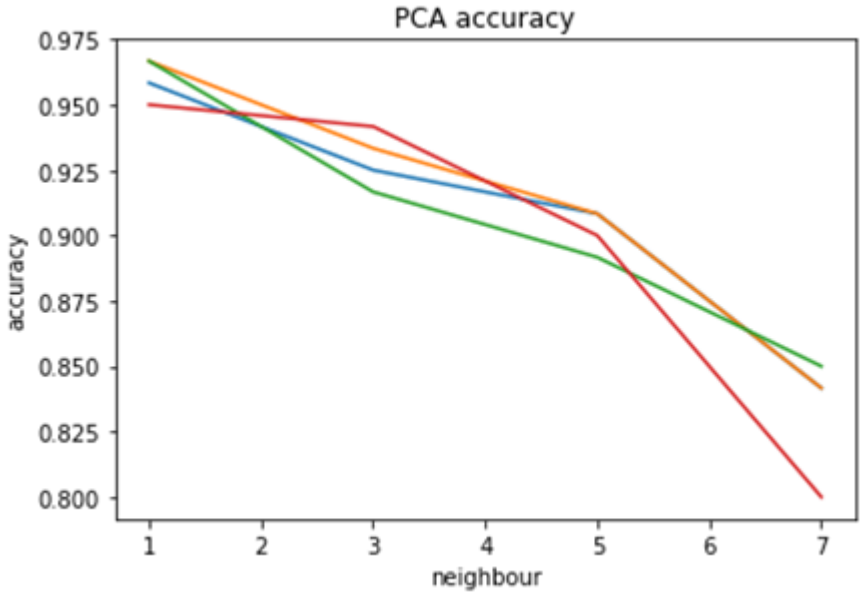
Using different Knn values (70 - 30):

```
Alpha = 0.8:
for 1 neigbour -> accuracy = 0.958
for 3 neigbour -> accuracy = 0.925
for 5 neigbour -> accuracy = 0.908
for 7 neigbour -> accuracy = 0.842
Alpha = 0.85:
for 1 neigbour -> accuracy = 0.967
for 3 neigbour -> accuracy = 0.933
for 5 neigbour -> accuracy = 0.908
for 7 neigbour -> accuracy = 0.842
Alpha = 0.9:
for 1 neigbour -> accuracy = 0.967
for 3 neigbour -> accuracy = 0.917
for 5 neigbour -> accuracy = 0.892
for 7 neigbour -> accuracy = 0.85
Alpha = 0.95:
for 1 neigbour -> accuracy = 0.95
for 3 neigbour -> accuracy = 0.942
for 5 neigbour -> accuracy = 0.9
for 7 neigbour -> accuracy = 0.8
```

USing different Knn values (50 - 50):



Using different Knn values (70 - 30):

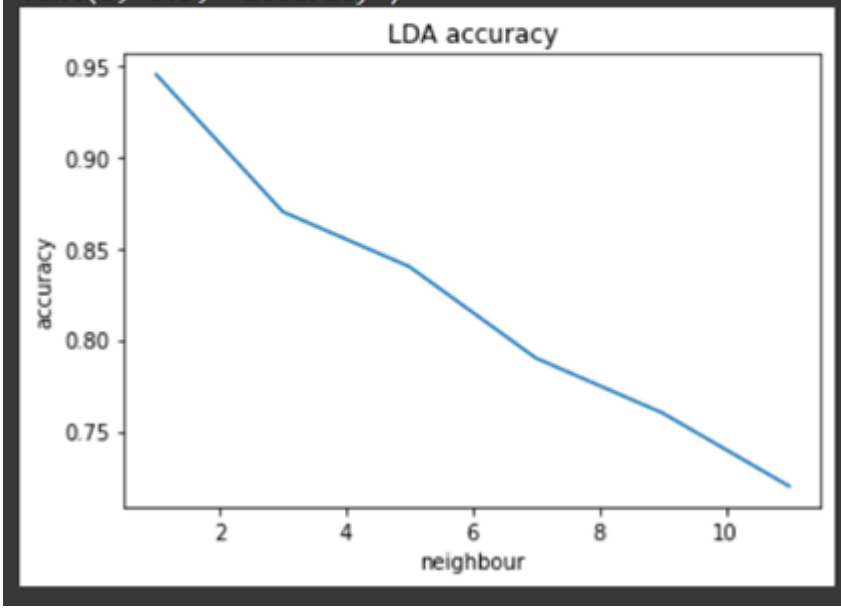


LDA

LDA accuracy with different neighbours

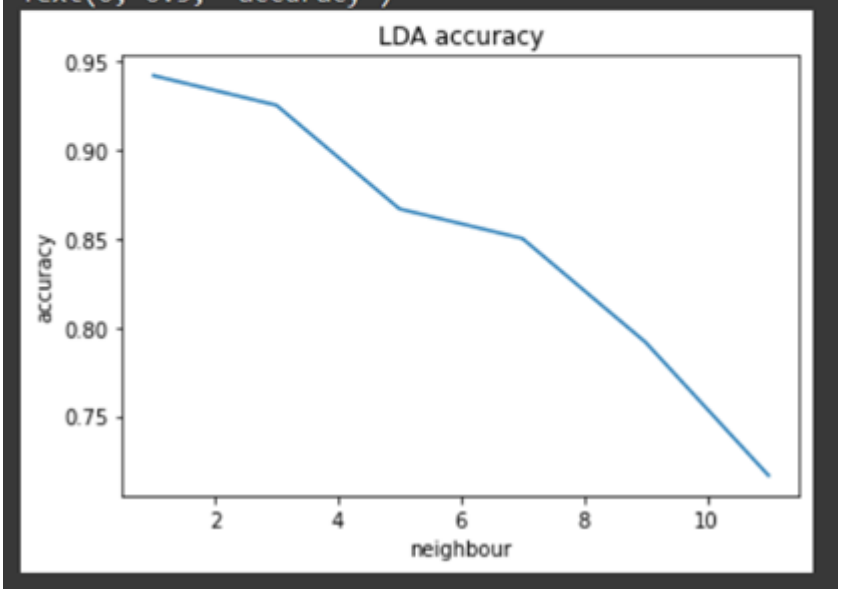
50 - 50 split:

```
for 0 neigbour -> accuracy = 0.945
for 1 neigbour -> accuracy = 0.87
for 2 neigbour -> accuracy = 0.84
for 3 neigbour -> accuracy = 0.79
for 4 neigbour -> accuracy = 0.76
for 5 neigbour -> accuracy = 0.72
Text(0, 0.5, 'accuracy')
```



70 - 30 split:

```
for 1 neigbour -> accuracy = 0.9416666666666667
for 3 neigbour -> accuracy = 0.925
for 5 neigbour -> accuracy = 0.8666666666666667
for 7 neigbour -> accuracy = 0.85
for 9 neigbour -> accuracy = 0.7916666666666666
for 11 neigbour -> accuracy = 0.7166666666666667
Text(0, 0.5, 'accuracy')
```



b) There are other variations of PCA and LDA beyond the original algorithms. Please use one of the variations of PCA and one variations of LDA other than the original ones. Compare the time

complexity and accuracy between the 2 different PCA and LDA models.

Randomized PCA takes 2 parameters: # components, method (randomized)
As for the standard PCA, it keeps calculating # components until reaching alpha, so we cannot preset a components number.
Instead, we will preset the number of components in the built-in method to the equivalent alpha in the implemented one.

alpha --> # components:

0.8 --> 37

0.85 --> 53

0.9 --> 77

0.95 --> 116

We will try this four times, with different # samples, going from 100 to 400, then compare their runtime and accuracy

In [ ]:
```
import time
from sklearn.decomposition import PCA


alpha = [0.8, 0.85, 0.9, 0.95]
eq_compo = [37, 53, 77, 116]

implemented_times = []
builtin_times = []
implemented_proj_data_mat = []
builtin_proj_data_mat = []
implemented_proj_test_mat = []
builtin_proj_test_mat = []
```

In [ ]:
```
def PCA_variation(i):
  current_data = training_set[0:i*50+50]
  current_label = training_labels[0:i*50+50]
  current_test_set = testing_set[0:i*50+50]
  current_test_label = testing_labels[0:i*50+50]

  impl_times = []
  buil_times = []

  # implemented PCA:
  common_start = time.time()
  current_values, current_vectors = PCA_impl_repetetive(current_data)
  common_end = time.time()
  common_total_time = common_end - common_start

  current_projection_matrices = []
  proj_data_list = []
  proj_test_list = []
```

```
  for j in range(len(alpha)):
    current_start = time.time()
    current_projection_matrices.append(PCA_impl(alpha[j], current_values, current_vect
    proj_data_list.append(reduce_dimension(current_projection_matrices[j], current_dat
    proj_test_list.append(reduce_dimension(current_projection_matrices[j], current_tes
    current_end = time.time()
    impl_times.append(round(common_total_time + current_end - current_start, 3))

  implemented_proj_data_mat.append(proj_data_list)
  implemented_proj_test_mat.append(proj_test_list)

  # build-in PCA (Randomized)
  proj_data_list = []
  proj_test_list = []
  for j in range(len(alpha)):
    if(eq_compo[j] <= len(current_data)):
      current_start = time.time()
      random_pca = PCA(n_components=eq_compo[j], svd_solver='randomized')
      proj_data_list.append(random_pca.fit_transform(current_data))
      proj_test_list.append(random_pca.transform(current_test_set))
      current_end = time.time()
      buil_times.append(round(current_end - current_start, 3))
  builtin_proj_data_mat.append(proj_data_list)
  builtin_proj_test_mat.append(proj_test_list)

  print(f"Implemented PCA times for {i*50+50} items for different alphas:")
  print(impl_times)
  implemented_times.append(impl_times)

  print(f"Built-in PCA times for {i*50+50} items for different components numbers:")
  print(buil_times)
  builtin_times.append(buil_times)
```

In [ ]:
```
for i in range(4):
  PCA_variation(i)
```

If # calculated times is less than 4, then the built-in pca was calculated less than 4 times, this is because: randomized pca refuses to set # components > min(rows, dimensions) of data matrix

Implemented PCA times for 50 items for different alphas: [202.288, 202.3, 202.315, 202.341]
Built-in PCA times for 50 items for different components numbers: [0.077]

Implemented PCA times for 100 items for different alphas: [207.647, 207.692, 207.762, 207.875]
Built-in PCA times for 100 items for different components numbers: [0.132, 0.17, 0.252]

Implemented PCA times for 150 items for different alphas: [246.615, 246.796, 246.937, 248.042]
Built-in PCA times for 150 items for different components numbers: [0.321, 0.244, 0.456, 0.548]

Implemented PCA times for 200 items for different alphas: [276.097, 276.518, 276.245, 276.896]
Built-in PCA times for 200 items for different components numbers: [0.49, 0.535, 0.868, 0.998]

for randomized pca to work, condition: # components <= min(rows, dimensions) of data matrix

First alpha is equivalent to 37 eigen vectors, while the second is equivalent to 53, and we are testing for samples numbers of 50, 100, 150, 200

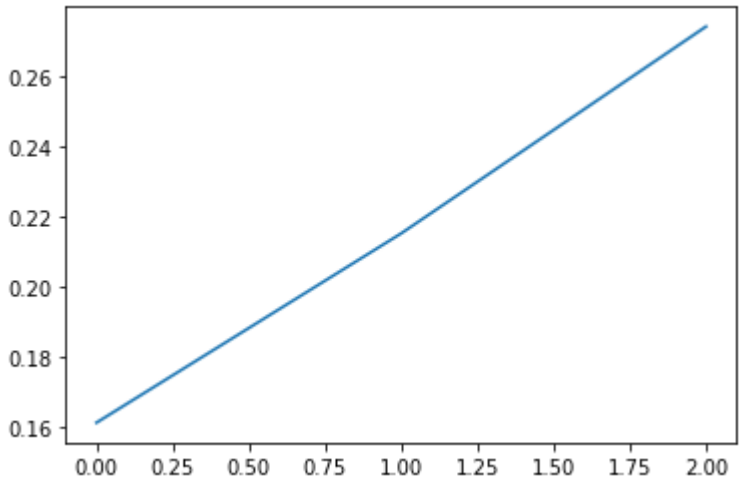
So in the first run, only the first alpha satisfies the condition, that's why we have only 1 data point, so their is no relation to draw!

We begin drawing from the second run, which contains 3 data points, next 2 runs each has 4 data points.

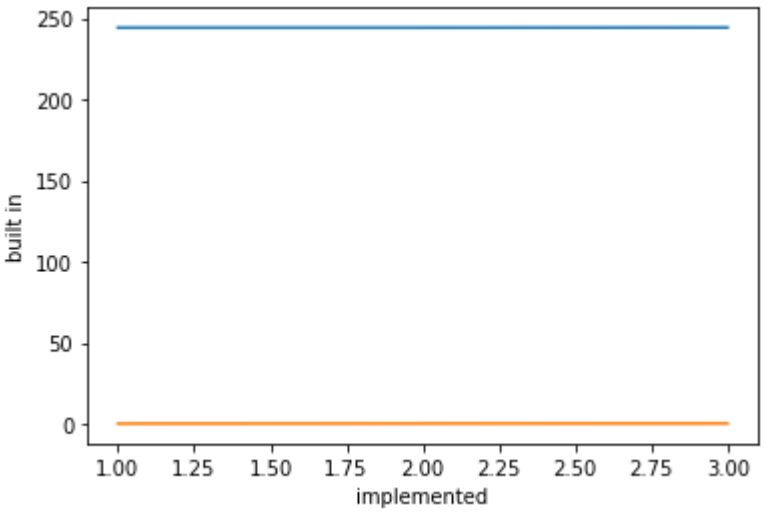
2nd run:

In [ ]:
```
plt.plot(implemented_times[1][:3])
```

In [ ]:
```
plt.plot(builtin_times[1][:3])
```

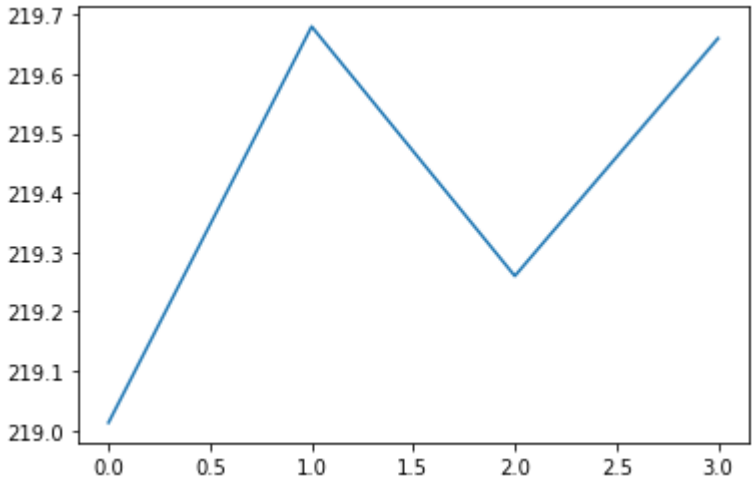


In [ ]:
```
plt.plot(implemented_times[1][:3])
plt.plot(builtin_times[1][:3])
```

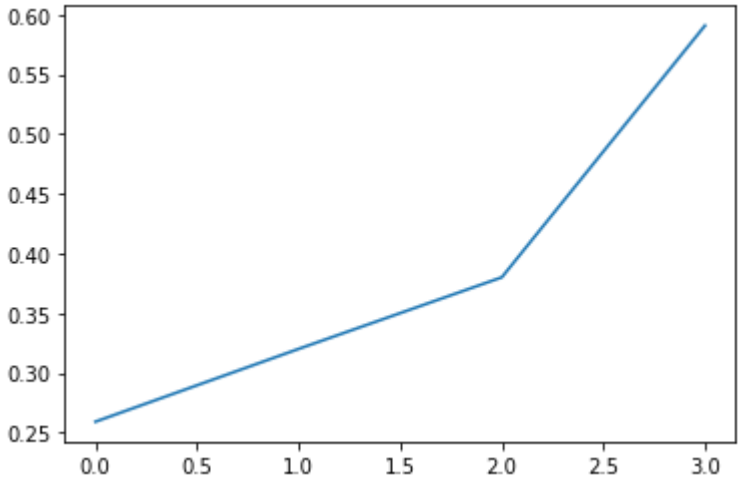


3rd run:

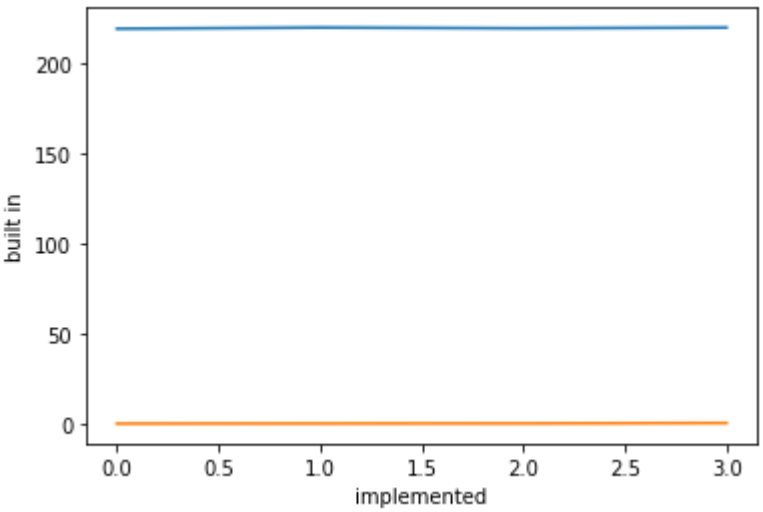
In [ ]:
```
plt.plot(implemented_times[2])
```



In [ ]:
```
plt.plot(builtin_times[2])
```

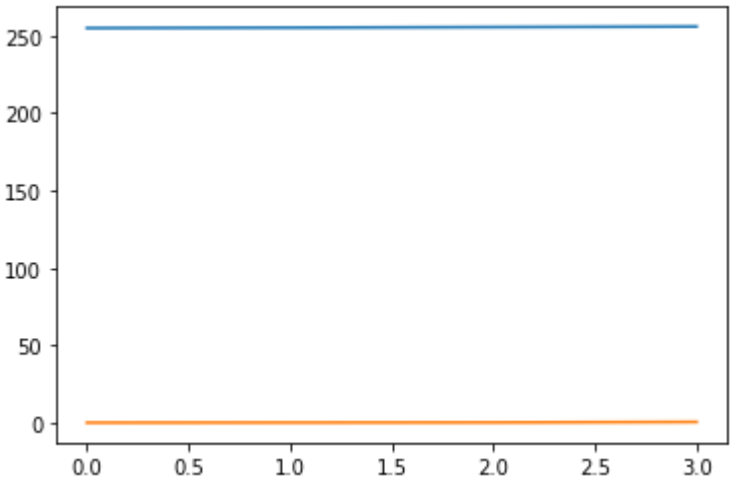


In [ ]:
```
plt.plot(implemented_times[2])
plt.plot(builtin_times[2])
```



3rd run:

In [ ]:
```
plt.plot(implemented_times[3])
plt.plot(builtin_times[3])
```



In [ ]:
```
print("Implemented PCA projected dimension - Build PCA projected dimension")
print()
print(len(implemented_proj_data_mat))
for i in range(len(builtin_proj_data_mat)):
  for j in range(len(builtin_proj_data_mat[i])):
    print(f"{str(j)}: {implemented_proj_data_mat[i][j].shape} - {builtin_proj_data_mat
```

Just to make sure they have the same # samples, and to see the number of components in each method:

Implemented PCA projected dimension - Build PCA projected dimension

0: (50, 14) - (50, 37)

1: (50, 14) - (100, 37)

2: (50, 19) - (100, 53)

3: (50, 25) - (100, 77)

4: (50, 34) - (150, 37)

5: (100, 25) - (150, 53)

6: (100, 34) - (150, 77)

7: (100, 46) - (150, 116)

8: (100, 65) - (200, 37)

9: (150, 31) - (200, 53)

10: (150, 43) - (200, 77)

11: (150, 62) - (200, 116)

Measuring Accuracies

In [ ]:
```
# Global Arrays (2d) to be exported
impl_pca_accuracies = []
builtin_pca_accuracies = []
for i in range(len(builtin_proj_data_mat)):
  # Local Arrays that help with forming global Arrays
  arr_impl = []
  arr_builtin = []
  for j in range(len(builtin_proj_data_mat[i])):
    arr_impl.append(knn1PCA(implemented_proj_data_mat[i][j], training_labels[0:i*50+50
    arr_builtin.append(knn1PCA(builtin_proj_data_mat[i][j], training_labels[0:i*50+50]
  impl_pca_accuracies.append(arr_impl)
  builtin_pca_accuracies.append(arr_builtin)

# plotAlphaTotal is used to know each dimension's alphas values
plotAlphaTotal = []
for i in range(len(impl_pca_accuracies)):
  plotAlpha = []
  for j in range(len(impl_pca_accuracies[i])):
    plotAlpha.append(alpha[j])
    print(f"for #samples = {i*50+50} & alpha = {alpha[j]} -->  accuracy of implemented
    print(f"for #samples = {i*50+50} & alpha = {alpha[j]} -->  accuracy of random buil
    print()
  print()
  plotAlphaTotal.append(plotAlpha)
```

PS: We notice that the accuracies are nearly the same for all cases, except for 200 samples & alpha 0.9, there is a very tiny error (0.945 - 0.94)

Result of the above code

for #samples = 50 & alpha = 0.8 --> accuracy of implemented pca0 = 0.96
for #samples = 50 & alpha = 0.8 --> accuracy of random built-in pca0 = 0.96

for #samples = 100 & alpha = 0.8 --> accuracy of implemented pca1 = 0.96
for #samples = 100 & alpha = 0.8 --> accuracy of random built-in pca1 = 0.96

for #samples = 100 & alpha = 0.85 --> accuracy of implemented pca1 = 0.96
for #samples = 100 & alpha = 0.85 --> accuracy of random built-in pca1 = 0.96

for #samples = 100 & alpha = 0.9 --> accuracy of implemented pca1 = 0.96
for #samples = 100 & alpha = 0.9 --> accuracy of random built-in pca1 = 0.96

for #samples = 150 & alpha = 0.8 --> accuracy of implemented pca2 = 0.973
for #samples = 150 & alpha = 0.8 --> accuracy of random built-in pca2 = 0.973

for #samples = 150 & alpha = 0.85 --> accuracy of implemented pca2 = 0.973
for #samples = 150 & alpha = 0.85 --> accuracy of random built-in pca2 = 0.973

for #samples = 150 & alpha = 0.9 --> accuracy of implemented pca2 = 0.973
for #samples = 150 & alpha = 0.9 --> accuracy of random built-in pca2 = 0.973

for #samples = 150 & alpha = 0.95 --> accuracy of implemented pca2 = 0.973
for #samples = 150 & alpha = 0.95 --> accuracy of random built-in pca2 = 0.973

for #samples = 200 & alpha = 0.8 --> accuracy of implemented pca3 = 0.93
for #samples = 200 & alpha = 0.8 --> accuracy of random built-in pca3 = 0.93

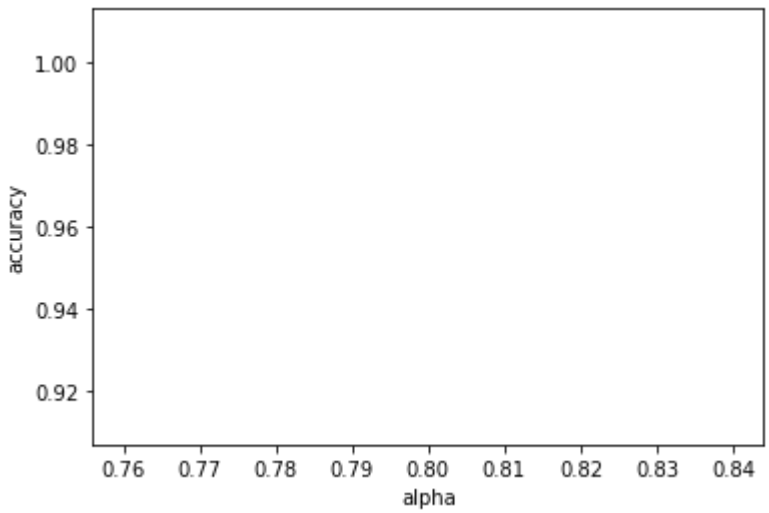
for #samples = 200 & alpha = 0.85 --> accuracy of implemented pca3 = 0.94
for #samples = 200 & alpha = 0.85 --> accuracy of random built-in pca3 = 0.94

for #samples = 200 & alpha = 0.9 --> accuracy of implemented pca3 = 0.945
for #samples = 200 & alpha = 0.9 --> accuracy of random built-in pca3 = 0.94

for #samples = 200 & alpha = 0.95 --> accuracy of implemented pca3 = 0.935
for #samples = 200 & alpha = 0.95 --> accuracy of random built-in pca3 = 0.94

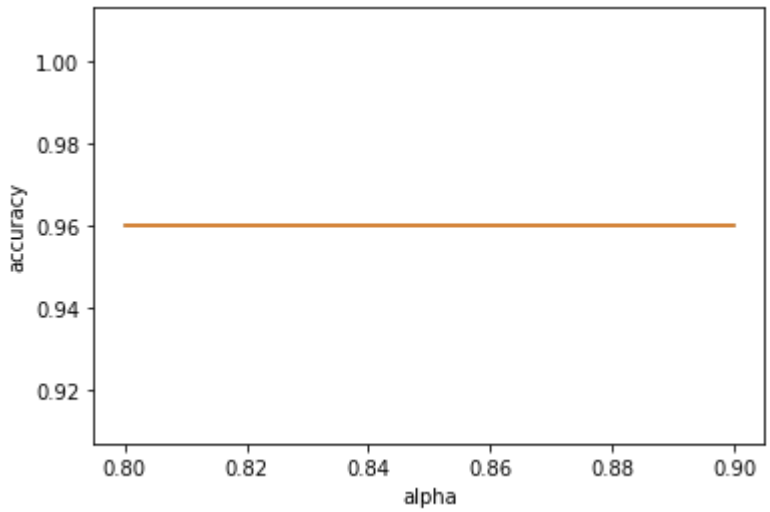
Nothing shown for only 1 data point!

In [ ]:
```
plt.plot(np.array(plotAlphaTotal[0]), np.array(impl_pca_accuracies[0]))
  plt.plot(np.array(plotAlphaTotal[0]), np.array(builtin_pca_accuracies[0]))
  plt.xlabel('alpha')
  plt.ylabel('accuracy')
```



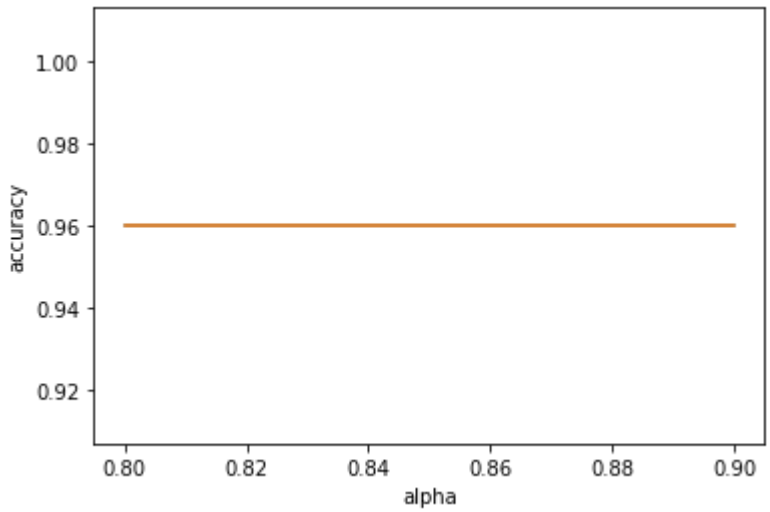
2nd run: 3 data points

In [ ]:
```
plt.plot(np.array(plotAlphaTotal[1]), np.array(impl_pca_accuracies[1]))
  plt.plot(np.array(plotAlphaTotal[1]), np.array(builtin_pca_accuracies[1]))
  plt.xlabel('alpha')
  plt.ylabel('accuracy')
```



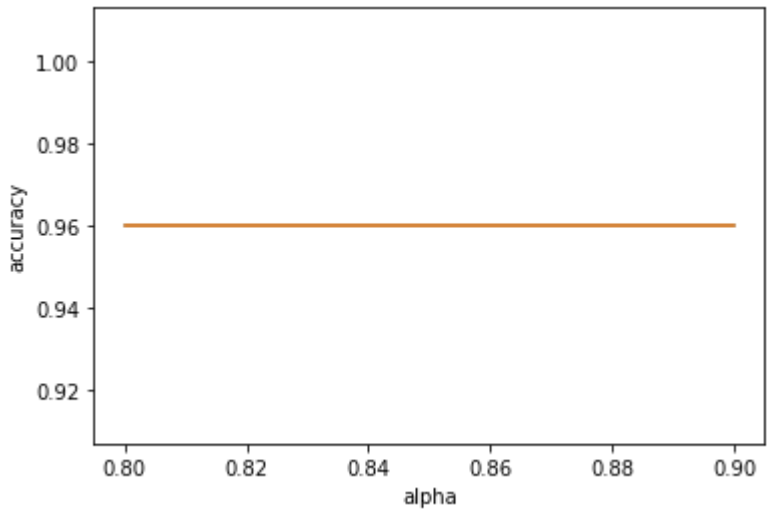
3rd run: 4 data points

In [ ]:
```
plt.plot(np.array(plotAlphaTotal[1]), np.array(impl_pca_accuracies[1]))
  plt.plot(np.array(plotAlphaTotal[1]), np.array(builtin_pca_accuracies[1]))
  plt.xlabel('alpha')
  plt.ylabel('accuracy')
```



4th run: 4 data points

In [ ]:
```
plt.plot(np.array(plotAlphaTotal[1]), np.array(impl_pca_accuracies[1]))
  plt.plot(np.array(plotAlphaTotal[1]), np.array(builtin_pca_accuracies[1]))
  plt.xlabel('alpha')
  plt.ylabel('accuracy')
```



bonus 2 - LDA part

Not implemented!

In [ ]:
```
bonus2_proj_matrices = []
bonus2_proj_train = []
bonus2_proj_test = []

componentsN = [15, 30, 45]
```

In [ ]:
```
for i in range(3):
  bonusLDA_projection_matrix, mean_vectors = LDA(bonus_training_set, bonus_training_la
  bonusLDA_projected_train_data = np.dot(bonus_training_set, bonusLDA_projection_matri
  bonusLDA_projected_test_data = np.dot(bonus_testing_set, bonusLDA_projection_matrix.

  bonus2_proj_matrices.append(bonusLDA_projection_matrix)
  bonus2_proj_train.append(bonusLDA_projected_train_data)
  bonus2_proj_test.append(bonusLDA_projected_test_data)
```

In [ ]:
```
accuracy = []
for i in range(1,8,2):
  knn = KNeighborsClassifier(n_neighbors= i)
  knn.fit(projected_train_data, training_labels)
  # print(knn.predict(projected_test_data))
  acc = knn.score(projected_test_data, testing_labels)
  print(acc)
  accuracy.append(acc)
```

Comments: