TP2_MachineLearning_BIM_1part_FEI

December 3, 2020

Emotion Recognition based on facial landmarks

This part of the practical session is about **emotion recognition** based on facial landmarks. We will use the FEI dataset (https://fei.edu.br/~cet/facedatabase.html) to recognize the emotion of a person by analyzing 68 facial landmarks (already estimated and placed). Below, you will find a picture with an example. We will focus on two emotions neutral and happy.

Please answer all questions and complete the code where you see **XXXXXXXXXXXXX**

Deadline: Upload this notebook and the one about Toy Examples as a single .zip file to the Moodle. You have one week.

[1]:
```
if 'google.colab' in str(get_ipython()):
  from google_drive_downloader import GoogleDriveDownloader as gdd
  gdd.
 ↪download_file_from_google_drive(file_id='15vsAdMepHzdoZ3iqNS3kpI3KGW7DOvRs',
  dest_path='./Data_FEI.npz')
  gdd.
 ↪download_file_from_google_drive(file_id='1ywQbf23-JoPklWCcH_mi5Nuw5BQskxvB',
  dest_path='./facial_landmarks_68markup.jpg')
else:
  print('You are not using Colab. Please define working_dir with the absolute␣
 ↪path to the folder where you downloaded the data')

# Please modify working_dir only if you are using your Anaconda (and not Google␣
 ↪Colab)
# You should write the absolute path of your working directory with the data
Working_directory="./"
```

Otherwise, you can also load them from your local machine using the following code

[2]:
```
"""
from google.colab import files

uploaded = files.upload()

for fn in uploaded.keys():
  print('User uploaded file "{name}" with length {length} bytes'.format(
      name=fn, length=len(uploaded[fn])))
"""
```

Let's load the Python packages containing the functions needed for the practical session.

[3]:
```
import numpy as np
from time import time

import itertools
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import paired_distances
from sklearn.model_selection import  cross_val_score, cross_validate,␣
↪GridSearchCV, KFold, StratifiedKFold
from sklearn.metrics import classification_report
from sklearn.utils.multiclass import unique_labels
from sklearn.metrics import confusion_matrix
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn import decomposition
from sklearn.preprocessing import StandardScaler, MinMaxScaler
import matplotlib.pyplot as plt
# this is needed to plot figures within the notebook
%matplotlib inline
np.random.seed(seed=666)

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.simplefilter(action='ignore', category=FutureWarning)
from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings(action='ignore', category=ConvergenceWarning)
```

We also load a user-defined function useful for plotting the confusion matrix

[4]:
```
# Code from scikit-learn

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')
```

```
    print(cm)

    fig, ax = plt.subplots()
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)

    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")


    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax
```

Now, let's load the data

e have a list of images, the position of the original landmarks (aligned to the images), the position of the landmarks after a normalization process called Generalized Procrustes Analysis (please refer to https://en.wikipedia.org/wiki/Generalized_Procrustes_analysis), the outputs with the class labels and the names of the images.

Generalized Procrustes Analysis (GPA) is used to keep only shape differences between the configurations of landmarks. That is to say, we align all configurations to an average one using only rigid transformations (uniform scaling, rotation and translation). This means that if I take a facial picture of subject A, then step back, translate and rotate a bit the camera and retake a facial picture of the same subject (who has not moved) the two picture will be different with therefore different landmark position. However, after a GPA, the two landmark positions should be perfectly aligned removing the "nuisance" differences related to rotation, translation and uniform scaling.

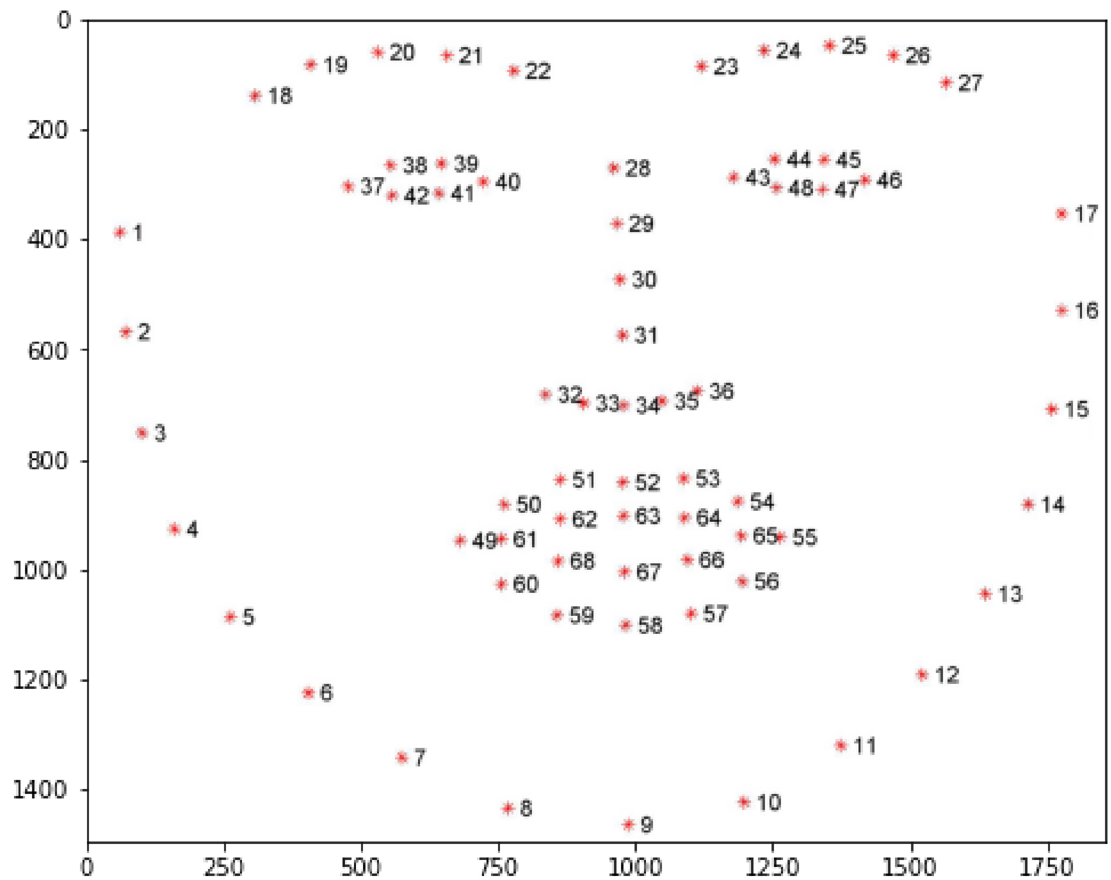
[5]:
```
# Parameters
dim=2 # dimension
# Loading data
```

```python
with np.load(Working_directory + 'Data_FEI.npz') as data:
    Images=data['Images_FEI'] # list of images
    X = data['Landmarks_FEI'] # original landmarks
    XGPA = data['Landmarks_FEI_GPA'] # landmarks after GPA (Generalized␣
 ↪Prcrustes Analysis, https://en.wikipedia.org/wiki/
 ↪Generalized_Procrustes_analysis)
    Y = data['Emotions_FEI'] # class, 0 for neutral and 1 for happy
    Names = data['Names_FEI']
N,M = X.shape # number subjects
M = int(M/2) # Number of landmarks (they are in 2D)
print('Number of subjects:', N, '; Number of landmarks:',M)
class_names = ["neutral","happy"]
```

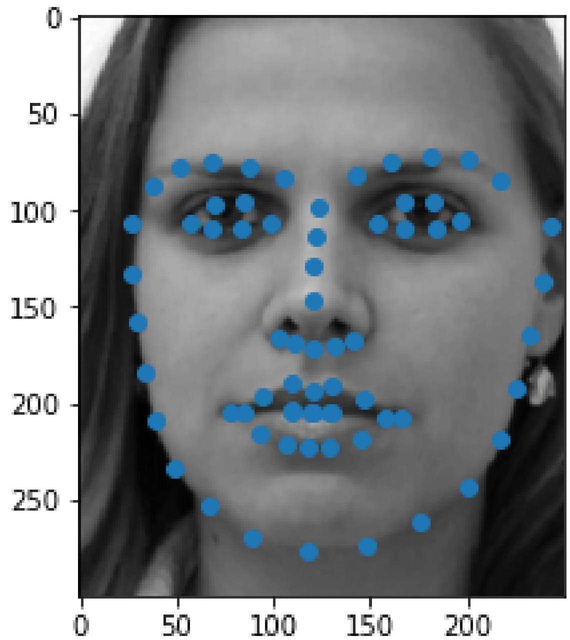
```
Number of subjects: 400 ; Number of landmarks: 68
```

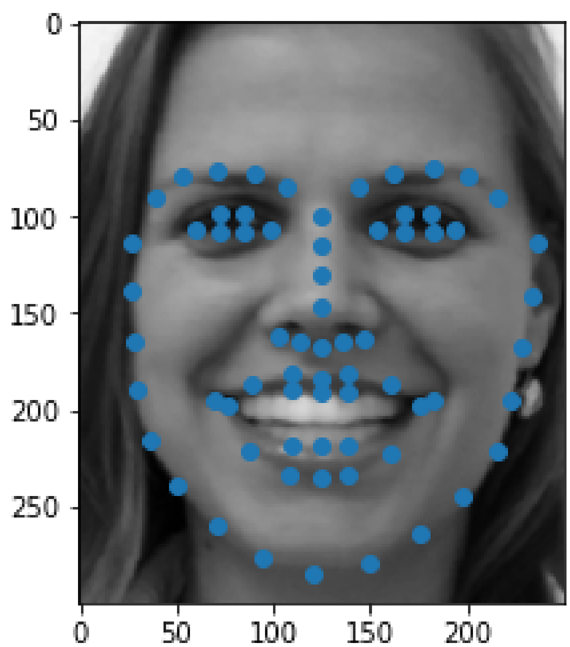
Here, we show an example of facial landmarks

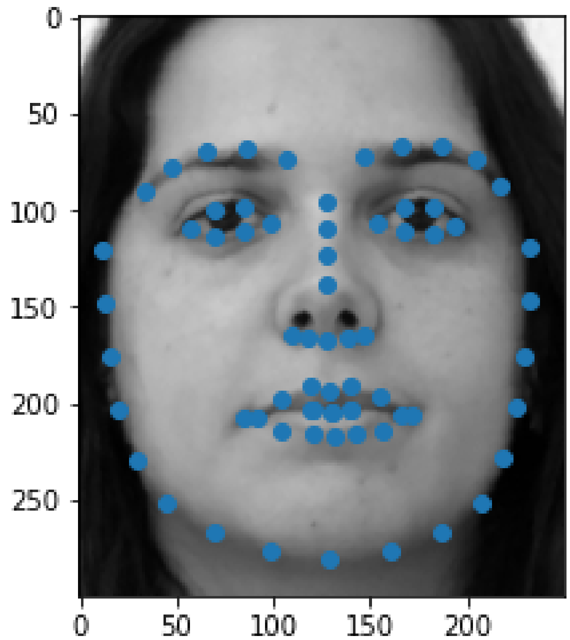
[6]:
```python
# Plot the facial landmarks
Example=plt.imread(Working_directory + 'facial_landmarks_68markup.jpg') #␣
 ↪function to read a jpg image
plt.figure(figsize = (8,8)) # Size of the plot
plt.imshow(Example)
plt.show()
```

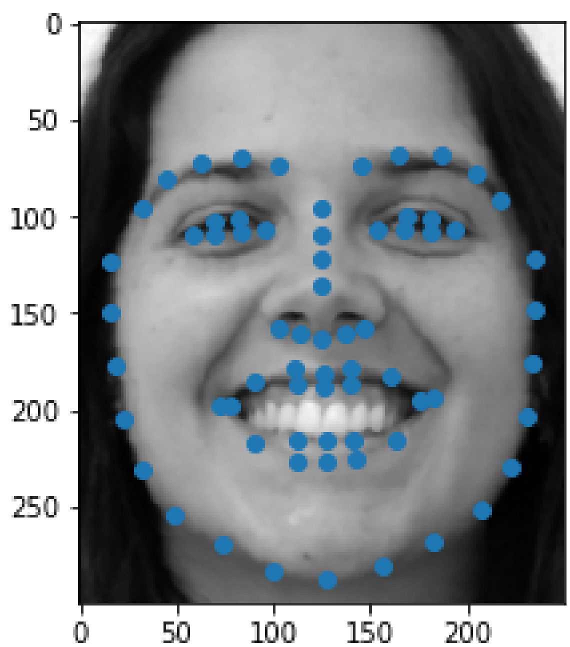


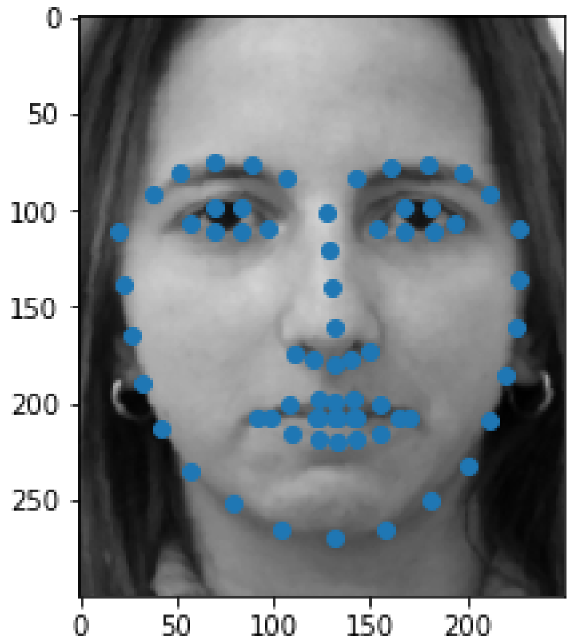
[7]:
```
# plot the first 6 images of the data-set
for i in range(0,6):
    image = Images[i,:,:]
    plt.figure()
    plt.imshow(image, cmap='gray', origin='upper')
    landmark=X[i,:]
    x=landmark[::2]
    y=landmark[1::2]
    plt.plot(x,y,'o')
    plt.show()
```

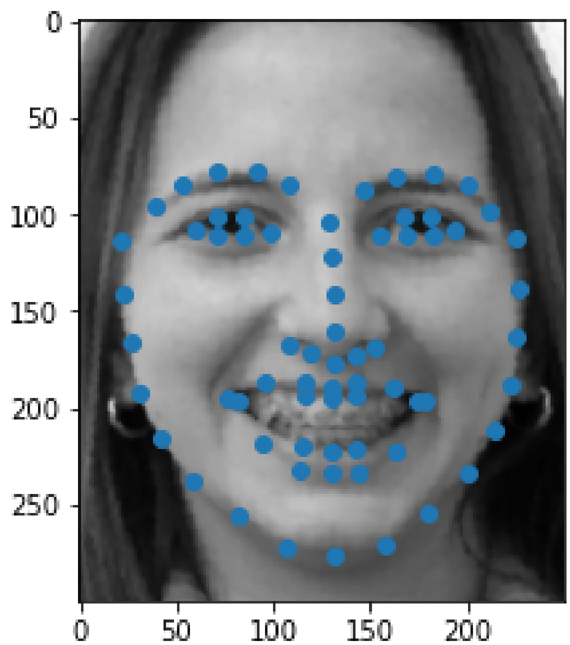

0
50
100
150
200
250
0
50
100
150
200


0
50
100
150
200
250
0
50
100
150
200


0
50
100
150
200
250
0
50
100
150
200


0
50
100
150
200
250
0
50
100
150
200





Question (IMP+IMH): after plotting the first 6 images of the data-set, what do you notice ? Do you notice a regular pattern ? Do you think that it would be worth it to randomly shuffle the data ?

Answer:

Yes we do need to shuffle since we see its alternate between with smile and without a smile and we might get a biased algorithm.

---

[8]:
```
# Shuffle data randomly
indeces=np.arange(N) # Integers from 0 to N-1
#print(indeces)

#Hint: Use np.random.shuffle
np.random.shuffle(indeces)
#print(indeces)

XpGPA=XGPA[indeces]
Xp=X[indeces]
Yp=Y[indeces]
Imagesp=Images[indeces]
Xmean = np.mean(XpGPA,axis=0) # Compute average

Namesp=[''] * N
for i in range(0,N):
    Namesp[i]=Names[indeces[i]]
```

Among the loaded data, we also have aligned landmarks after a Generalized Procrustes Analysis. Let's check them and compare them with the landmarks before alignement.

QUESTION (IMP+IMH): Please comment the results. What can you notice ?

Answer:

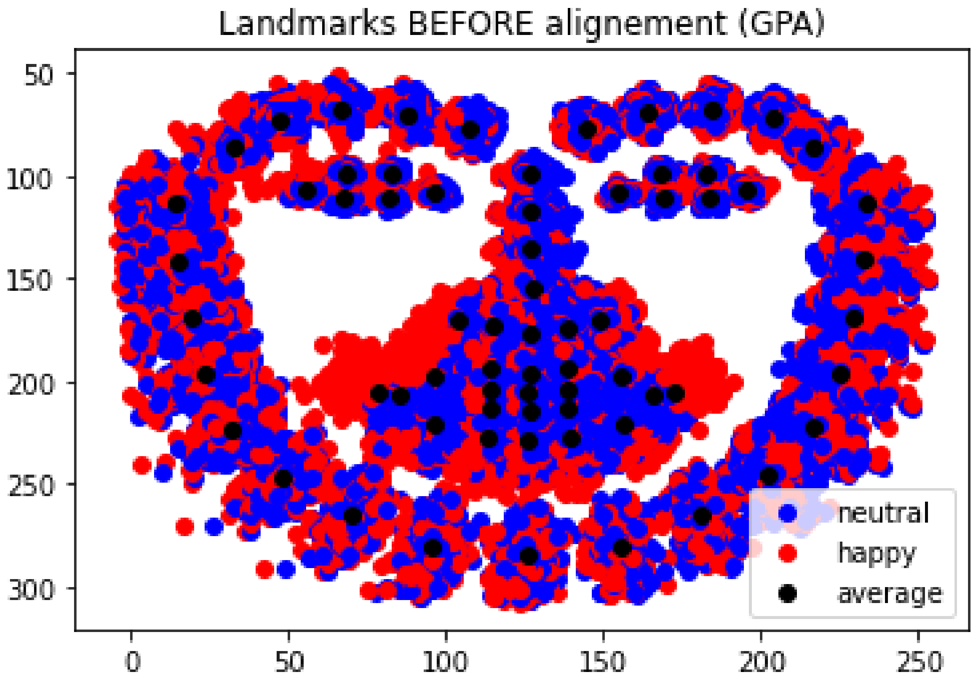
We can see that after the GPA we get much clearer "face" with smile.

---

[9]:
```
# Plot all landmarks BEFORE GPA
plt.figure()
for i in range(0,N):
    landmark=Xp[i]
    x=landmark[::2]
    y=landmark[1::2]
    if Yp[i].astype(int)==0:
        neutral=plt.scatter(x, y, c='b')
    else:
        happy=plt.scatter(x, y, c='r')
Xaverage = np.mean(Xp,axis=0) # Compute average
average=plt.scatter(Xaverage[::2],Xaverage[1::2],color='k')
plt.legend((neutral,happy,average),('neutral','happy','average'))
```

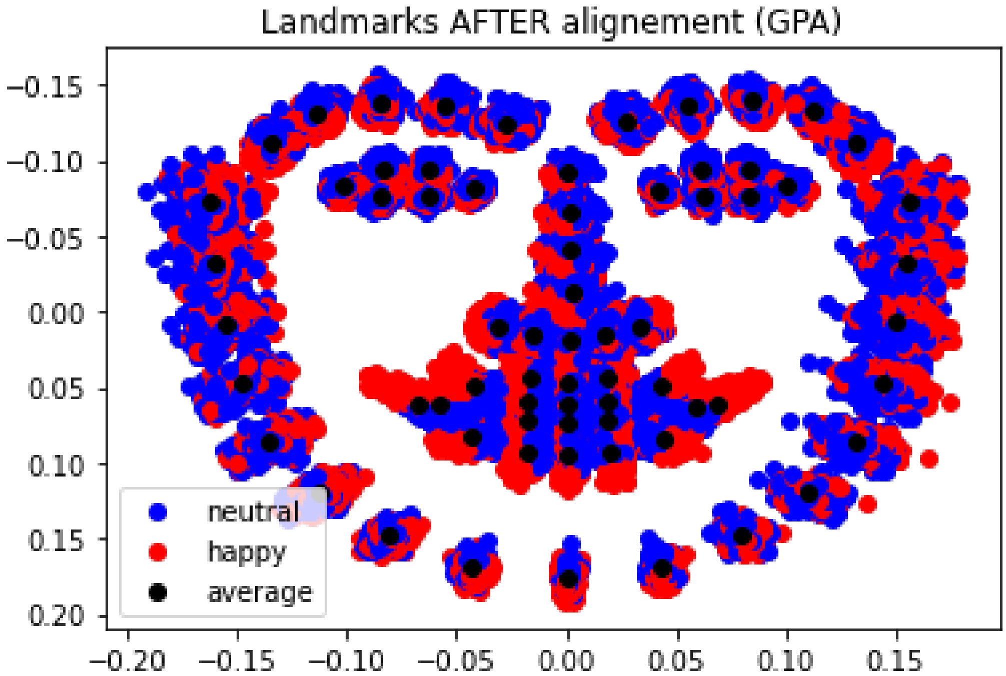
```
plt.gca().invert_yaxis()
plt.title('Landmarks BEFORE alignement (GPA)')
```

[9]: Text(0.5, 1.0, 'Landmarks BEFORE alignement (GPA)')




```
[10]: # Plot all landmarks AFTER GPA
plt.figure()
for i in range(0,N):
    landmark=XpGPA[i]
    x=landmark[::2]
    y=landmark[1::2]
    if Yp[i].astype(int)==0:
        neutral=plt.scatter(x, y, c='b')
    else:
        happy=plt.scatter(x, y, c='r')
average=plt.scatter(Xmean[::2],Xmean[1::2],color='k')
plt.legend((neutral,happy,average),('neutral','happy','average'))
plt.gca().invert_yaxis()
plt.title('Landmarks AFTER alignement (GPA)')
```

[10]: Text(0.5, 1.0, 'Landmarks AFTER alignement (GPA)')



We need now to compute some features for the classification algorithms. As first idea, we could use the paired Euclidean distances between the landmarks of every subject and the landmarks of the average configuration.

---

[11]:
```
# Compute distances from the average configuration (features)

dist_average=np.zeros((N,M))
average=np.reshape(Xmean,(M,2)) # Reshape average as matrix
#print(Xmean.shape)
#print(average.shape)

for i in range(N):
    landmark=np.reshape(XpGPA[i],(M,2)) # Reshape all landmarks as matrices
    dist_average[i]=paired_distances(landmark, average)

print('Number of subjects N is: ', dist_average.shape[0], ' ; number of␣
 ↪features is: ',  dist_average.shape[1] )
```

```
Number of subjects N is:  400  ; number of features is:  68
```

Question (IMP+IMH): One usual question in Machine Learning is, do we need to scale/normalize the features ? What do you think ? Should we do it in this case ? Compute both scaled and normalized data.

Answer:

I wouldent scale or normalize the features since all of them are in the same scale already and we wont normalize it since we wont want to "loss" the meaning of the diffrence between the values.

---

[12]:
```
# Scale data (each feature will have average equal to 0 and unit variance)
# https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.
↪StandardScaler.html
scaler = StandardScaler()
scaler.fit(dist_average)
dist_average_scale=scaler.transform(dist_average)
print('Scaler')
print('Number of subjects N is: ', dist_average_scale.shape[0], ' ; number of␣
↪features is: ',  dist_average_scale.shape[1] )


# Normalize data (each feature will be scaled into the range 0,1)
# https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.
↪MinMaxScaler.html

normalizer = MinMaxScaler()
normalizer.fit(dist_average)
dist_average_normalize=normalizer.transform(dist_average)
print('Normalizer')
print('Number of subjects N is: ', dist_average_normalize.shape[0], ' ; number␣
↪of features is: ',  dist_average_normalize.shape[1] )
```

```
Scaler
Number of subjects N is:  400  ; number of features is:  68
Normalizer
Number of subjects N is:  400  ; number of features is:  68
```

Let's divide the data-set into Training and Test sets using original, scaled and normalized data.

[13]:
```
# Create training and test set
X_train, X_test, y_train, y_test = train_test_split(dist_average, np.ravel(Yp),␣
↪test_size=0.33, random_state=42)

# Create training and test set
X_train_scale, X_test_scale, y_train_scale, y_test_scale =␣
↪train_test_split(dist_average_scale, np.ravel(Yp), test_size=0.33,␣
↪random_state=42)

# Create training and test set
```

```
X_train_normalize, X_test_normalize, y_train_normalize, y_test_normalize =␣
↪train_test_split(dist_average_normalize, np.ravel(Yp), test_size=0.33,␣
↪random_state=42)
```

Let's try to fit LDA to all training sets and predict the error on their respective test sets.

Question (IMP+IMH): Compare the performnces between original, scaled and normalized data. Comment the results.

Answer:

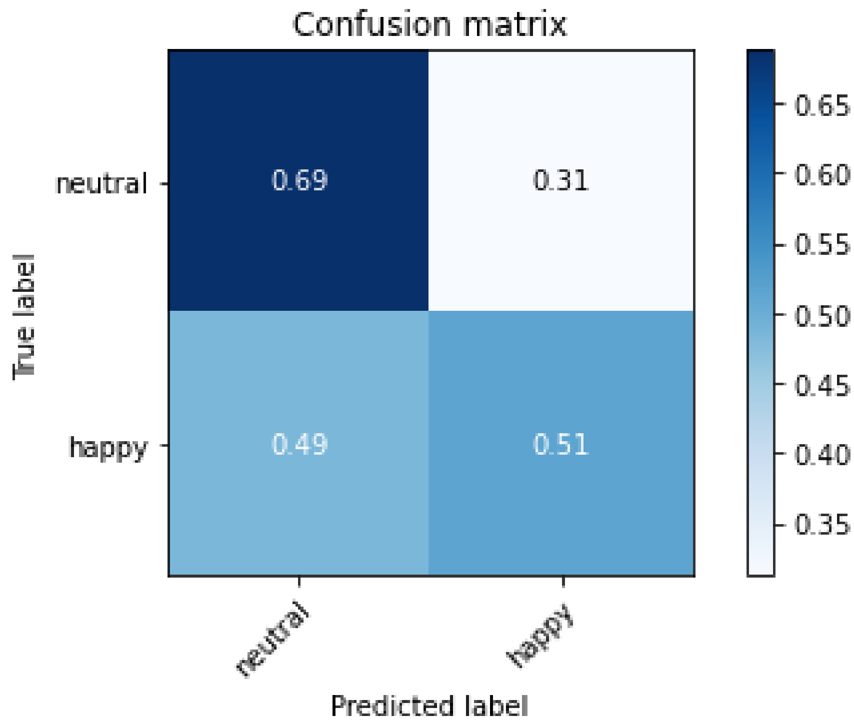
We can see that indeed there is no sense to normalize and scale the data because the result stays the same with the normalize and scale or without it.

---

[14]:
```
# Fitting LDA to original data
print("Fitting LDA to training set")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)
y_pred = lda.predict(X_test)
print("done in %0.3fs" % (time() - t0))
print(classification_report(y_test, y_pred))

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True)
plt.show()
```

```
Fitting LDA to training set
done in 0.006s
              precision    recall  f1-score   support

           0       0.57      0.69      0.62        64
           1       0.64      0.51      0.57        68

    accuracy                           0.60       132
   macro avg       0.60      0.60      0.60       132
weighted avg       0.60      0.60      0.60       132

Normalized confusion matrix
[[0.6875     0.3125    ]
 [0.48529412 0.51470588]]
```

```
<Figure size 432x288 with 0 Axes>
```



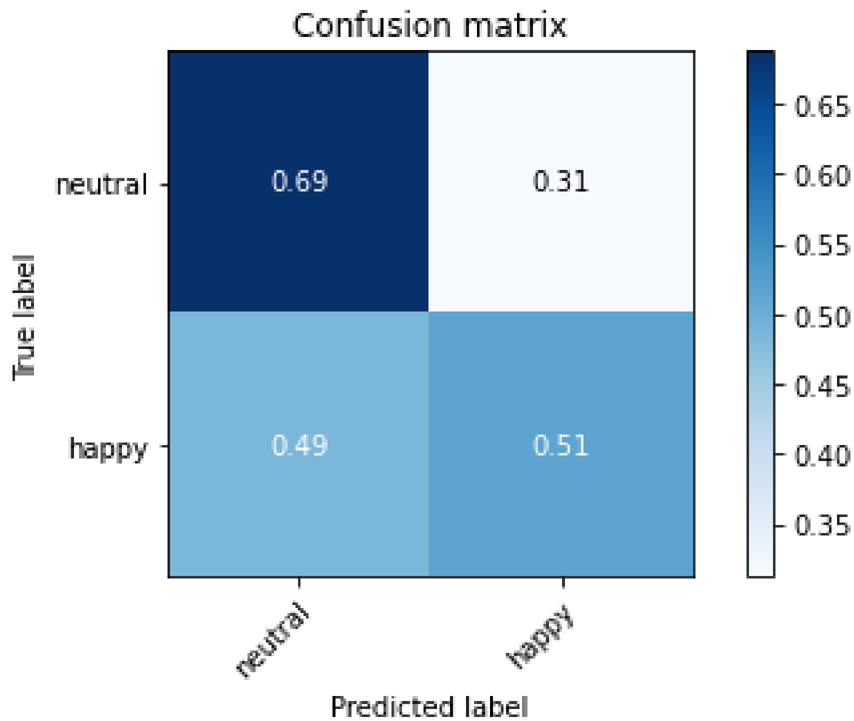
[15]:

```
# Fitting LDA to scaled data
print("Fitting LDA to scaled dataset")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda.fit(X_train_scale, y_train_scale)
y_pred_scale = lda.predict(X_test_scale)
print("done in %0.3fs" % (time() - t0))
print(classification_report(y_test_scale, y_pred_scale))

# Compute confusion matrix
cnf_matrix_scale = confusion_matrix(y_test_scale, y_pred_scale)

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix_scale, classes=class_names, normalize=True)
plt.show()
```

```
Fitting LDA to scaled dataset
done in 0.007s
              precision    recall  f1-score   support

           0       0.57      0.69      0.62        64
           1       0.64      0.51      0.57        68
```

```
    accuracy                           0.60       132
   macro avg       0.60      0.60      0.60       132
weighted avg       0.60      0.60      0.60       132

Normalized confusion matrix
[[0.6875     0.3125    ]
 [0.48529412 0.51470588]]
```

```
<Figure size 432x288 with 0 Axes>
```



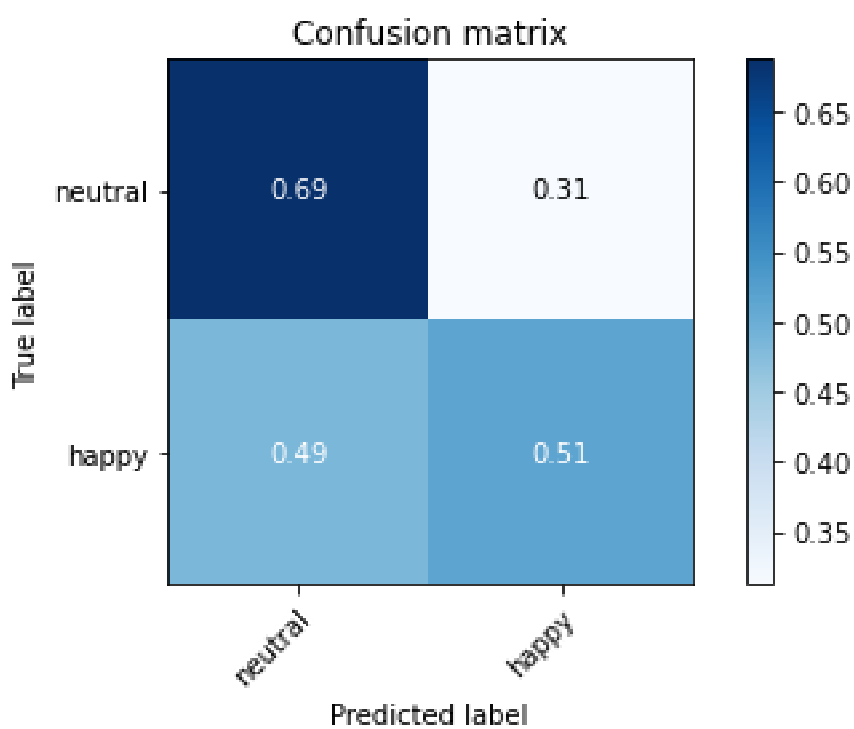
[16]:

```python
# Fitting LDA to normalized data
print("Fitting LDA to normalized dataset")
t0 = time()
lda_normalize = LinearDiscriminantAnalysis()
lda_normalize.fit(X_train_normalize, y_train_normalize)
y_pred_normalize = lda_normalize.predict(X_test_normalize)
print("done in %0.3fs" % (time() - t0))
print(classification_report(y_test_normalize, y_pred_normalize))

# Compute confusion matrix
cnf_matrix_normalize = confusion_matrix(y_test_normalize, y_pred_normalize)

# Plot normalized confusion matrix
plt.figure()
```

```
plot_confusion_matrix(cnf_matrix_normalize, classes=class_names, normalize=True)
plt.show()
```

```
Fitting LDA to normalized dataset
done in 0.006s
              precision    recall  f1-score   support

           0       0.57      0.69      0.62        64
           1       0.64      0.51      0.57        68

    accuracy                           0.60       132
   macro avg       0.60      0.60      0.60       132
weighted avg       0.60      0.60      0.60       132

Normalized confusion matrix
[[0.6875     0.3125    ]
 [0.48529412 0.51470588]]
```

```
<Figure size 432x288 with 0 Axes>
```



We can then use the function 'cross_val_score' to compute the CV score. Let's use all methods seen today.

Question (IMP+IMH):

1. Analyze the code and explain what it does.
2. compare the performances between original, scaled and normalized data

Answers:

1. the code takes different subsets of training and test (cv=5 so that means 5 different subsets) and gives the accuracy of each subset using the model that was defined for him. we get 5 different models for each method and therefor 5 accuracies.
2. we can see again and reasure that we do not need to sacling and normlize this data

---

[17]:
```
# Cross-validation for Model Assessment
# raw data
print('raw data')
# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda_score = cross_val_score(lda,X=dist_average, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(lda_score.mean(),␣
 ↪lda_score.std() ))

# Fitting QDA
print("Fitting QDA")
t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=dist_average, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
 ↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=dist_average, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
 ↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=dist_average, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
```

```
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))

# Fitting K-nearest neighbour
print("Fitting K-nearest neighbour")
t0 = time()
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=dist_average, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(neigh_score.mean(),␣
↪neigh_score.std() ))
```

```
raw data
Fitting LDA
done in 0.033s
 Average and std CV score : 0.5574999999999999 +- 0.045138675213169485
Fitting QDA
done in 0.017s
 Average and std CV score : 0.5625 +- 0.044721359549995794
Fitting Logistic Regression
done in 0.023s
 Average and std CV score : 0.5349999999999999 +- 0.0483476990145343
Fitting Naive-Bayes
done in 0.011s
 Average and std CV score : 0.5725 +- 0.058843011479699094
Fitting K-nearest neighbour
done in 0.043s
 Average and std CV score : 0.6049999999999999 +- 0.032210246816812824
```

[18]:
```
# Cross-validation for Model Assessment
# normalized data
print('normalized data')
# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda_score = cross_val_score(lda,X=dist_average_normalize, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(lda_score.mean(),␣
↪lda_score.std() ))

# Fitting QDA
print("Fitting QDA")
t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=dist_average_normalize, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
```

```
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=dist_average_normalize, y=np.
↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=dist_average_normalize, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))

# Fitting K-nearest neighbour
print("Fitting K-nearest neighbour")
t0 = time()
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=dist_average_normalize, y=np.
↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(neigh_score.mean(),␣
↪neigh_score.std() ))
```

```
normalized data
Fitting LDA
done in 0.037s
 Average and std CV score : 0.5574999999999999 +- 0.045138675213169485
Fitting QDA
done in 0.024s
 Average and std CV score : 0.5625 +- 0.044721359549995794
Fitting Logistic Regression
done in 0.059s
 Average and std CV score : 0.5675000000000001 +- 0.03674234614174766
Fitting Naive-Bayes
done in 0.007s
 Average and std CV score : 0.5725 +- 0.058843011479699094
Fitting K-nearest neighbour
done in 0.053s
```

```
Average and std CV score : 0.5775 +- 0.03482097069296032
```

[19]:
```
# Cross-validation for Model Assessment
# scaled data
print('scaled data')
# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda_score = cross_val_score(lda,X=dist_average_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(lda_score.mean(),␣
↪lda_score.std() ))

# Fitting QDA
print("Fitting QDA")
t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=dist_average_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=dist_average_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=dist_average_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))

# Fitting K-nearest neighbour
print("Fitting K-nearest neighbour")
t0 = time()
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=dist_average_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
```

```
print(" Average and std CV score : {0} +- {1}".format(neigh_score.mean(),␣
↪neigh_score.std() ))
```

```
scaled data
Fitting LDA
done in 0.030s
 Average and std CV score : 0.5574999999999999 +- 0.045138675213169485
Fitting QDA
done in 0.027s
 Average and std CV score : 0.5625 +- 0.044721359549995794
Fitting Logistic Regression
done in 0.122s
 Average and std CV score : 0.55 +- 0.01767766952966367
Fitting Naive-Bayes
done in 0.009s
 Average and std CV score : 0.5725 +- 0.058843011479699094
Fitting K-nearest neighbour
done in 0.053s
 Average and std CV score : 0.5625 +- 0.017677669529663688
```

In the previous example we have fixed the hyper-parameter K to 3. We coudl use CV to find the best value.

Question(IMP+IMH): Comment the code and explain what it does.

Answer:

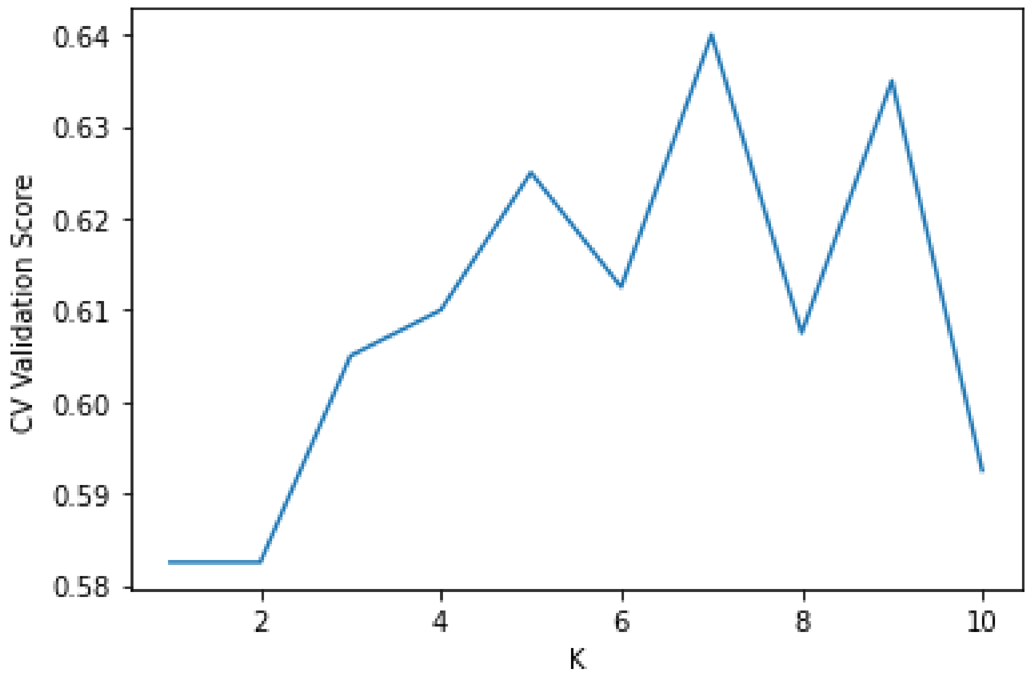
The code search for the best K for the K-nearest neighbour between thr values 1 to 10 these values used in GridSearchCV to find the optimal one for the whole dataset, then plot the score of the CV for every number of K

---

[20]:
```
# Looking for the best K in K-nearest neighbour
parameters = {'n_neighbors':[1,2,3,4,5,6,7,8,9,10]}
neighCV = KNeighborsClassifier()
grid = GridSearchCV(neighCV, parameters, cv=5, n_jobs=-1)
grid.fit(dist_average, np.ravel(Yp))

print('The best K is', grid.best_params_.get('n_neighbors'), ' with an average␣
↪validation score equal to ', grid.best_score_)

# plot the CV validation score for each K value
plt.plot([1,2,3,4,5,6,7,8,9,10], grid.cv_results_.get('mean_test_score'))
plt.xlabel('K')
plt.ylabel('CV Validation Score')
```

```
The best K is 7  with an average validation score equal to  0.6399999999999999
```

[20]: Text(0, 0.5, 'CV Validation Score')



We could also use CV to assess the prediction error (generalization error) in a left-out test set.

Question(IMP+IMH): Comment the code and explain what it does.

Answer:

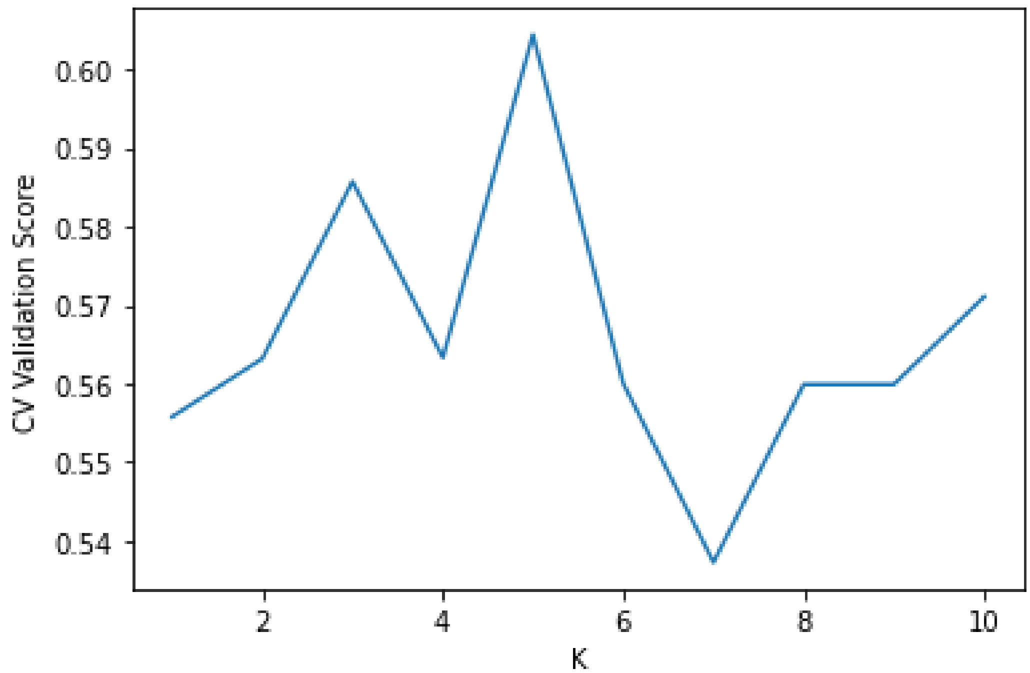
The code search for the best K for the K-nearest neighbor between the values 1 to 10 these values used in GridSearchCV to find the optimal one for the X_train and y_train dataset therefor we comput it for 5 (cv=5) subsets of train and validation sets, then plots the score of the CV for every number. then takes the best model from the CV and computes only its score for the test set.

---

[21]:
```
# We only use the training set for finding the best hyper-parameter
parameters = {'n_neighbors':[1,2,3,4,5,6,7,8,9,10]}
neighCV = KNeighborsClassifier()
grid = GridSearchCV(neighCV, parameters, cv=5, n_jobs=-1)
grid.fit(X_train, y_train)

print('The best K is', grid.best_params_.get('n_neighbors'), ' with an average␣
 ↪validation score equal to ', grid.best_score_)

# plot the CV validation score for each K value
```

```
plt.plot([1,2,3,4,5,6,7,8,9,10], grid.cv_results_.get('mean_test_score'))
plt.xlabel('K')
plt.ylabel('CV Validation Score')

# Let's now use the best model to assess the test score
BestModel=grid.best_estimator_
print('The test score is', BestModel.score(X_test, y_test))
```

```
The best K is 5  with an average validation score equal to  0.6044025157232704
The test score is 0.6590909090909091
```



Question(IMP+IMH): Comment the results of the two previous experiments. What about the best K and validation/test error ? Are the results the same ? Why in your opinion ?

Answer: The results are not the same because the second model has fewer data to train on, but it is the more correct way to train the model. to have a train and validation sets to find the hyperparameters and then try the best model on the test set.

It seems that these features do not work very well... let's try to change them. We can use the distances between all combinations of landmarks. Each subject has M*(M-1)/2 features.

---

[22]:
```python
# Use distances between all combinations of landmarks. Each subject has M*(M-1)/
↪2 features
dist_combination=np.zeros((N,int((M*(M-1)/2))))
for s in range(N):
    temp=[]
    landmarks=np.reshape(XpGPA[s],(M,2))
    for i in range(M-1):
        a=landmarks[i,:]
        for j in range(i+1,M):
            b=landmarks[j,:]
            dist_2=np.sqrt(np.dot(a, a) - 2 * np.dot(a, b) + np.dot(b, b))
            temp.append(dist_2)
    dist_combination[s]=np.array(temp)

# Scale data (each feature will have average equal to 0 and unit variance)
scaler.fit(dist_combination)
dist_combination_scale=scaler.transform(dist_combination)

print('Number of subjects N is: ', dist_combination_scale.shape[0], ' ; number␣
↪of features is: ',  dist_combination_scale.shape[1] )
```

```
Number of subjects N is:  400  ; number of features is:  2278
```

Question (IMP+IMH): Should we scale/normalize the new features ?

Answer:

Yes we should since we no longer in a known dimension of the data If not scale, the feature with a higher value range starts dominating in our model and we might have a biased model

Use the classification algorithms seen before to test the discriminative power of the new features.

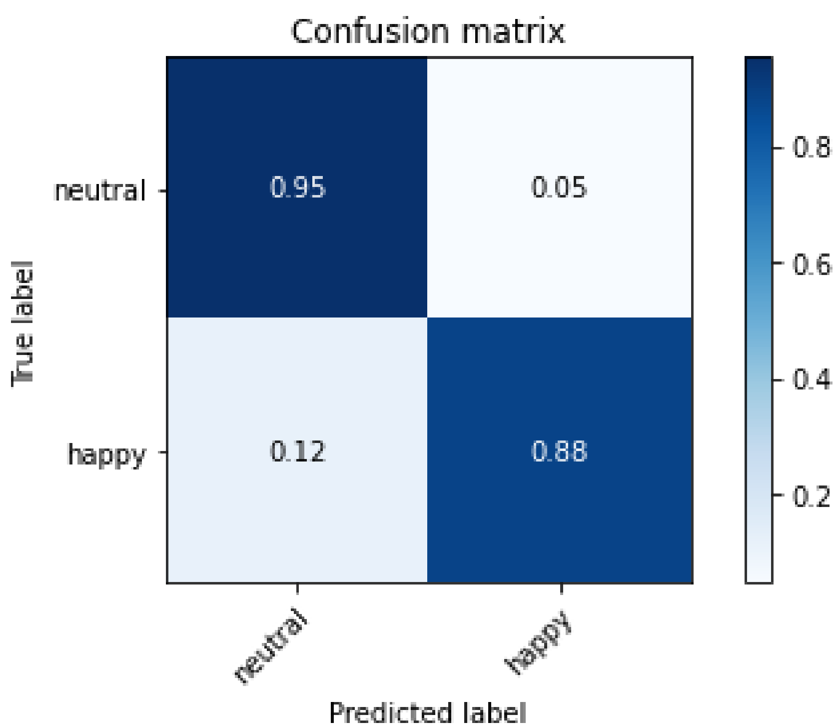
[23]:
```python
# Create training and test set
X_train_scale, X_test_scale, y_train_scale, y_test_scale =␣
↪train_test_split(dist_combination_scale, np.ravel(Yp), test_size=0.33,␣
↪random_state=42)

# Fitting LDA to scaled data
print("Fitting LDA to scaled dataset")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda.fit(X_train_scale, y_train_scale)
y_pred_scale = lda.predict(X_test_scale)
print("done in %0.3fs" % (time() - t0))
print(classification_report(y_test_scale, y_pred_scale))

# Compute confusion matrix
cnf_matrix_scale = confusion_matrix(y_test_scale, y_pred_scale)
```

```
# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix_scale, classes=class_names, normalize=True)
plt.show()

# Cross-validation for Model Assessment

# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda_score = cross_val_score(lda,X=dist_combination_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(lda_score.mean(),␣
↪lda_score.std() ))

# Fitting QDA
print("Fitting QDA")
t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=dist_combination_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=dist_combination_scale, y=np.
↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=dist_combination_scale, y=np.ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))

# Fitting K-nearest neighbour
print("Fitting K-nearest neighbour")
t0 = time()
```

```
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=dist_combination_scale, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(neigh_score.mean(),␣
 ↪neigh_score.std() ))
```

```
Fitting LDA to scaled dataset
done in 0.203s
              precision    recall  f1-score   support

           0       0.88      0.95      0.92        64
           1       0.95      0.88      0.92        68

    accuracy                           0.92       132
   macro avg       0.92      0.92      0.92       132
weighted avg       0.92      0.92      0.92       132

Normalized confusion matrix
[[0.953125   0.046875  ]
 [0.11764706 0.88235294]]
```

```
<Figure size 432x288 with 0 Axes>
```




```
Fitting LDA
```

```
done in 1.542s
 Average and std CV score : 0.9174999999999999 +- 0.023184046238739257
Fitting QDA
```

```
C:\Users\eidan\anaconda3\envs\env_full\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
C:\Users\eidan\anaconda3\envs\env_full\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
C:\Users\eidan\anaconda3\envs\env_full\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
C:\Users\eidan\anaconda3\envs\env_full\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
C:\Users\eidan\anaconda3\envs\env_full\lib\site-
packages\sklearn\discriminant_analysis.py:715: UserWarning: Variables are
collinear
  warnings.warn("Variables are collinear")
```

```
done in 0.502s
 Average and std CV score : 0.6924999999999999 +- 0.07441438033068606
Fitting Logistic Regression
done in 0.632s
 Average and std CV score : 0.9625 +- 0.0262202212042538
Fitting Naive-Bayes
done in 0.096s
 Average and std CV score : 0.9475 +- 0.031024184114977156
Fitting K-nearest neighbour
done in 0.952s
 Average and std CV score : 0.9225 +- 0.03482097069296032
```

---

mmmm it seems that some variables are collinear. Collinearity means that one variable can be linearly predicted by the other, basically it means that there is redundancy…

Question (IMP+IMH): Which technique could you use to reduce the collinearity/redundancy ? Use it and test the new features.

Answer:

DONE!

---

[24]:
```
# Hint: we saw this technique during the last lecture...
from sklearn.decomposition import PCA
pca = PCA(random_state=1) # by fixing the random_state we are sure that results␣
↪are always the same
dist_combination_scale_pca=pca.fit_transform(dist_combination_scale)
var_explained_pca=pca.explained_variance_ratio_

Threshold_PCA = 0.99
CumulativePca=np.cumsum(var_explained_pca)
indexPCA=np.argwhere(CumulativePca>Threshold_PCA)
PCAComp=indexPCA[0][0]
dist_combination_scale_pca = dist_combination_scale_pca[:,:PCAComp]

print('Number of subjects N is: ', dist_combination_scale_pca.shape[0], ' ;␣
↪number of features is: ',  dist_combination_scale_pca.shape[1])
```

```
Number of subjects N is:  400  ; number of features is:  42
```

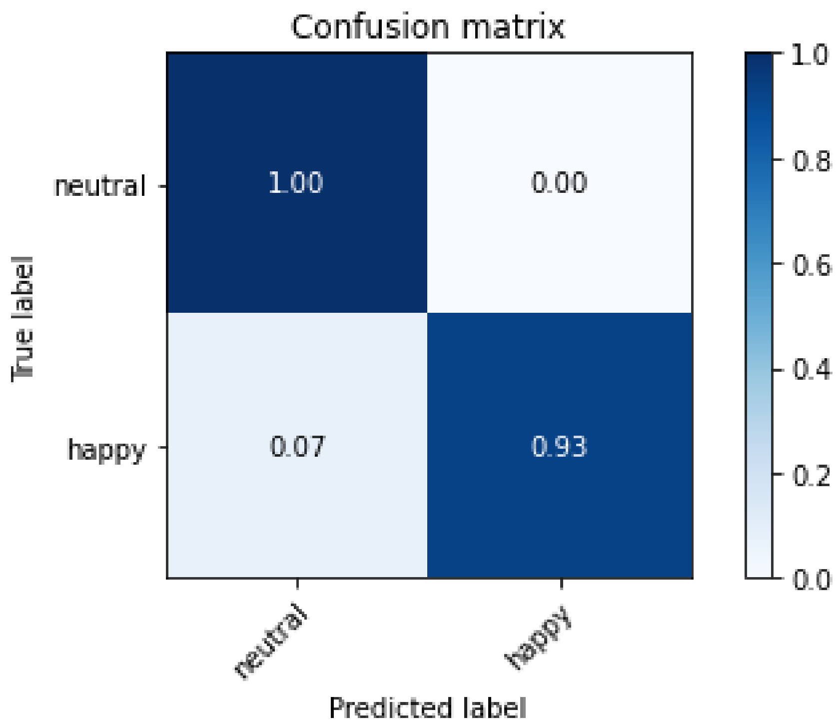
[25]:
```
# Test the predictive power of the new features using LDA, QDA, etc.
# Create training and test set
X_train_scale, X_test_scale, y_train_scale, y_test_scale =␣
↪train_test_split(dist_combination_scale_pca, np.ravel(Yp), test_size=0.33,␣
↪random_state=42)

# Fitting LDA to scaled data
print("Fitting LDA to scaled dataset")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda.fit(X_train_scale, y_train_scale)
y_pred_scale = lda.predict(X_test_scale)
print("done in %0.3fs" % (time() - t0))
print(classification_report(y_test_scale, y_pred_scale))

# Compute confusion matrix
cnf_matrix_scale = confusion_matrix(y_test_scale, y_pred_scale)

# Plot normalized confusion matrix
plt.figure()
plot_confusion_matrix(cnf_matrix_scale, classes=class_names, normalize=True)
plt.show()

# Cross-validation for Model Assessment

# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
```

```
lda_score = cross_val_score(lda,X=dist_combination_scale_pca, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(lda_score.mean(),␣
 ↪lda_score.std() ))

# Fitting QDA
print("Fitting QDA")
t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=dist_combination_scale_pca, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
 ↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=dist_combination_scale_pca, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
 ↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=dist_combination_scale_pca, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
 ↪GNB_score.std() ))

# Fitting K-nearest neighbour
print("Fitting K-nearest neighbour")
t0 = time()
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=dist_combination_scale_pca, y=np.
 ↪ravel(Yp),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(neigh_score.mean(),␣
 ↪neigh_score.std() ))
```

```
Fitting LDA to scaled dataset
```

```
done in 0.005s
              precision    recall  f1-score   support

           0       0.93      1.00      0.96        64
           1       1.00      0.93      0.96        68

    accuracy                           0.96       132
   macro avg       0.96      0.96      0.96       132
weighted avg       0.96      0.96      0.96       132

Normalized confusion matrix
[[1.         0.        ]
 [0.07352941 0.92647059]]
```

<Figure size 432x288 with 0 Axes>




```
Fitting LDA
done in 0.027s
 Average and std CV score : 0.9574999999999999 +- 0.023184046238739265
Fitting QDA
done in 0.020s
 Average and std CV score : 0.9125 +- 0.023717082451262854
Fitting Logistic Regression
done in 0.163s
```

 Average and std CV score : 0.9625 +- 0.02850438562747848
Fitting Naive-Bayes
done in 0.012s
 Average and std CV score : 0.9475 +- 0.022912878474779196
Fitting K-nearest neighbour
done in 0.053s
 Average and std CV score : 0.9225 +- 0.030000000000000002

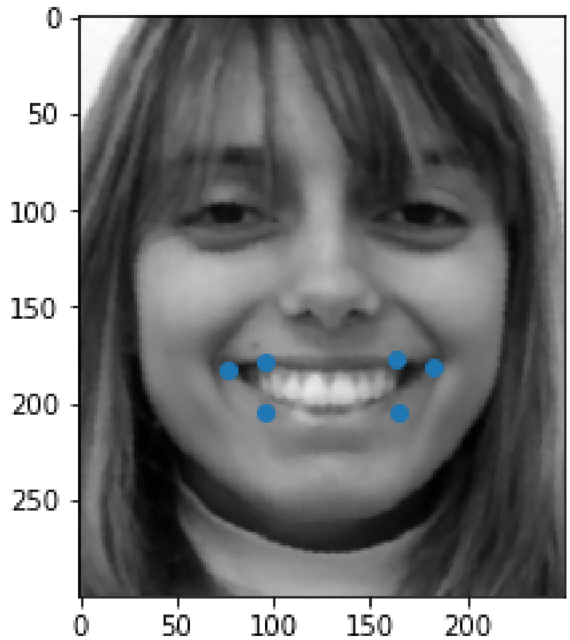
A second solution, would be to manually select few landmarks

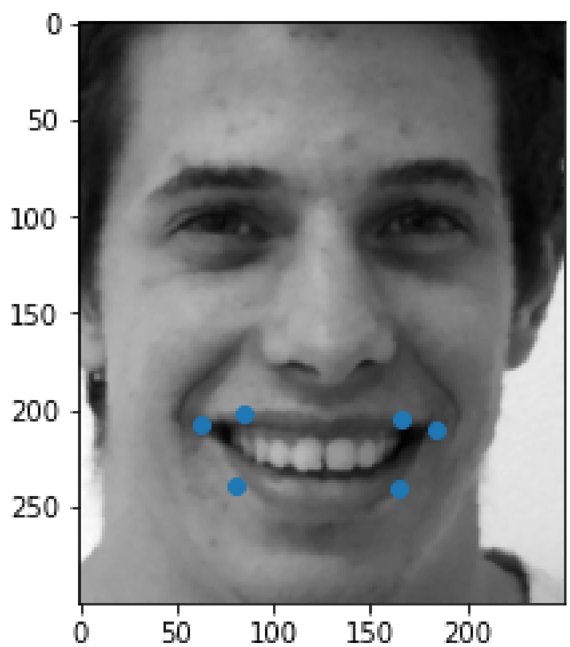
---

[26]:
```
# Select lateral landmarks mouth
select_land=[49,50,60,55,54,56]
indeces_central=[]
for k in range(0,len(select_land)):
    indeces_central.append(select_land[k]*2-2) # Remember that landmarks are␣
 ↪M*2 vectors (odds values are the x and even values are the y)
    indeces_central.append(select_land[k]*2-1)

indeces_central=np.array(indeces_central,dtype=int)
Ms=int(len(indeces_central)/2)
Xps=np.zeros((N,Ms*dim))
XpsGPA=np.zeros((N,Ms*dim))
for i in range(0,N):
    XpsGPA[i,:]=XpGPA[i,indeces_central]
    Xps[i,:]=Xp[i,indeces_central]

Yps=Yp

print('Number of subjects N is: ', XpsGPA.shape[0], ' ; number of features is:␣
 ↪',  XpsGPA.shape[1] )
```

```
Number of subjects N is:  400  ; number of features is:  12
```

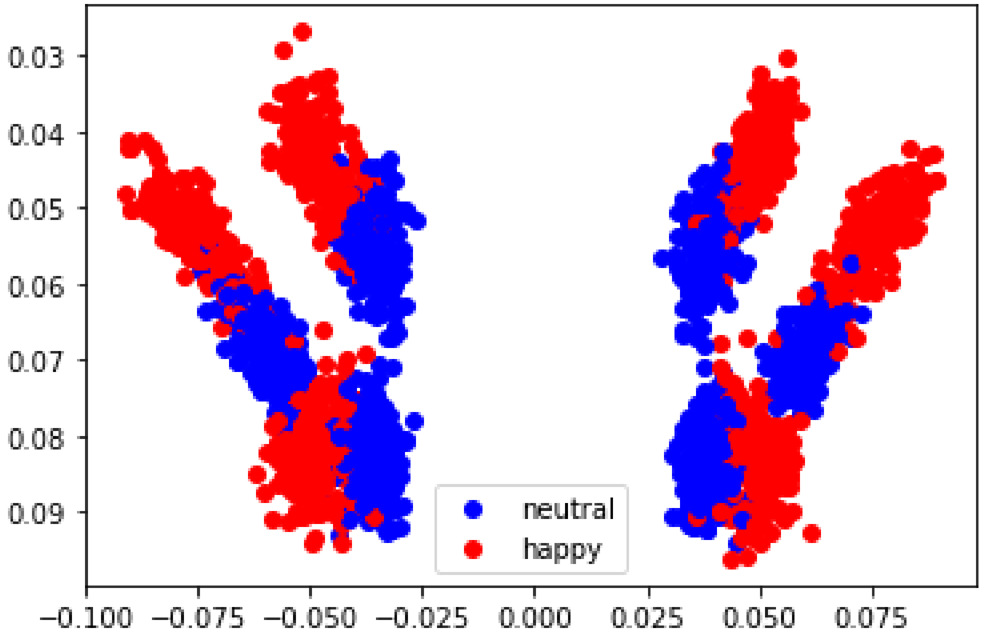
[27]:
```
# plot two test images
for i in range(0,2):
    image = Imagesp[i,:,:]
    plt.figure()
    plt.imshow(image, cmap='gray', origin='upper')
    landmark=Xps[i,:]
    x=landmark[::2]
    y=landmark[1::2]
    plt.plot(x,y,'o')
    plt.show()
```





[28]:
```
# Plot only selected landmarks
plt.figure()
for i in range(0,N):
    landmark=XpsGPA[i]
```

```
    x=landmark[::2]
    y=landmark[1::2]
    if Yps[i].astype(int)==0:
        neutral=plt.scatter(x, y, c='b')
    else:
        happy=plt.scatter(x, y, c='r')

plt.legend((neutral,happy),('neutral','happy'))
plt.gca().invert_yaxis()
```



[29]:
```
# Fitting QDA
print("Fitting QDA")

t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=XpsGPA, y=np.ravel(Yps),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
 ↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=XpsGPA, y=np.ravel(Yps),cv=5)
```

```
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=XpsGPA, y=np.ravel(Yps),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))
```

```
Fitting QDA
done in 0.021s
 Average and std CV score : 0.96 +- 0.031024184114977135
Fitting Logistic Regression
done in 0.039s
 Average and std CV score : 0.945 +- 0.016955824957813174
Fitting Naive-Bayes
done in 0.010s
 Average and std CV score : 0.9375 +- 0.019364916731037066
```

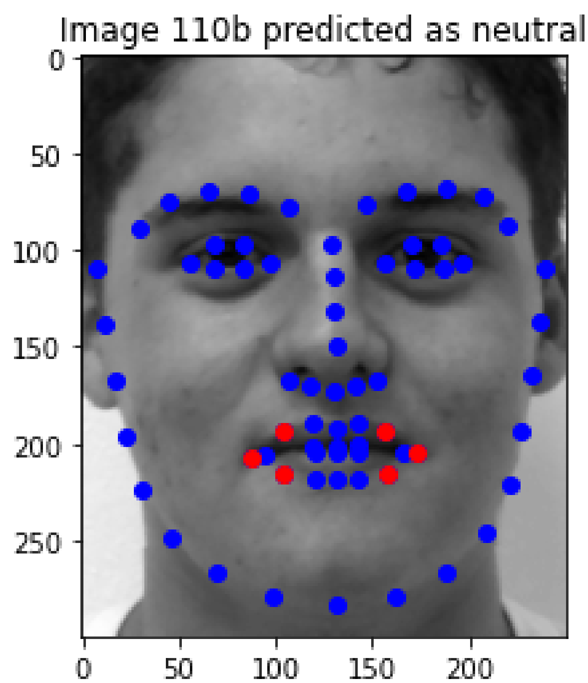
[30]:

```
# Fitting LDA
print("Fitting LDA")
lda = LinearDiscriminantAnalysis()
lda_validate = cross_validate(lda,X=XpsGPA, y=np.ravel(Yps), cv=5, n_jobs=-1,␣
↪return_train_score=True, return_estimator=True )
print(" Average and std train score : {0} +- {1}".
↪format(lda_validate['train_score'].mean(), lda_validate['train_score'].std()␣
↪))
print(" Average and std test score : {0} +- {1}".
↪format(lda_validate['test_score'].mean(), lda_validate['test_score'].std() ))

# Let's look for the best CV model (the one with the best test score)
best_estimator=lda_validate['estimator'][np.argmax(lda_validate['test_score'])]
C=best_estimator.predict(XpsGPA)

# Let's find the images where it did a mistake
error=np.ravel(np.array(np.where(np.abs(C-np.ravel(Yps)))))
if len(error)>5:
    kk=5
else:
    kk=len(error)
```

```
Fitting LDA
 Average and std train score : 0.95625 +- 0.006846531968814562
 Average and std test score : 0.9400000000000001 +- 0.021505813167606556
```

Let's plot some images where the best model was wrong.

[31]:
```
# plot error images
for i in range(0,kk):
    image = Imagesp[error[i],:,:]
    plt.figure()
    plt.imshow(image, cmap='gray', origin='upper')
    landmarkALL=Xp[error[i],:]
    landmark=Xps[error[i],:]
    xALL=landmarkALL[::2]
    yALL=landmarkALL[1::2]
    x=landmark[::2]
    y=landmark[1::2]
    plt.plot(xALL,yALL,'ob')
    plt.plot(x,y,'or')
    if  C[error[i]]==0:
        plt.title('Image ' + Namesp[error[i]] + ' predicted as neutral')
    elif C[error[i]]==1:
        plt.title('Image ' + Namesp[error[i]] + ' predicted as happy')
    plt.show()
```



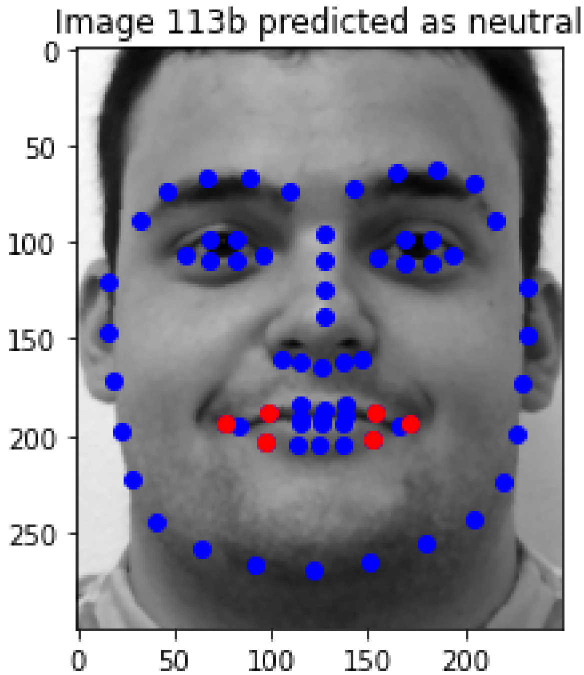


Image 113b predicted as neutral
0
50
100
150
200
250
0
50
100
150
200

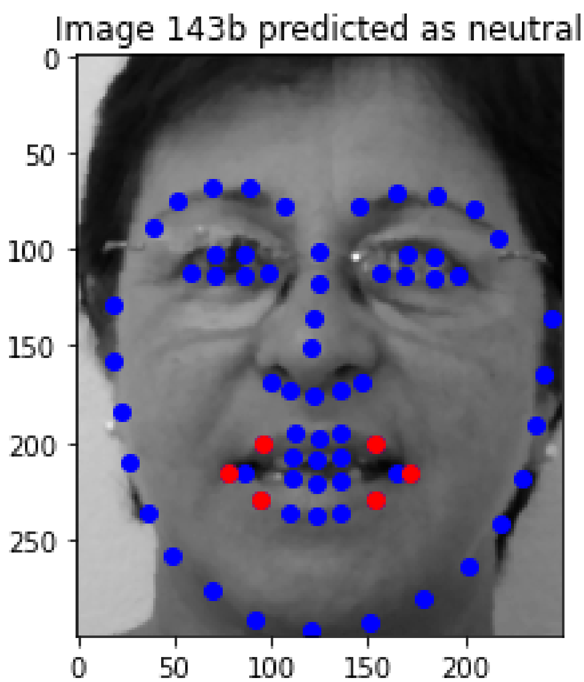
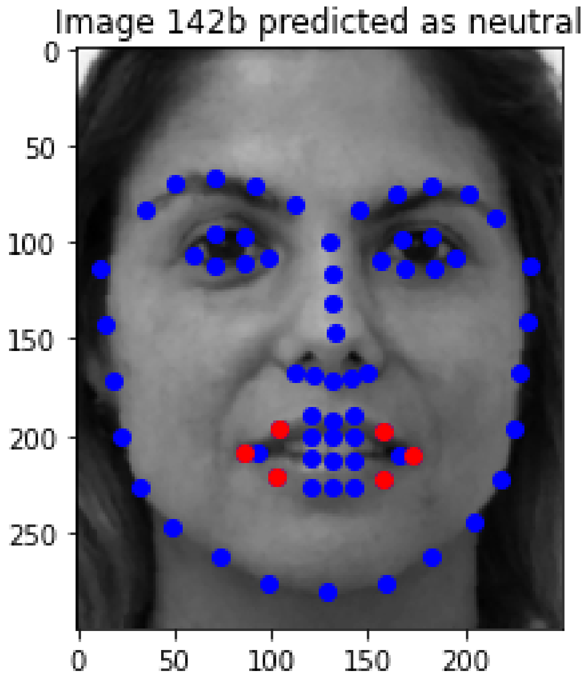
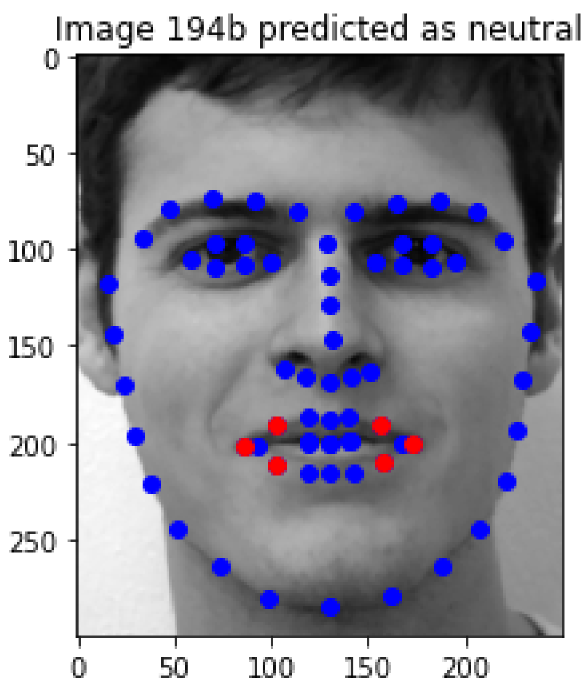


Image 143b predicted as neutral
0
50
100
150
200
250
0
50
100
150
200





Question (IMP+IMH): Comment the results. Why did the algorithm make a mistake ? Would you choose other landmarks ? Try at least another combination of landmarks

Answer:

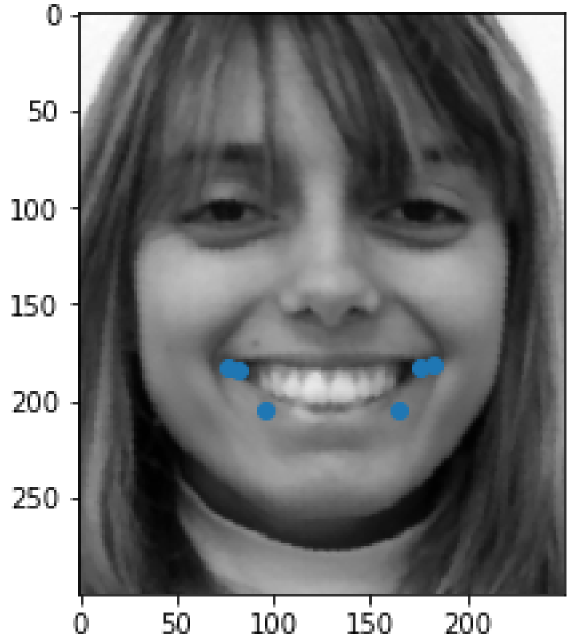
I would choose a bit different landmarks since it makes sense to choose the edge of the smile has the landmarks as I did.

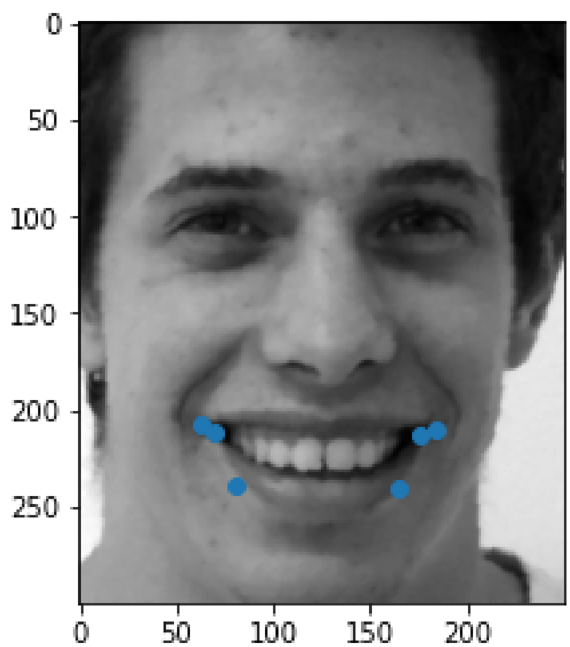
[32]:
```
# Select lateral landmarks mouth
select_land=[49,61,60,56,65,55]
indeces_central=[]
for k in range(0,len(select_land)):
    indeces_central.append(select_land[k]*2-2) # Remember that landmarks are␣
 ↪M*2 vectors (odds values are the x and even values are the y)
    indeces_central.append(select_land[k]*2-1)

indeces_central=np.array(indeces_central,dtype=int)
Ms=int(len(indeces_central)/2)
Xps=np.zeros((N,Ms*dim))
XpsGPA=np.zeros((N,Ms*dim))
for i in range(0,N):
    XpsGPA[i,:]=XpGPA[i,indeces_central]
    Xps[i,:]=Xp[i,indeces_central]

Yps=Yp

print('Number of subjects N is: ', XpsGPA.shape[0], ' ; number of features is:␣
 ↪',  XpsGPA.shape[1] )
# plot two test images
for i in range(0,2):
    image = Imagesp[i,:,:]
    plt.figure()
    plt.imshow(image, cmap='gray', origin='upper')
    landmark=Xps[i,:]
    x=landmark[::2]
    y=landmark[1::2]
    plt.plot(x,y,'o')
    plt.show()
# Plot only selected landmarks
plt.figure()
for i in range(0,N):
    landmark=XpsGPA[i]
    x=landmark[::2]
    y=landmark[1::2]
    if Yps[i].astype(int)==0:
        neutral=plt.scatter(x, y, c='b')
    else:
        happy=plt.scatter(x, y, c='r')

plt.legend((neutral,happy),('neutral','happy'))
plt.gca().invert_yaxis()
```

```
# Fitting QDA
print("Fitting QDA")

t0 = time()
qda = QuadraticDiscriminantAnalysis()
qda_score = cross_val_score(qda,X=XpsGPA, y=np.ravel(Yps),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(qda_score.mean(),␣
↪qda_score.std() ))

# Fitting Logistic-regression
print("Fitting Logistic Regression")
t0 = time()
logit = LogisticRegression(solver='lbfgs')
logit_score = cross_val_score(logit,X=XpsGPA, y=np.ravel(Yps),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(logit_score.mean(),␣
↪logit_score.std() ))

# Fitting Naive-Bayes
print("Fitting Naive-Bayes")
t0 = time()
GNB = GaussianNB()
GNB_score = cross_val_score(GNB,X=XpsGPA, y=np.ravel(Yps),cv=5)
print("done in %0.3fs" % (time() - t0))
print(" Average and std CV score : {0} +- {1}".format(GNB_score.mean(),␣
↪GNB_score.std() ))
# Fitting LDA
print("Fitting LDA")
lda = LinearDiscriminantAnalysis()
lda_validate = cross_validate(lda,X=XpsGPA, y=np.ravel(Yps), cv=5, n_jobs=-1,␣
↪return_train_score=True, return_estimator=True )
print(" Average and std train score : {0} +- {1}".
↪format(lda_validate['train_score'].mean(), lda_validate['train_score'].std()␣
↪))
print(" Average and std test score : {0} +- {1}".
↪format(lda_validate['test_score'].mean(), lda_validate['test_score'].std() ))

# Let's look for the best CV model (the one with the best test score)
best_estimator=lda_validate['estimator'][np.argmax(lda_validate['test_score'])]
C=best_estimator.predict(XpsGPA)

# Let's find the images where it did a mistake
error=np.ravel(np.array(np.where(np.abs(C-np.ravel(Yps)))))
if len(error)>5:
    kk=5
```

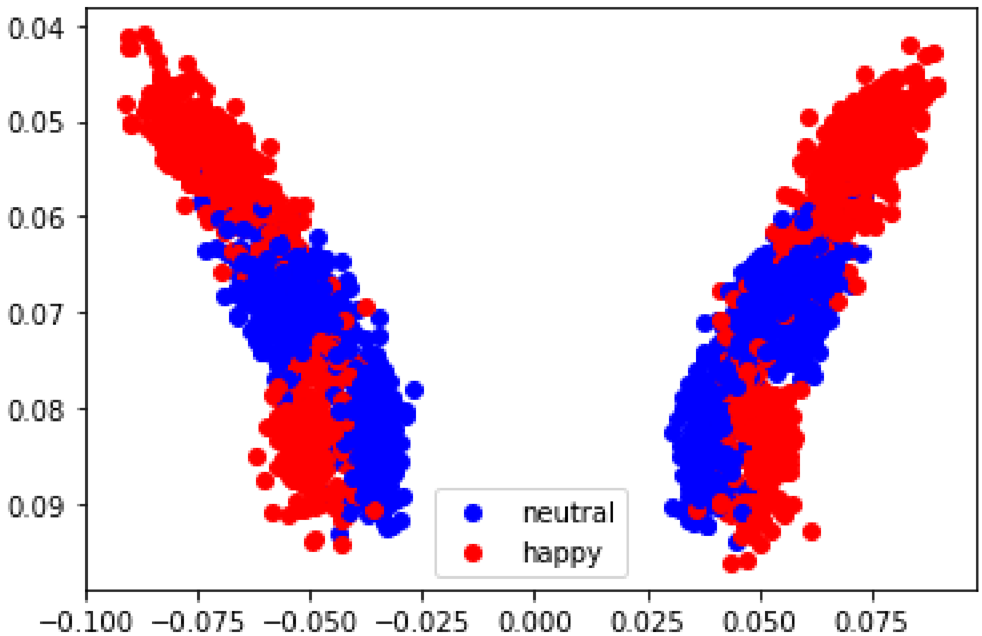
```
else:
    kk=len(error)
```

```
Number of subjects N is:  400  ; number of features is:  12
```





```
Fitting QDA
done in 0.007s
 Average and std CV score : 0.9625 +- 0.02091650066335188
Fitting Logistic Regression
done in 0.018s
 Average and std CV score : 0.9425000000000001 +- 0.016955824957813174
Fitting Naive-Bayes
done in 0.010s
 Average and std CV score : 0.9400000000000001 +- 0.02893959225697557
Fitting LDA
 Average and std train score : 0.9568749999999999 +- 0.008244316223920583
 Average and std test score : 0.9525 +- 0.026692695630078273
```



Here, we use Nested Cross-Validation for finding the generalization error and the best K value

---

[33]:
```
# Fitting K-nearest neighbour with Nested Cross-Validation

print("Fitting K-nearest neighbour with Nested CV")
t0 = time()
neigh = KNeighborsClassifier()
parameters = {'n_neighbors':[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]}
inner_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=666) # we fix␣
↪the random state to always have the same results if we relaunch the code
```

```
outer_cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=666)
# Nested CV with parameter optimization
clf = GridSearchCV(estimator=neigh, param_grid=parameters, cv=inner_cv)
nested_CV = cross_validate(estimator=clf, X=XpsGPA, y=np.ravel(Yps),␣
↪cv=outer_cv,return_train_score=True, return_estimator=True, n_jobs=-1)
print("done in %0.3fs" % (time() - t0))
print("Average and std Nested Cv train score : {0} +- {1}".
↪format(nested_CV['train_score'].mean(), nested_CV['train_score'].std() ))
print("Average and std Nested Cv test score : {0} +- {1}".
↪format(nested_CV['test_score'].mean(), nested_CV['test_score'].std() ))
```

```
Fitting K-nearest neighbour with Nested CV
done in 0.870s
Average and std Nested Cv train score : 0.95625 +- 0.006249999999999978
Average and std Nested Cv test score : 0.9550000000000001 +-
0.020310096011589906
```

Question (IMP+IMH): Are Training and Test scores similar ? What does it mean ?

Answer:

yes they are similar that's means we have a very good moodel with no overfitting. The perfect model.

Question (IMP+IMH) (OPTIONAL): Please propose at least another set of features using landmarks and/or pixel intensities of the images and test its discriminative power

[ ]:

```

```