TP_SVM

December 8, 2020

1 Skin lesion classification

Deadline: Upload this notebook (rename it as 'TP3-SVM-YOUR-SURNAME.ipynb') to the Moodle before the 9th of December 2020 (23h59).

All questions are for both IMP and IMH

Context A skin lesion is defined as a superficial growth or patch of the skin that is visually different and/or has a different texture than its surrounding area. Skin lesions, such as moles or birthmarks, can degenerate and become melanoma, one of the deadliest skin cancer. Its incidence has been increasing during the last decades, especially in the areas mostly populated by white people.

The most effective treatment is an early detection followed by surgical excision. This is why several approaches for melanoma detection have been proposed in the last years (non-invasive computer-aided diagnosis (CAD)).

Goal The goal of this practical session is to classify images of skin lesions as either benign or melanoma using machine learning algorithms. In order to do that, you will have at your disposal a set of 30 features already extracted from 600 dermoscopic images (both normal skin lesions and melanoma from the ISIC database - https://isic-archive.com/). These features characterize the Asymmetry, the Border irregularity, the Colour and the Dimension of the lesion (the so-called ABCD rule).

The features are: - shape asimmetry (f0 and f1) - difference in colors between center and periphery of the image (f2, f3, f4, f27, f28, f29) - geometry (f5, f6, f7) - other features related to eccentricity,entropy, mean, standard deviation and maximum value of each channel in RGB and HSV (f8,...,f24) - asimmetry of color intensity (f25, f26)

Features are computed using *manually checked segmentations* and following *Ganster et al. 'Automated melanoma recognition', IEEE TMI, 2001* and *Zortea et al. 'Performance of a dermoscopy-based computer vision system for the diagnosis of pigmented skin lesions compared with visual evaluation by experienced dermatologists', Artificial Intelligence in Medicine, 2014.*

First load all necessary packages

[1]:
```
import os
import numpy as np
import pandas as pd
from skimage.io import imread
from time import time
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.axes_grid1 import AxesGrid
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report
from sklearn.metrics import confusion_matrix, f1_score, make_scorer
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split
from sklearn.model_selection import  cross_val_score
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import GridSearchCV, KFold, cross_validate
from sklearn.linear_model import Perceptron
from sklearn.decomposition import PCA

%matplotlib inline

import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
warnings.simplefilter(action='ignore', category=FutureWarning)

from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings(action='ignore', category=ConvergenceWarning)

# Code from scikit-learn
import itertools
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)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)
```

```
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="black" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

Then load the data from my Google Drive

[2]:
```
# from google_drive_downloader import GoogleDriveDownloader as gdd
# gdd.
 ↪download_file_from_google_drive(file_id='18hrQVGBCfW7SKTnzmWUONo8iowBsi1DL',
# dest_path='./data/features.csv')
# gdd.
 ↪download_file_from_google_drive(file_id='1iQZdUiuK_FwZ7mik7LB3eN_H_IUc5l7b',
# dest_path='./data/im/nevus-seg.jpg')
# gdd.
 ↪download_file_from_google_drive(file_id='1_TeYzLLDoKbPX4xXAOAM_mQiT2nLHgvp',
# dest_path='./data/im/nevus.jpg')
# gdd.
 ↪download_file_from_google_drive(file_id='1B2Ol92mBcHN6ah3bpoucBbBbHkPMGC8D',
# dest_path='./data/im/melanoma-seg.jpg')
# gdd.
 ↪download_file_from_google_drive(file_id='1yZ46UzGhwO7g5T8397JpewBl6UqgRo5J',
# dest_path='./data/im/melanoma.jpg')
```

Or from yout local computer. Please download the 'data' folder in the same folder as your notebook and do not modifiy it.

Then read the data

[3]:
```
## Read data
Working_directory="./data/"
df = pd.read_csv(Working_directory + 'features.csv') # reading data
y = df['Malignant'].values # 1 for Melanoma and 0 for healthy
class_names = ["healthy","melanoma"]
X = df.iloc[:,3:33].values # Features
N,M=X.shape
print('Number of images: {0}; Number of features per image: {1}'.format(N,M))
print('Number of healthy nevus: {0}; Number of melanoma: {1}'.format(N-np.
 ↪sum(y), np.sum(y)))
```

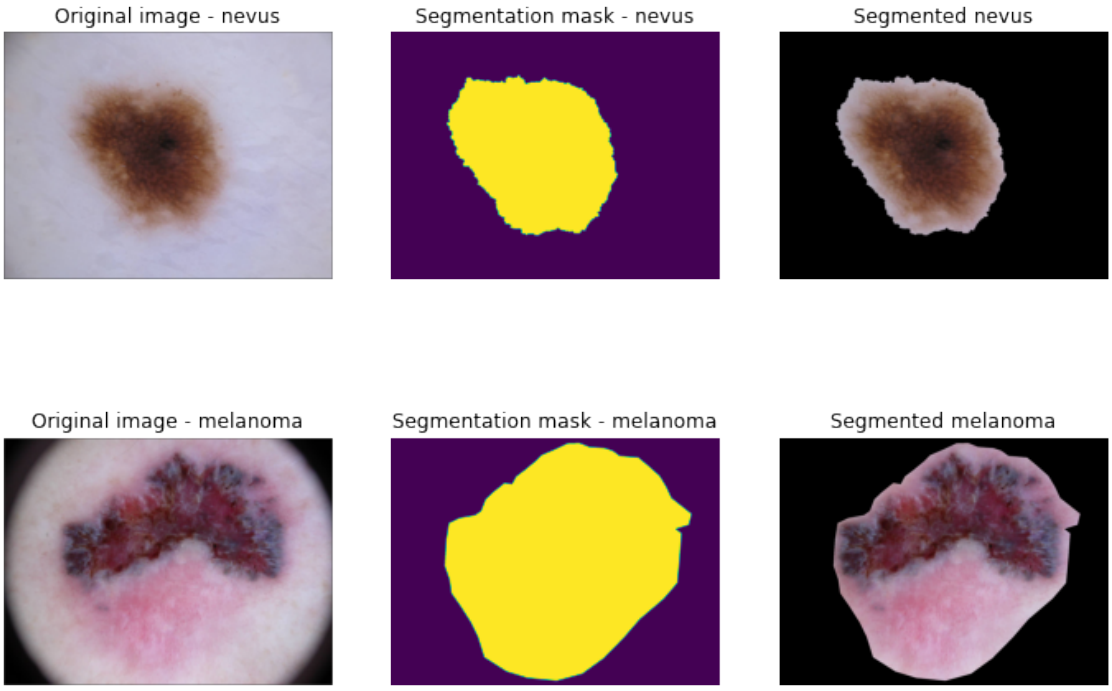
```
Number of images: 600; Number of features per image: 30
Number of healthy nevus: 485; Number of melanoma: 115
```

[4]:
```
## Plot two examples of nevus and melanoma
print('Two examples of healthy nevus and melanoma')
nevus = imread(Working_directory + 'im/nevus.jpg')
nevus_Segmentation = imread(Working_directory + 'im/nevus-seg.jpg')
nevus_Segmentation_boolean = (nevus_Segmentation/255).astype(np.uint8) # To get␣
 ↪uint8 (integer numbers)
nevus_Segmentation_3D = np.expand_dims(nevus_Segmentation_boolean, axis=2) # To␣
 ↪have a binary mask for the three channels (RGB)
nevus_mul_mask = (nevus_Segmentation_3D*nevus) # we apply the binary mask to␣
 ↪all channels pixel-wise

fig = plt.figure(figsize=(12, 12)) # size of the figure
grid = AxesGrid(fig, 111,
                nrows_ncols = (1, 3),
                axes_pad = 0.5) # code to create subplots
grid[0].imshow(nevus)
grid[0].axis('off')
grid[0].set_title('Original image - nevus')
grid[1].imshow(nevus_Segmentation)
grid[1].axis('off')
grid[1].set_title("Segmentation mask - nevus")
grid[2].imshow(nevus_mul_mask)
grid[2].axis('off')
grid[2].set_title("Segmented nevus")

###

melanoma = imread(Working_directory + 'im/melanoma.jpg')
melanoma_Segmentation = imread(Working_directory + 'im/melanoma-seg.jpg')
melanoma_Segmentation_boolean = (melanoma_Segmentation/255).astype(np.uint8) #␣
 ↪To get uint8 (integer numbers)
melanoma_Segmentation_3D = np.expand_dims(melanoma_Segmentation_boolean,␣
 ↪axis=2) # To have a binary mask for the three channels (RGB)
melanoma_mul_mask = (melanoma_Segmentation_3D*melanoma) # we apply the binary␣
 ↪mask to all channels pixel-wise

fig = plt.figure(figsize=(12, 12)) # size of the figure
grid = AxesGrid(fig, 111,
                nrows_ncols = (1, 3),
                axes_pad = 0.5) # code to create subplots
grid[0].imshow(melanoma)
grid[0].axis('off')
grid[0].set_title('Original image - melanoma')
grid[1].imshow(melanoma_Segmentation)
grid[1].axis('off')
grid[1].set_title("Segmentation mask - melanoma")
grid[2].imshow(melanoma_mul_mask)
```

```
grid[2].axis('off')
grid[2].set_title("Segmented melanoma")
```

```
Two examples of healthy nevus and melanoma
```

[4]:
```
Text(0.5, 1.0, 'Segmented melanoma')
```



Now, as in the previous practical session you should shuffle the data randomly

[5]:
```
# 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)

Xp=X[indeces]
yp=y[indeces]
```

As we have already seen, it might be very important to scale the data such that each feature has, for instance, average equal to 0 and unit variance.

[6]:
```
# Scale data (each feature will have average equal to 0 and unit variance)
scaler = StandardScaler()
scaler.fit(Xp)
```

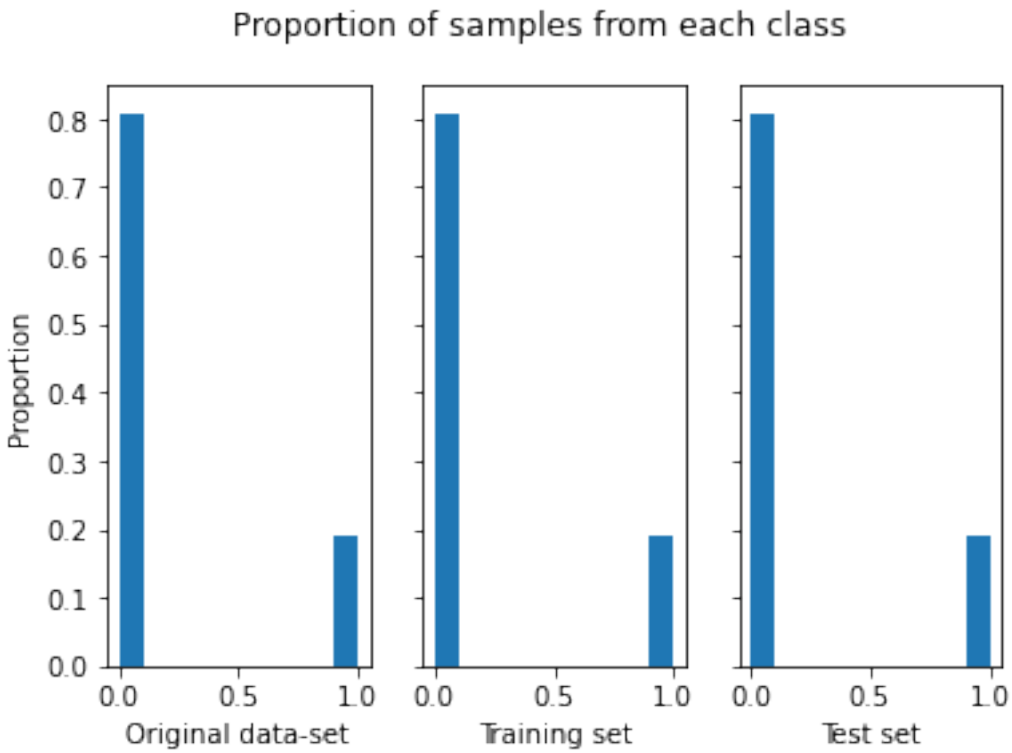
```
dist_average_scale=scaler.transform(Xp)
print('Scaler')
print('Number of subjects N is: ', Xp.shape[0], ' ; number of features is: ',␣
↪Xp.shape[1] )
```

```
Scaler
Number of subjects N is:  600  ; number of features is:  30
```

We should now test the discriminative power of our features. Fist, let divide the entire data-set into training and test set using the `stratify` option. This will preserve the original proportion between nevus and melanoma also in the training and test set. You can check that from the plot.

[7]:
```
# Create training and test set
X_train, X_test, y_train, y_test = train_test_split(Xp, yp, test_size=0.33,␣
↪random_state=42,stratify=yp)

fig, axs = plt.subplots(1, 3, sharey=True)
fig.suptitle('Proportion of samples from each class')
axs[0].hist(yp,weights=np.ones_like(yp)/len(yp))
axs[0].set_xlabel('Original data-set')
axs[1].hist(y_train,weights=np.ones_like(y_train)/len(y_train))
axs[1].set_xlabel('Training set')
axs[2].hist(y_test,weights=np.ones_like(y_test)/len(y_test))
axs[2].set_xlabel('Test set')
axs[0].set_ylabel('Proportion')
```

[7]:
```
Text(0, 0.5, 'Proportion')
```



Now, use two simple classification algorithms, for instance LDA and QDA, and look at the confusion matrices.

Question: Comment the results.

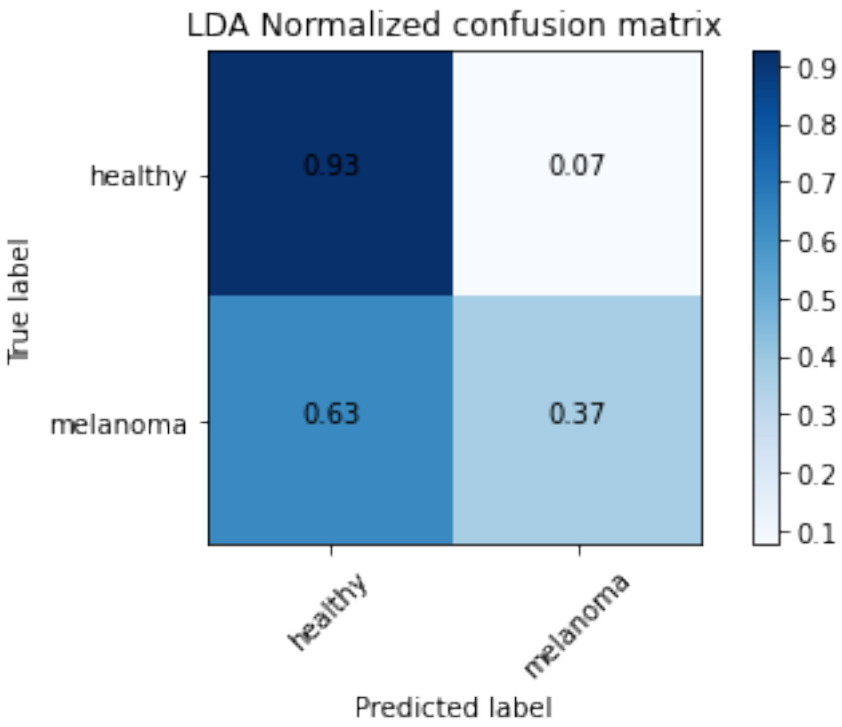
Answer:

The confusion matrices looks very similar and the accuracy is similar as well.

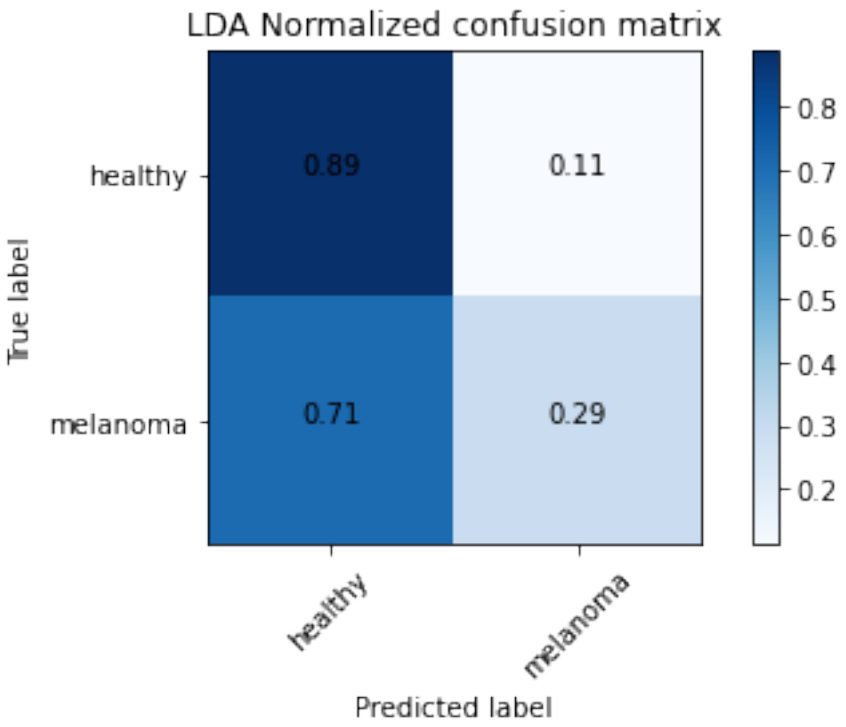
[8]:
```
# Fitting LDA
print("Fitting LDA to training set")
lda = LinearDiscriminantAnalysis()
lda.fit(X_train, y_train)
y_pred = lda.predict(X_test)
print(classification_report(y_test, y_pred))

# 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,
                      title='LDA Normalized confusion matrix')
plt.show()
```

```python
# Fitting QDA
print("Fitting QDA to training set")
qda = QuadraticDiscriminantAnalysis()
qda.fit(X_train, y_train)
y_pred = qda.predict(X_test)
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,
                      title='LDA Normalized confusion matrix')
plt.show()
```

```
Fitting LDA to training set
              precision    recall  f1-score   support

           0       0.86      0.93      0.89       160
           1       0.54      0.37      0.44        38

    accuracy                           0.82       198
   macro avg       0.70      0.65      0.66       198
weighted avg       0.80      0.82      0.80       198

Normalized confusion matrix
[[0.925      0.075     ]
 [0.63157895 0.36842105]]
```




```
Fitting QDA to training set
              precision    recall  f1-score   support

           0       0.84      0.89      0.86       160
           1       0.38      0.29      0.33        38

    accuracy                           0.77       198
   macro avg       0.61      0.59      0.60       198
weighted avg       0.75      0.77      0.76       198

Normalized confusion matrix
[[0.8875     0.1125    ]
 [0.71052632 0.28947368]]
```



The results you obtained are based on a precise subdivision of your data into training and test. This can thus bias your results. Which technique could you use instead ? Test it with LDA, QDA and K-NN.

[9]:
```
# Fitting LDA
print("Fitting LDA")
t0 = time()
lda = LinearDiscriminantAnalysis()
lda_score = cross_val_score(lda,X=Xp, 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=Xp, 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 K-nearest neighbour
print("Fitting K-nearest neighbour")
```

```
t0 = time()
neigh = KNeighborsClassifier(n_neighbors=3)
neigh_score = cross_val_score(neigh,X=Xp, 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
done in 0.031s
 Average and std CV score : 0.8099999999999999 +- 0.027080128015453172
Fitting QDA
done in 0.010s
 Average and std CV score : 0.7899999999999999 +- 0.014337208778404373
Fitting K-nearest neighbour
done in 0.046s
 Average and std CV score : 0.7516666666666667 +- 0.020682789409984786
```

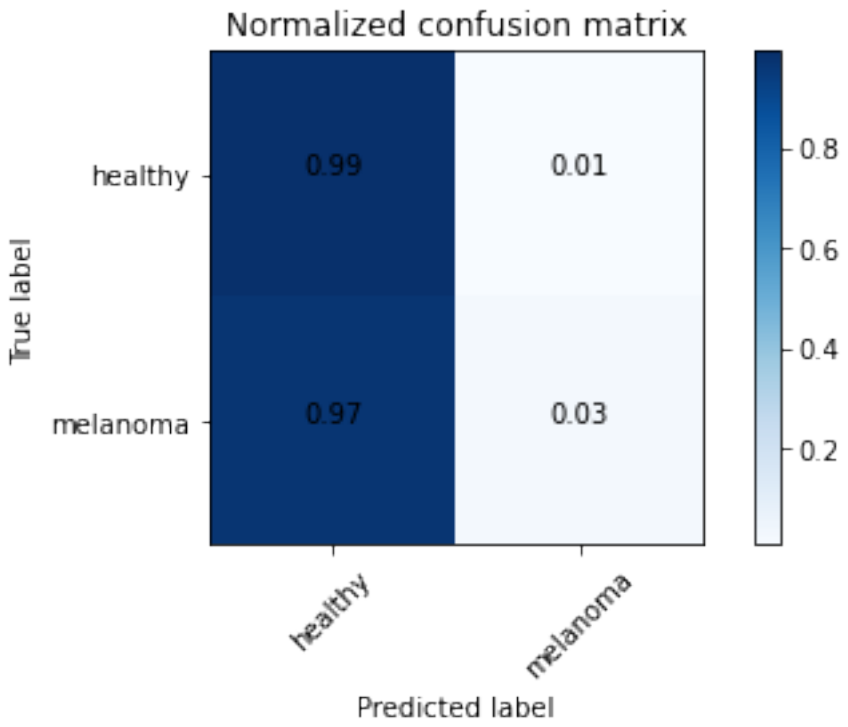
When using K-NN, instead than fixing the number of nearest neighbours, we could also estimate the best value using Cross Validation. Do it and plot the confusion matrix. Do you notice anything strange ? Why in your opinion do you have this kind of result ?

Answer:

We still miss classification the melanoma but we do very well with the healthy samples, the dataset is heavily imbalanced

[10]:
```
# Looking for the best hyperparameters
p_grid_KNN = {'n_neighbors': [1,2,3,4,5,6,7,8,9,10]}
KNN = KNeighborsClassifier()
grid_KNN = GridSearchCV(estimator=KNN, param_grid=p_grid_KNN,␣
 ↪scoring="accuracy", cv=5)
grid_KNN.fit(X_train, y_train)
print("Best training Score: {}".format(grid_KNN.best_score_))
print("Best training params: {}".format(grid_KNN.best_params_))
y_pred = grid_KNN.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Best training Score: 0.8084876543209877
Best training params: {'n_neighbors': 10}
Normalized confusion matrix
[[0.99375    0.00625   ]
 [0.97368421 0.02631579]]
```

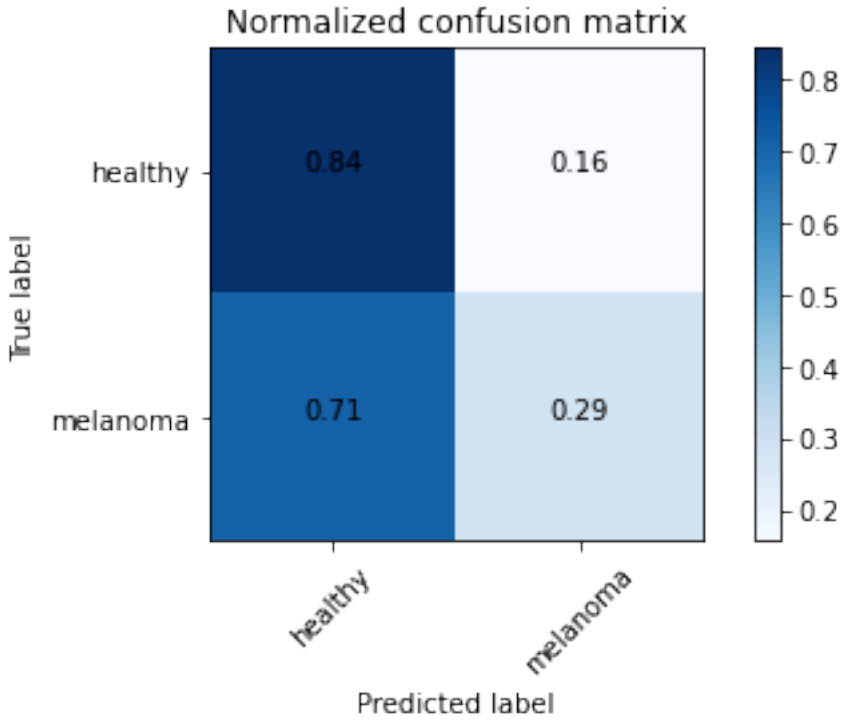


In order to deal with this problem we have two possible solutions.

First: Please look at this webpage (https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter) and try MORE APPROPRIATE scoring functions than accuracy when looking for the best K value of K-NN (thus within the Cross Validation as before..).

[11]:
```
# Looking for the best hyperparameters
p_grid_KNN = {'n_neighbors': [1,2,3,4,5,6,7,8,9,10]}
KNN = KNeighborsClassifier()
# let us try with f1 score
grid_KNN = GridSearchCV(estimator=KNN, param_grid=p_grid_KNN, scoring="f1",␣
 ↪cv=5)
grid_KNN.fit(X_train, y_train)
print("Best training Score: {}".format(grid_KNN.best_score_))
print("Best training params: {}".format(grid_KNN.best_params_))
y_pred = grid_KNN.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Best training Score: 0.2774888652779119
Best training params: {'n_neighbors': 1}
Normalized confusion matrix
[[0.84375    0.15625   ]
 [0.71052632 0.28947368]]
```



Second: when dealing with such a problem (the one you should find !) a possible solution would be to oversample a class (which one in your opinion ?) Please look at this web page for more information (https://imbalanced-learn.readthedocs.io/en/stable/over_sampling.html) and try at least one naive random over-sampling (look at the following code…).

[12]:
```
from imblearn.over_sampling import RandomOverSampler
from collections import Counter
ros = RandomOverSampler(random_state=0)
X_resampled, y_resampled = ros.fit_resample(X_train, y_train)
print(sorted(Counter(y_resampled).items()))
```

```
[(0, 325), (1, 325)]
```

Let's look for the best K in KNN (as before using Cross validation) but this time on the new training set.

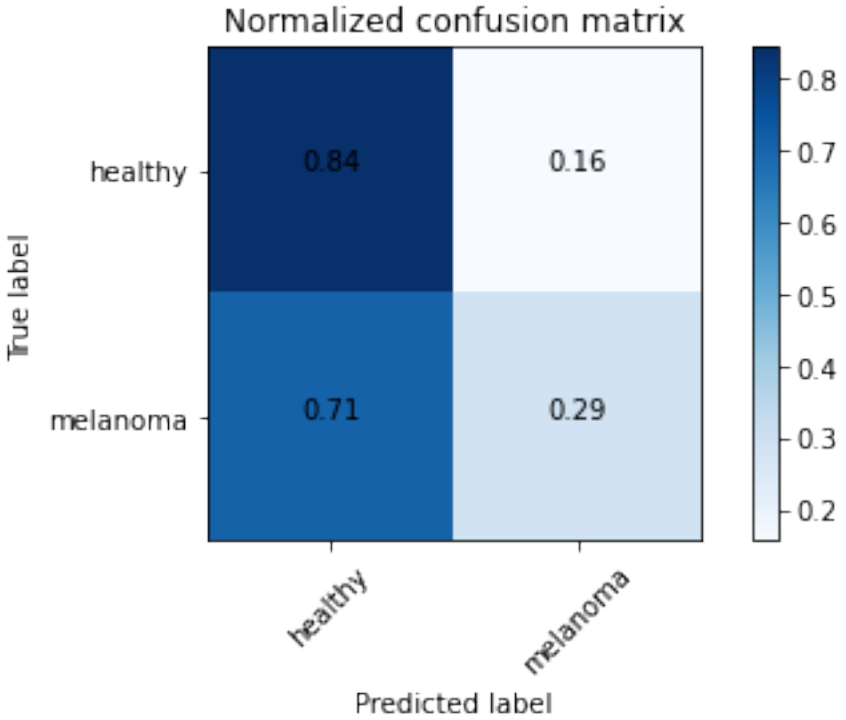
Question: Are the results better ? Do they change now if you modify the scoring function ? Why ?

Answer:

A bit better but I think we can do better with f1_weighted but I failed to implement it small error. and now the dataset is more balanced

[13]:
```
p_grid_KNN = {'n_neighbors': [1,2,3,4,5,6,7,8,9,10]}
KNN = KNeighborsClassifier()
grid_KNN = GridSearchCV(estimator=KNN, param_grid=p_grid_KNN,␣
 ↪scoring="accuracy", cv=5)
grid_KNN.fit(X_resampled, y_resampled)
print("Best training Score: {}".format(grid_KNN.best_score_))
print("Best training params: {}".format(grid_KNN.best_params_))
y_pred = grid_KNN.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Best training Score: 0.8892307692307693
Best training params: {'n_neighbors': 1}
Normalized confusion matrix
[[0.84375    0.15625   ]
 [0.71052632 0.28947368]]
```



Let's use the techniques seen today: Perceptron and linear SVM.

[14]:
```
# Fitting Perceptron
print("Fitting Perceptron")
Perc = Perceptron()
Perc_cv = cross_validate(Perc,Xp,␣
 ↪yp,cv=5,scoring='accuracy',return_train_score=True)
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['train_score'].mean(), Perc_cv['train_score'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['test_score'].mean(), Perc_cv['test_score'].std() ))


# Fitting linear SVM
print("Fitting Linear SVM")
Lsvm = LinearSVC()
Lsvm_cv = cross_validate(Lsvm,Xp,␣
 ↪yp,cv=5,scoring='accuracy',return_train_score=True)
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['train_score'].mean(), Lsvm_cv['train_score'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['test_score'].mean(), Lsvm_cv['test_score'].std() ))
```

```
Fitting Perceptron
 Average and std TRAIN CV accuracy : 0.52125 +- 0.25347720695250775
 Average and std TEST CV accuracy : 0.4966666666666667 +- 0.26223399220289245
Fitting Linear SVM
 Average and std TRAIN CV accuracy : 0.6891666666666667 +- 0.22698354076403376
 Average and std TEST CV accuracy : 0.6916666666666667 +- 0.22527760652137618
```

We can easily use different scoring functions within the cross validate function of scikit-learn. Check the code.

[15]:
```
# Fitting Perceptron
print("Fitting Perceptron")
Perc = Perceptron()
Perc_cv = cross_validate(Perc,Xp, yp,cv=5,scoring=('accuracy',␣
 ↪'f1'),return_train_score=True)
print(Perc_cv.keys())
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['train_accuracy'].mean(), Perc_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['test_accuracy'].mean(), Perc_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Perc_cv['train_f1'].
 ↪mean(), Perc_cv['train_f1'].std() ))
```

```
print(" Average and std TEST CV f1 : {0} +- {1}".format(Perc_cv['test_f1'].
 ↪mean(), Perc_cv['test_f1'].std() ))


# Fitting linear SVM
print("Fitting Linear SVM")
Lsvm = LinearSVC()
Lsvm_cv = cross_validate(Lsvm,Xp, yp,cv=5,scoring=('accuracy',␣
 ↪'f1'),return_train_score=True)
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['train_accuracy'].mean(), Lsvm_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['test_accuracy'].mean(), Lsvm_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Lsvm_cv['train_f1'].
 ↪mean(), Lsvm_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Lsvm_cv['test_f1'].
 ↪mean(), Lsvm_cv['test_f1'].std() ))
```

```
Fitting Perceptron
dict_keys(['fit_time', 'score_time', 'test_accuracy', 'train_accuracy',
'test_f1', 'train_f1'])
 Average and std TRAIN CV accuracy : 0.52125 +- 0.25347720695250775
 Average and std TEST CV accuracy : 0.4966666666666667 +- 0.26223399220289245
 Average and std TRAIN CV f1 : 0.20804233645925302 +- 0.17087717695558147
 Average and std TEST CV f1 : 0.18947817858708949 +- 0.15637387360514374
Fitting Linear SVM
 Average and std TRAIN CV accuracy : 0.69125 +- 0.2373369323228992
 Average and std TEST CV accuracy : 0.6799999999999999 +- 0.23638480868655198
 Average and std TRAIN CV f1 : 0.13024350122618372 +- 0.15127847872690117
 Average and std TEST CV f1 : 0.06524822695035462 +- 0.13049645390070924
```

Question Please do the same on the oversampled data and compare the results with the previous ones.

Answer:

Done!

[16]:
```
X_resampled_All, y_resampled_All = ros.fit_resample(Xp, yp)

# Fitting Perceptron
print("Fitting Perceptron")
Perc = Perceptron()
Perc_cv = cross_validate(Perc,X_resampled_All,␣
 ↪y_resampled_All,cv=5,scoring=('accuracy', 'f1'),return_train_score=True)
print(Perc_cv.keys())
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['train_accuracy'].mean(), Perc_cv['train_accuracy'].std() ))
```

```
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['test_accuracy'].mean(), Perc_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Perc_cv['train_f1'].
 ↪mean(), Perc_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Perc_cv['test_f1'].
 ↪mean(), Perc_cv['test_f1'].std() ))


# Fitting linear SVM
print("Fitting Linear SVM")
Lsvm = LinearSVC()
Lsvm_cv = cross_validate(Lsvm,X_resampled_All,␣
 ↪y_resampled_All,cv=5,scoring=('accuracy', 'f1'),return_train_score=True)
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['train_accuracy'].mean(), Lsvm_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['test_accuracy'].mean(), Lsvm_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Lsvm_cv['train_f1'].
 ↪mean(), Lsvm_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Lsvm_cv['test_f1'].
 ↪mean(), Lsvm_cv['test_f1'].std() ))
```

```
Fitting Perceptron
dict_keys(['fit_time', 'score_time', 'test_accuracy', 'train_accuracy',
'test_f1', 'train_f1'])
 Average and std TRAIN CV accuracy : 0.5930412371134021 +- 0.027775327297115664
 Average and std TEST CV accuracy : 0.5845360824742267 +- 0.02272722441757588
 Average and std TRAIN CV f1 : 0.6187776836165093 +- 0.0875303325352087
 Average and std TEST CV f1 : 0.6153272850045957 +- 0.09409530188069755
Fitting Linear SVM
 Average and std TRAIN CV accuracy : 0.5353092783505154 +- 0.07061855670103094
 Average and std TEST CV accuracy : 0.5247422680412371 +- 0.049484536082474224
 Average and std TRAIN CV f1 : 0.1238239757207891 +- 0.24764795144157822
 Average and std TEST CV f1 : 0.1070063694267516 +- 0.21401273885350316
```

We can also ask to save the estimated models at each split (i.e. fold) with the option `return_estimator=True`. Using the perceptron, we will look for the best model using the over-sampled training data and check the confusion matrix on the test data.

Question Do it the same with the linear SVM.

Answer:

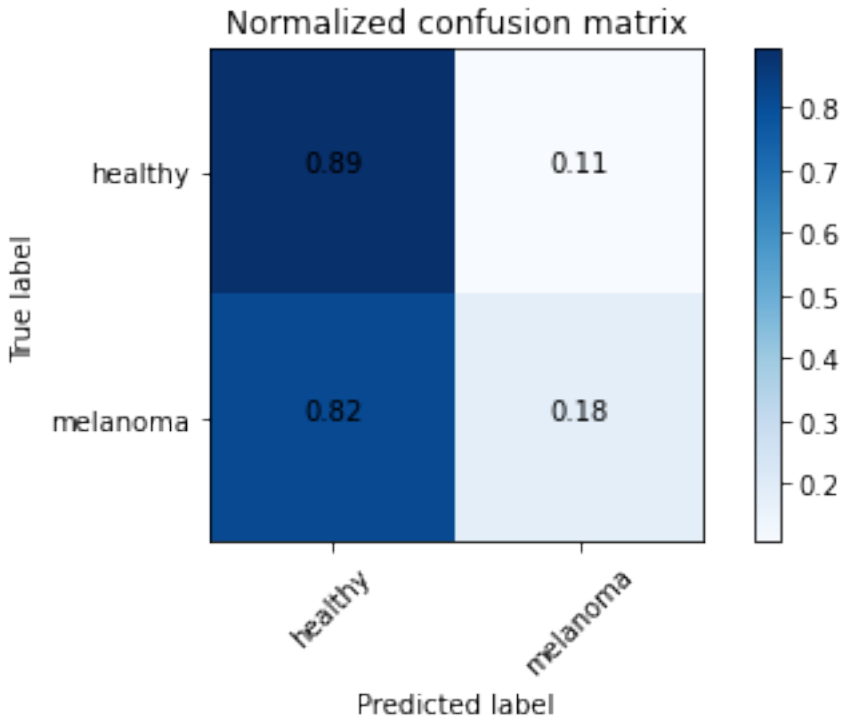
Done!

[17]:
```
# Fitting Perceptron
print("Fitting Perceptron")
Perc = Perceptron()
```

```
Perc_cv = cross_validate(Perc,X_resampled,␣
 ↪y_resampled,cv=5,scoring=('accuracy',␣
 ↪'f1'),return_train_score=True,return_estimator=True)
print(Perc_cv.keys())
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['train_accuracy'].mean(), Perc_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Perc_cv['test_accuracy'].mean(), Perc_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Perc_cv['train_f1'].
 ↪mean(), Perc_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Perc_cv['test_f1'].
 ↪mean(), Perc_cv['test_f1'].std() ))

# Look for the best estimator (the one with the greatest test accuracy)
index_best = np.argmax(Perc_cv['test_accuracy'])
estimator_best=Perc_cv['estimator'][index_best]
y_pred = estimator_best.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()


# Fitting linear SVM
print("Fitting Linear SVM")
Lsvm = LinearSVC()
Lsvm_cv = cross_validate(Lsvm,X_resampled,␣
 ↪y_resampled,cv=5,scoring=('accuracy',␣
 ↪'f1'),return_train_score=True,return_estimator=True)
print(Lsvm_cv.keys())
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['train_accuracy'].mean(), Lsvm_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['test_accuracy'].mean(), Lsvm_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Lsvm_cv['train_f1'].
 ↪mean(), Lsvm_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Lsvm_cv['test_f1'].
 ↪mean(), Lsvm_cv['test_f1'].std() ))
# Look for the best estimator (the one with the greatest test accuracy)
index_best = np.argmax(Lsvm_cv['test_accuracy'])
estimator_best=Lsvm_cv['estimator'][index_best]
y_pred = estimator_best.predict(X_test)
```

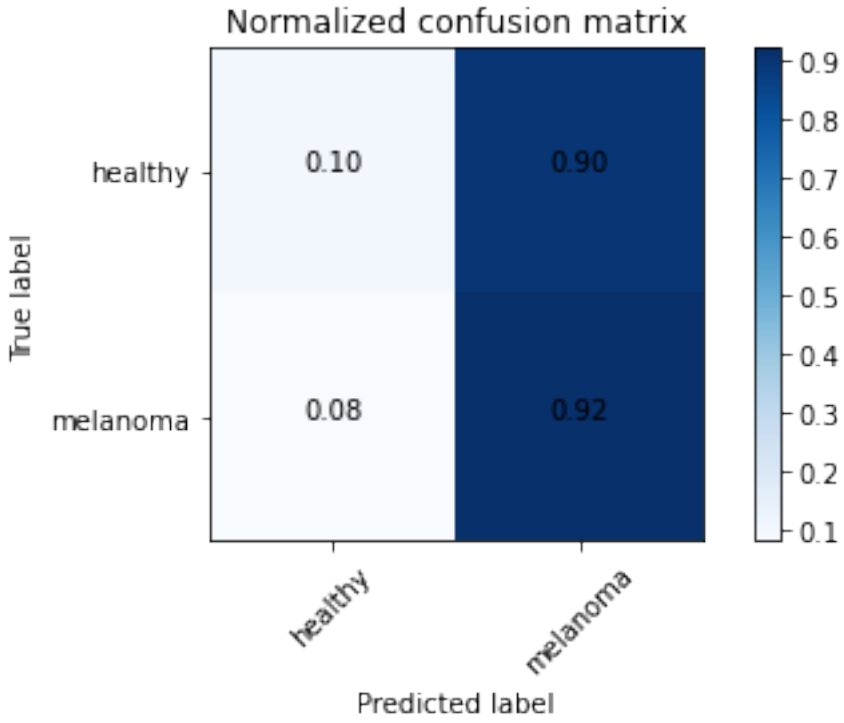
```
# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test, y_pred)

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

```
Fitting Perceptron
dict_keys(['fit_time', 'score_time', 'estimator', 'test_accuracy',
'train_accuracy', 'test_f1', 'train_f1'])
 Average and std TRAIN CV accuracy : 0.55 +- 0.041991827889095945
 Average and std TEST CV accuracy : 0.5446153846153845 +- 0.035551446179357966
 Average and std TRAIN CV f1 : 0.5102989688804164 +- 0.23396334904168817
 Average and std TEST CV f1 : 0.5010821422586128 +- 0.23659829810750618
Normalized confusion matrix
[[0.89375    0.10625   ]
 [0.81578947 0.18421053]]
```




```
Fitting Linear SVM
dict_keys(['fit_time', 'score_time', 'estimator', 'test_accuracy',
'train_accuracy', 'test_f1', 'train_f1'])
 Average and std TRAIN CV accuracy : 0.5476923076923077 +- 0.02871508685373861
```

```
 Average and std TEST CV accuracy : 0.536923076923077 +- 0.020868707640385432
 Average and std TRAIN CV f1 : 0.4193290507493434 +- 0.23336546134140515
 Average and std TEST CV f1 : 0.4059960409837363 +- 0.23793001284270193
Normalized confusion matrix
[[0.1        0.9       ]
 [0.07894737 0.92105263]]
```



Suppose that there are overlapping classes, we need to set the hyper-parameter C for the SVM model.

Question Use Cross-Validation on the oversampled data to find the best C value. Plot the confusion matrix using the best estimator (as before).

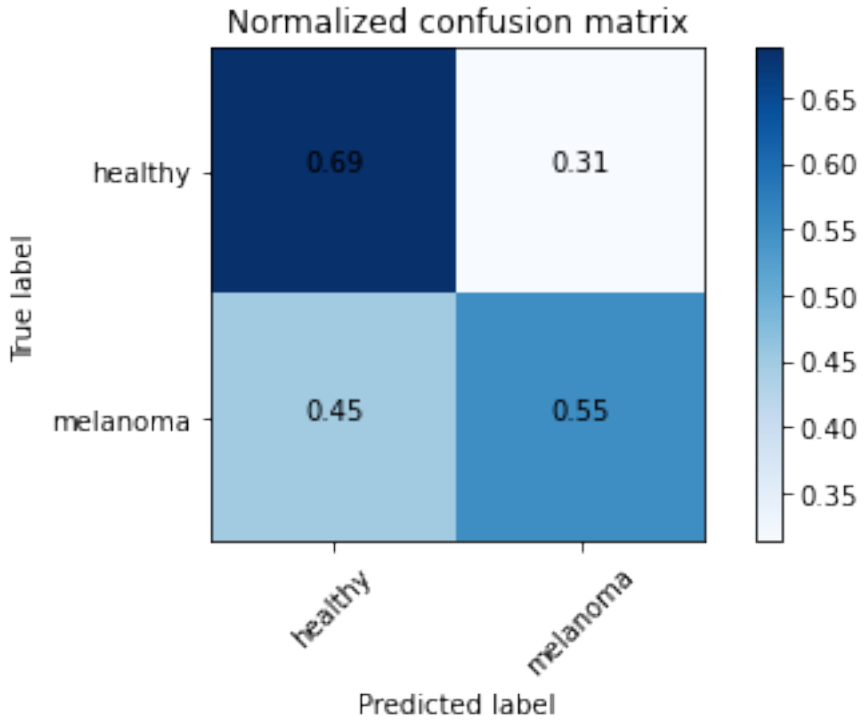
Answer:

Done!

[18]:
```
# Looking for the best hyperparameter C
p_grid_lsvm = {'C': [1e-3,1e-2,1e-1,1,2,3,4,5,6,7,8,9,1e1]}
Lsvm = LinearSVC()
grid_lsvm = GridSearchCV(estimator=Lsvm, param_grid=p_grid_lsvm,␣
 ↪scoring="accuracy", cv=5)
grid_lsvm.fit(X_resampled, y_resampled)
print("Best training Score: {}".format(grid_lsvm.best_score_))
print("Best training params: {}".format(grid_lsvm.best_params_))
```

```
y_pred = grid_lsvm.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Best training Score: 0.6292307692307693
Best training params: {'C': 0.001}
Normalized confusion matrix
[[0.6875     0.3125    ]
 [0.44736842 0.55263158]]
```



Here it is the code for non-linear SVM using radial basis function. We need to tune another hyper-parameter *gamma*. We look for the best C and *gamma* at the same time.

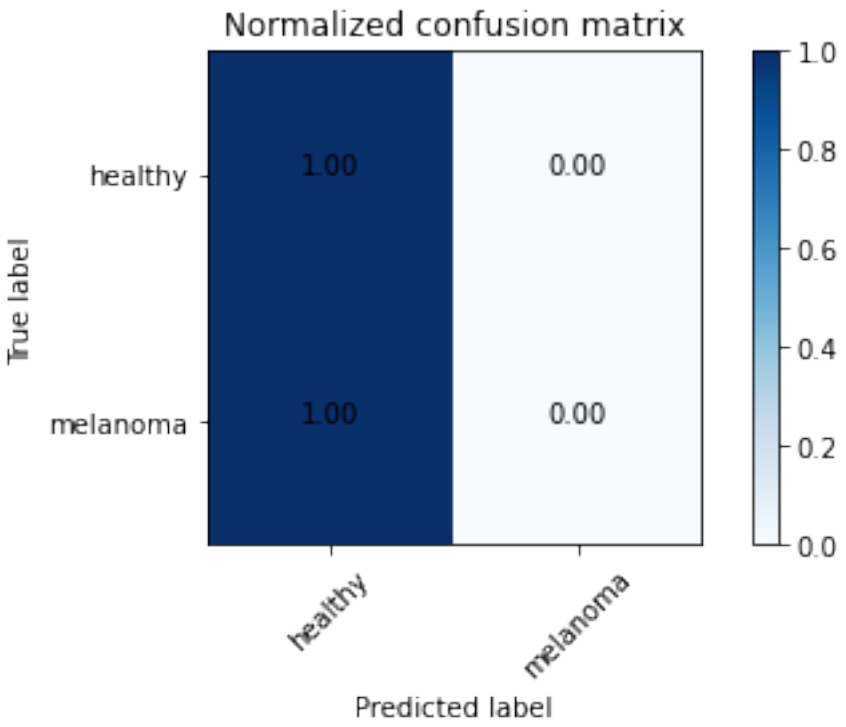
Question Use Cross-Validation on the oversampled data to find the best C and *gamma* value. Plot the confusion matrix using the best estimator (as before).

Answer:

Done!

[19]:
```
# Fitting Non-linear SVM
print("Fitting Non-linear SVM to the training set")
t0 = time()
p_grid_nlsvm = {'C': [1e-3,1e-2,1e-1,1,2,3,4,5,6,7,8,9,1e1],
              'gamma': [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.1], }
NLsvm = SVC(kernel='rbf')
grid_nlsvm = GridSearchCV(estimator=NLsvm, param_grid=p_grid_nlsvm,␣
 ↪scoring="accuracy", cv=5)
grid_nlsvm.fit(X_resampled, y_resampled)
print("Best training Score: {}".format(grid_nlsvm.best_score_))
print("Best training params: {}".format(grid_nlsvm.best_params_))
y_pred = grid_nlsvm.predict(X_test)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Fitting Non-linear SVM to the training set
Best training Score: 0.9938461538461538
Best training params: {'C': 1, 'gamma': 0.005}
Normalized confusion matrix
[[1. 0.]
 [1. 0.]]
```



Question Use the non-linear SVM with the two strategies seen before (different scoring function and/or oversampled data). Do the results change ? Why in your opinion ?

Answer:

It does change with resampling our accuracy is very high (99.38%), but without its 80%

Question Try to draw a conclusion from the different experiments. Which is the best method ? Which scoring function should you use ? Is it worth it to oversample one of the two classes ?

Answer:

It is important to resample and balance the dataset. The scoring function is important as well since each one of them measures differently the distance between the model and the data I would use accuracy scoring.

OPTIONAL Another interesting question is: what about the number of features ? Can we reduce the dimensionality ? You could use one of the techniques seen during the previous lectures (i.e. PCA) ...

Answer:

Done!

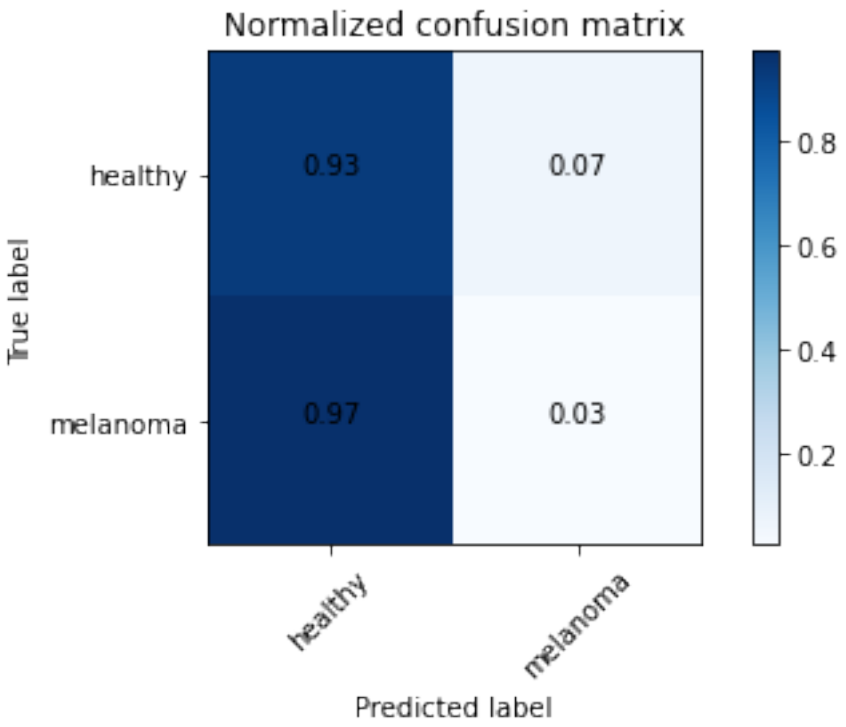
[20]:
```
# Test PCA with a linear SVM
pca = PCA(n_components=0.90) # use number of components to explain 90% of␣
↪variability
```

```
pca.fit(X_train)
X_train_pca = pca.transform(X_train)
X_test_pca = pca.transform(X_test)

print('Using PCA number of features went from ', X_train.shape[1], ' to ', ␣
 ↪X_train_pca.shape[1] )
```

```
Using PCA number of features went from  30  to  5
```

[21]:
```
# Fitting Linear SVM
print("Fitting Linear SVM")
Lsvm = LinearSVC()
Lsvm_cv = cross_validate(Lsvm,X_train_pca, y_train,cv=5,scoring=('accuracy',␣
 ↪'f1'),return_train_score=True,return_estimator=True)
print(Lsvm_cv.keys())
print(" Average and std TRAIN CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['train_accuracy'].mean(), Lsvm_cv['train_accuracy'].std() ))
print(" Average and std TEST CV accuracy : {0} +- {1}".
 ↪format(Lsvm_cv['test_accuracy'].mean(), Lsvm_cv['test_accuracy'].std() ))
print(" Average and std TRAIN CV f1 : {0} +- {1}".format(Lsvm_cv['train_f1'].
 ↪mean(), Lsvm_cv['train_f1'].std() ))
print(" Average and std TEST CV f1 : {0} +- {1}".format(Lsvm_cv['test_f1'].
 ↪mean(), Lsvm_cv['test_f1'].std() ))
# Look for the best estimator (the one with the greatest test accuracy)
index_best = np.argmax(Lsvm_cv['test_accuracy'])
estimator_best=Lsvm_cv['estimator'][index_best]
y_pred = estimator_best.predict(X_test_pca)
# 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,
                      title='Normalized confusion matrix')
plt.show()
```

```
Fitting Linear SVM
dict_keys(['fit_time', 'score_time', 'estimator', 'test_accuracy',
'train_accuracy', 'test_f1', 'train_f1'])
 Average and std TRAIN CV accuracy : 0.7096050773011358 +- 0.055386992163591935
 Average and std TEST CV accuracy : 0.6864814814814815 +- 0.09504943202498964
 Average and std TRAIN CV f1 : 0.20717670521363857 +- 0.07571002869405347
 Average and std TEST CV f1 : 0.1853098946524797 +- 0.09893497372996904
Normalized confusion matrix
[[0.93125    0.06875   ]
 [0.97368421 0.02631579]]
```



OPTIONAL … or test the importance of the single features. The more naive technique would be to test each feature independently in a greedy fashion called sequential forward feature selection. Starting from an empty set and a classification model, you will first add the feature that maximizes a certain criterion (i.e. f1 score). Then, you will iterate this process until a chosen stopping criterion by adding at each iteration only the best feature. Each feature can be added of course only once. You could also use the opposite process by removing at each iteraton the least important feature starting from the entire set of features (i.e. sequential backward feature selection). Implement at least one of these ideas.

[22]:
```
# Implement forward feature selection and/or backward feature selection
# with a linear SVM

#XXXXXXXXXX
```

[ ]:
```

```