In [1]:
```python
import keras
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Dropout
from keras.optimizers import Adam
from keras.callbacks import TensorBoard

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import accuracy_score
from keras.utils import np_utils
import itertools
```

Step 2

Load Dataset : After loading the Dataset you have to normalize every image.

Note: an image is a Uint8 matrix of pixels and for calculation, you need to convert the format of the image to float or double

In [2]:
```python
#load dataset
data = np.load('../input/orl-faces/ORL_faces.npz')

# load the "Train Images"
x_train = data['trainX']
#normalize every image
x_train = np.array(x_train,dtype='float32')/255

x_test = data['testX']
x_test = np.array(x_test,dtype='float32')/255

# load the Label of Images
y_train= data['trainY']
y_test= data['testY']

# show the train and test Data format
print('x_train : {}'.format(x_train[:]))
print('Y-train shape: {}'.format(y_train))
print('x_test shape: {}'.format(x_test.shape))
```

```
x_train : [[0.1882353  0.19215687 0.1764706  ... 0.18431373 0.18039216 0.18039216]
 [0.23529412 0.23529412 0.24313726 ... 0.1254902  0.13333334 0.13333334]
 [0.15294118 0.17254902 0.20784314 ... 0.11372549 0.10196079 0.11372549]
 ...
 [0.44705883 0.45882353 0.44705883 ... 0.38431373 0.3764706  0.38431373]
 [0.4117647  0.4117647  0.41960785 ... 0.21176471 0.18431373 0.16078432]
 [0.45490196 0.44705883 0.45882353 ... 0.37254903 0.39215687 0.39607844]]
Y-train shape: [ 0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  1
1
  2  2  2  2  2  2  2  2  2  2  2  2  3  3  3  3  3  3  3  3  3  3  3  3
  4  4  4  4  4  4  4  4  4  4  4  4  5  5  5  5  5  5  5  5  5  5  5  5
  6  6  6  6  6  6  6  6  6  6  6  6  7  7  7  7  7  7  7  7  7  7  7  7
  8  8  8  8  8  8  8  8  8  8  8  8  9  9  9  9  9  9  9  9  9  9  9  9
 10 10 10 10 10 10 10 10 10 10 10 10 11 11 11 11 11 11 11 11 11 11 11 11
 12 12 12 12 12 12 12 12 12 12 12 12 13 13 13 13 13 13 13 13 13 13 13 13
 14 14 14 14 14 14 14 14 14 14 14 14 15 15 15 15 15 15 15 15 15 15 15 15
```

```
 16 16 16 16 16 16 16 16 16 16 16 16 17 17 17 17 17 17 17 17 17 17 17 17
 18 18 18 18 18 18 18 18 18 18 18 18 19 19 19 19 19 19 19 19 19 19 19 19]
x_test shape: (160, 10304)
```

Step 3

Split DataSet : Validation data and Train

Validation DataSet: this data set is used to minimize overfitting.If the accuracy over the training data set increases, but the accuracy over then validation data set stays the same or decreases, then you're overfitting your neural network and you should stop training.

Note: we usually use 30 percent of every dataset as the validation data but Here we only used 5 percent because the number of images in this dataset is very low.

In [3]:
```
x_train, x_valid, y_train, y_valid= train_test_split(
    x_train, y_train, test_size=.05, random_state=1234,)
```

Step 4

for using the CNN, we need to change The size of images (The size of images must be the same)

In [4]:
```
im_rows=112
im_cols=92
batch_size=512
im_shape=(im_rows, im_cols, 1)

#change the size of images
x_train = x_train.reshape(x_train.shape[0], *im_shape)
x_test = x_test.reshape(x_test.shape[0], *im_shape)
x_valid = x_valid.reshape(x_valid.shape[0], *im_shape)

print('x_train shape: {}'.format(y_train.shape[0]))
print('x_test shape: {}'.format(y_test.shape))
```

```
x_train shape: 228
x_test shape: (160,)
```

Step 5

Build CNN model: CNN have 3 main layer:

1-Convolotional layer 2- pooling layer 3- fully connected layer

we could build a new architecture of CNN by changing the number and position of layers.

In [5]:
```
#filters= the depth of output image or kernels

cnn_model= Sequential([
    Conv2D(filters=36, kernel_size=7, activation='relu', input_shape= im_shape),
    MaxPooling2D(pool_size=2),
    Conv2D(filters=54, kernel_size=5, activation='relu', input_shape= im_shape),
    MaxPooling2D(pool_size=2),
    Flatten(),
```

```
    Dense(2024, activation='relu'),
     Dropout(0.5),
    Dense(1024, activation='relu'),
    Dropout(0.5),
    Dense(512, activation='relu'),
    Dropout(0.5),
    #20 is the number of outputs
    Dense(20, activation='softmax')
])

cnn_model.compile(
    loss='sparse_categorical_crossentropy',#'categorical_crossentropy',
    optimizer=Adam(lr=0.0001),
    metrics=['accuracy']
)
```

Show the model's parameters.

In [6]:
```
cnn_model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d (Conv2D)              (None, 106, 86, 36)       1800      
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 53, 43, 36)        0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 49, 39, 54)        48654     
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 19, 54)        0         
_________________________________________________________________
flatten (Flatten)            (None, 24624)             0         
_________________________________________________________________
dense (Dense)                (None, 2024)              49841000  
_________________________________________________________________
dropout (Dropout)            (None, 2024)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              2073600   
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0         
_________________________________________________________________
dense_2 (Dense)              (None, 512)               524800    
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 20)                10260     
=================================================================
Total params: 52,500,114
Trainable params: 52,500,114
Non-trainable params: 0
_________________________________________________________________
```

Step 6 Train the Model

Note: You can change the number of epochs

In [7]:
```
history=cnn_model.fit(
    np.array(x_train), np.array(y_train), batch_size=512,
    epochs=250, verbose=2,
    validation_data=(np.array(x_valid),np.array(y_valid)),
)
```

```
Epoch 1/250
1/1 - 5s - loss: 3.0126 - accuracy: 0.0526 - val_loss: 2.9990 - val_accuracy: 0.0000
e+00
Epoch 2/250
```

```
Epoch 201/250
1/1 - 3s - loss: 0.0251 - accuracy: 1.0000 - val_loss: 9.6885e-04 - val_accuracy: 1.
0000
Epoch 202/250
1/1 - 3s - loss: 0.0223 - accuracy: 0.9956 - val_loss: 6.8044e-04 - val_accuracy: 1.
0000
Epoch 203/250
1/1 - 3s - loss: 0.0102 - accuracy: 1.0000 - val_loss: 5.1768e-04 - val_accuracy: 1.
0000
Epoch 204/250
1/1 - 4s - loss: 0.0172 - accuracy: 0.9956 - val_loss: 4.5720e-04 - val_accuracy: 1.
0000
Epoch 205/250
1/1 - 3s - loss: 0.0191 - accuracy: 0.9956 - val_loss: 4.7776e-04 - val_accuracy: 1.
0000
Epoch 206/250
1/1 - 3s - loss: 0.0227 - accuracy: 1.0000 - val_loss: 5.8830e-04 - val_accuracy: 1.
0000
Epoch 207/250
1/1 - 3s - loss: 0.0169 - accuracy: 0.9956 - val_loss: 7.7509e-04 - val_accuracy: 1.
0000
Epoch 208/250
1/1 - 3s - loss: 0.0168 - accuracy: 0.9956 - val_loss: 9.2391e-04 - val_accuracy: 1.
0000
Epoch 209/250
1/1 - 3s - loss: 0.0155 - accuracy: 1.0000 - val_loss: 0.0011 - val_accuracy: 1.0000
Epoch 210/250
1/1 - 3s - loss: 0.0158 - accuracy: 0.9956 - val_loss: 9.4481e-04 - val_accuracy: 1.
0000
Epoch 211/250
1/1 - 3s - loss: 0.0286 - accuracy: 0.9912 - val_loss: 6.0312e-04 - val_accuracy: 1.
0000
Epoch 212/250
1/1 - 3s - loss: 0.0194 - accuracy: 0.9956 - val_loss: 4.4468e-04 - val_accuracy: 1.
0000
Epoch 213/250
1/1 - 3s - loss: 0.0318 - accuracy: 0.9868 - val_loss: 3.9425e-04 - val_accuracy: 1.
0000
Epoch 214/250
1/1 - 4s - loss: 0.0128 - accuracy: 0.9956 - val_loss: 3.9505e-04 - val_accuracy: 1.
0000
Epoch 215/250
1/1 - 3s - loss: 0.0121 - accuracy: 1.0000 - val_loss: 3.7706e-04 - val_accuracy: 1.
0000
Epoch 216/250
1/1 - 3s - loss: 0.0193 - accuracy: 1.0000 - val_loss: 4.1789e-04 - val_accuracy: 1.
0000
Epoch 217/250
1/1 - 3s - loss: 0.0186 - accuracy: 1.0000 - val_loss: 5.4611e-04 - val_accuracy: 1.
0000
Epoch 218/250
1/1 - 3s - loss: 0.0119 - accuracy: 1.0000 - val_loss: 7.8346e-04 - val_accuracy: 1.
0000
Epoch 219/250
1/1 - 3s - loss: 0.0125 - accuracy: 1.0000 - val_loss: 0.0011 - val_accuracy: 1.0000
Epoch 220/250
1/1 - 3s - loss: 0.0101 - accuracy: 1.0000 - val_loss: 0.0013 - val_accuracy: 1.0000
Epoch 221/250
1/1 - 3s - loss: 0.0113 - accuracy: 1.0000 - val_loss: 0.0014 - val_accuracy: 1.0000
Epoch 222/250
1/1 - 3s - loss: 0.0119 - accuracy: 1.0000 - val_loss: 0.0013 - val_accuracy: 1.0000
Epoch 223/250
1/1 - 4s - loss: 0.0139 - accuracy: 1.0000 - val_loss: 9.3666e-04 - val_accuracy: 1.
0000
Epoch 224/250
1/1 - 3s - loss: 0.0278 - accuracy: 0.9912 - val_loss: 0.0011 - val_accuracy: 1.0000
Epoch 225/250
1/1 - 3s - loss: 0.0178 - accuracy: 0.9956 - val_loss: 8.6807e-04 - val_accuracy: 1.
0000
```

```
0000
Epoch 250/250
1/1 - 3s - loss: 0.0050 - accuracy: 1.0000 - val_loss: 2.1116e-04 - val_accuracy: 1.
0000
```

Evaluate the test data

In [8]:
```
scor = cnn_model.evaluate( np.array(x_test),  np.array(y_test), verbose=0)

print('test los {:.4f}'.format(scor[0]))
print('test acc {:.4f}'.format(scor[1]))
```

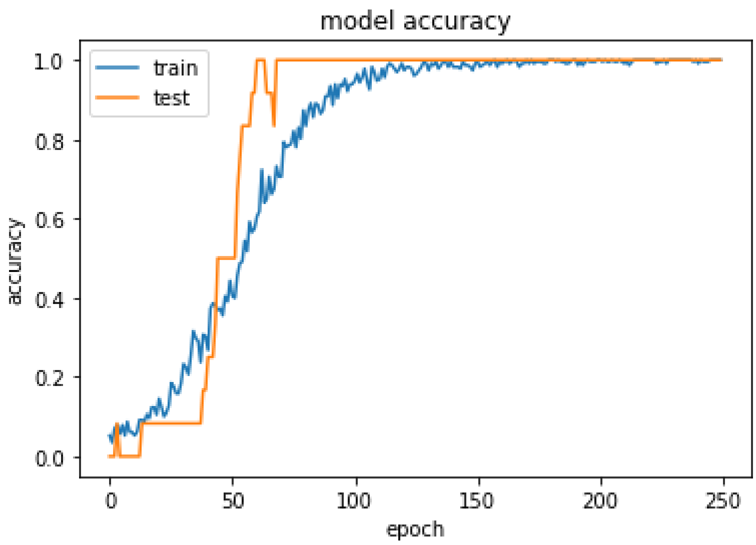
```
test los 0.3272
test acc 0.9375
```

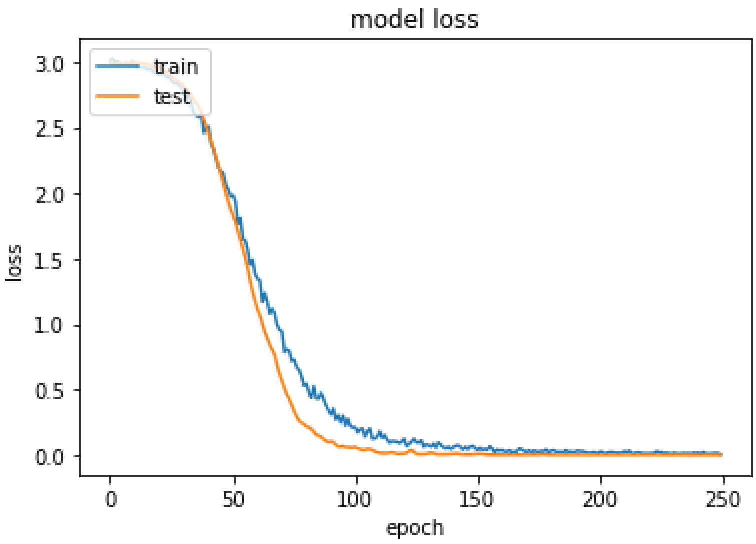
Step 7

plot the result

In [9]:
```
# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```





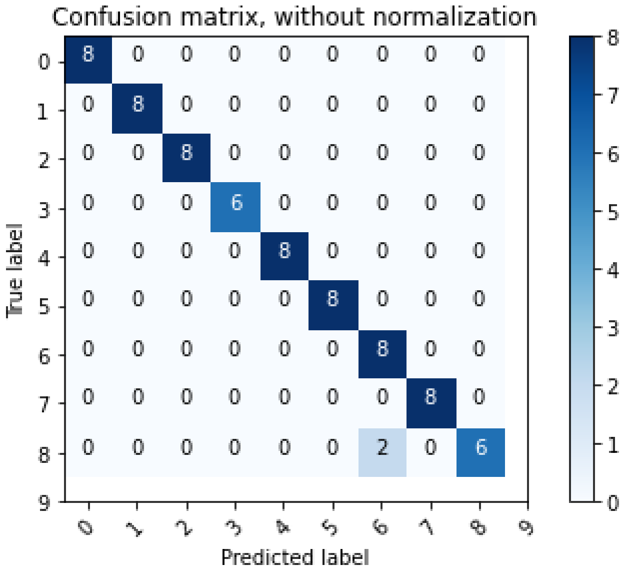
step 8

Plot Confusion Matrix

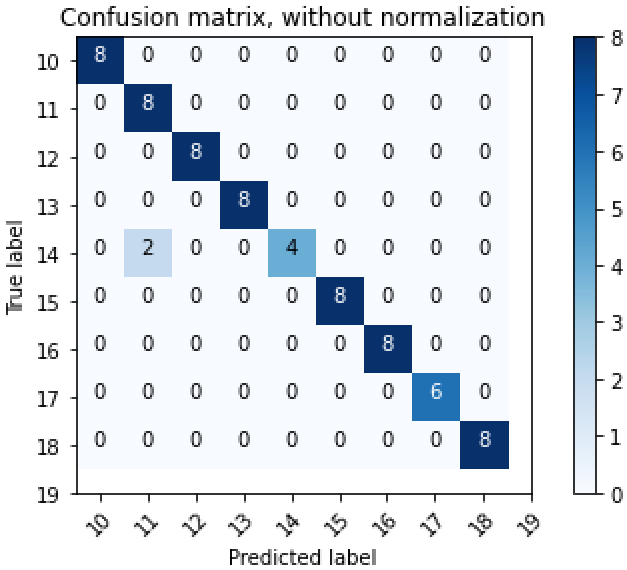
In [10]:
```
predicted =np.argmax(cnn_model.predict(x_test), axis=-1)

#print(predicted)
#print(y_test)
ynew = cnn_model.predict_classes(x_test)


Acc=accuracy_score(y_test, ynew)
print("accuracy : ")
print(Acc)
#/tn, fp, fn, tp = confusion_matrix(np.array(y_test), ynew).ravel()
cnf_matrix=confusion_matrix(np.array(y_test), ynew)

y_test1 = np_utils.to_categorical(y_test, 20)



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="white" if cm[i, j] > thresh else "black")

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


print('Confusion matrix, without normalization')
print(cnf_matrix)

plt.figure()
plot_confusion_matrix(cnf_matrix[1:10,1:10], classes=[0,1,2,3,4,5,6,7,8,9],
                      title='Confusion matrix, without normalization')

plt.figure()
plot_confusion_matrix(cnf_matrix[11:20,11:20], classes=[10,11,12,13,14,15,16,17,18,1
                      title='Confusion matrix, without normalization')

print("Confusion matrix:\n%s" % confusion_matrix(np.array(y_test), ynew))
print(classification_report(np.array(y_test), ynew))
```

```
/opt/conda/lib/python3.7/site-packages/tensorflow/python/keras/engine/sequential.py:
450: UserWarning: `model.predict_classes()` is deprecated and will be removed after
2021-01-01. Please use instead:* `np.argmax(model.predict(x), axis=-1)`,   if your m
odel does multi-class classification   (e.g. if it uses a `softmax` last-layer activ
ation).* `(model.predict(x) > 0.5).astype("int32")`,   if your model does binary cla
ssification   (e.g. if it uses a `sigmoid` last-layer activation).
  warnings.warn('`model.predict_classes()` is deprecated and '
accuracy :
0.9375
Confusion matrix, without normalization
[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]
 [0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 2 0 6 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 4 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0]
 [0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 6 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]]
Confusion matrix, without normalization
```



Confusion matrix, without normalization




```
Confusion matrix:
[[8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0]
 [0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 2 0 6 0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0]
 [2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 4 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0]
 [0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 6 0]
 [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]]
              precision    recall  f1-score   support

           0       0.80      1.00      0.89         8
           1       1.00      1.00      1.00         8
           2       1.00      1.00      1.00         8
           3       1.00      1.00      1.00         8
           4       1.00      0.75      0.86         8
           5       1.00      1.00      1.00         8
```

6	1.00	1.00	1.00	8
7	0.67	1.00	0.80	8
8	1.00	1.00	1.00	8
9	1.00	0.75	0.86	8
10	1.00	1.00	1.00	8
11	1.00	1.00	1.00	8
12	0.80	1.00	0.89	8
13	1.00	1.00	1.00	8
14	1.00	1.00	1.00	8
15	1.00	0.50	0.67	8
16	1.00	1.00	1.00	8
17	0.80	1.00	0.89	8
18	1.00	0.75	0.86	8
19	1.00	1.00	1.00	8
accuracy			0.94	160
macro avg	0.95	0.94	0.94	160
weighted avg	0.95	0.94	0.94	160