Identification and Separation of Musical Instruments

by

19BEC0839 Aryan Malik
19BEC0806 Abhishek P Iyer
19BEC0836 Akshad Patel
19BEC0838Anmol Narain

Under the guidance of

Prof. Kalaivani S

SENSE

VIT, Vellore

OBJECTIVE

In this project, we utilize machine learning techniques to construct a classifier for automatic instrument recognition given a mono-track audio recording. We focus on the classification of six instruments commonly used in music. By examining the spectral content of each instrument, we propose a set of features that can be used to accurately classify musical instruments and we present results from both supervised and unsupervised learning algorithms applied to our generated data set.

INTRODUCTION

The primary goal of this project is to classify and separate single-instrument recordings for Cello, Flute, Oboe, Sax, Trumpet and Viola by the means of MFCC and K-NN.
The input to our model consists of individual audio recordings for each instrument in MP3 format. We perform feature extraction from the discrete Fourier transform (DFT) of the normalized signal and label each example with a unique integer corresponding to the instruments' class. We then use this data for analysis and train multiple SVMs to make predictions on the test set.

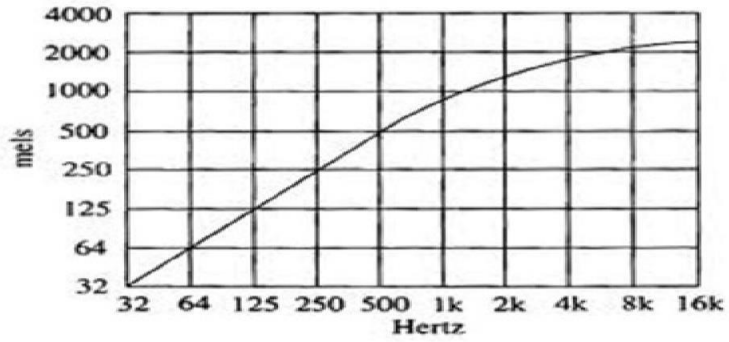
FEATURES

1. Mel Frequency Cepstral Coefficients (MFCC)

The Mel frequency cepstral coefficients (MFCCs) of a signal are a small set of features whichconcisely describe the overall shape of a spectral envelope. Each coefficient has a value for each frame of the sound.

The formula to convert frequency f hertz into Mel mf is given by:

$$m_f = 2595 log_{10} (\frac{f}{700} + 1)$$



Thus, with the help of Filter bank with proper spacing done by Mel scaling it becomes easy to get the estimation about the energies at each spot and once these energies are estimated then the log of these energies also known as Mel spectrum can be used for calculating first 13 coefficients using DCT. Since, the increasing numbers of coefficients represent faster change in the estimated energies and thus have less information to be used for classifying the given images. Hence, first 13 coefficients are calculated using DCT and higher are discarded.

How to obtain MFCC?

- Take the Fourier transform of (a windowed excerpt of) a signal.
- Map the powers of the spectrum obtained above onto the mel scale, using triangular overlapping windows.
- Take the logs of the powers at each of the mel frequencies.
- Take the discrete cosine transform (DCT) of the list of mel log powers, as if it were a signal.
- The MFCCs are the amplitudes of the resulting spectrum.

2. Fast Fourier Transform (FFT)

Features for each sample are obtained by using the DFT and identifying the 10 most predominant frequencies with the greatest percentage contribution to the total power of the signal. Given that our samples are not uniform in length, we first normalize each sample to have unit energy. This also removes any effects due to variance in volume levels among the samples.

After normalizing, we find the 10 frequencies with greatest amplitude and integrate over a log range centered at each frequency to compute the power contribution. Using a log range accounts for a larger octave band for notes at higher frequencies and models our assumption that the guitars are tuned using twelve-tone equal temperament.

Thus, for each of the 10 selected frequencies, we extract a 3-tuple of frequency, power, and amplitude such that the nth tuple represents the frequency component with the nth largest power contribution. An example of feature extraction is shown in Figure 1 for the same note played on the bass guitar and acoustic guitar. We observe that each sample gives a unique signature of each instrument in the frequency domain.

FFT is used for doing conversion from the spatial domain to the frequency domain. Each

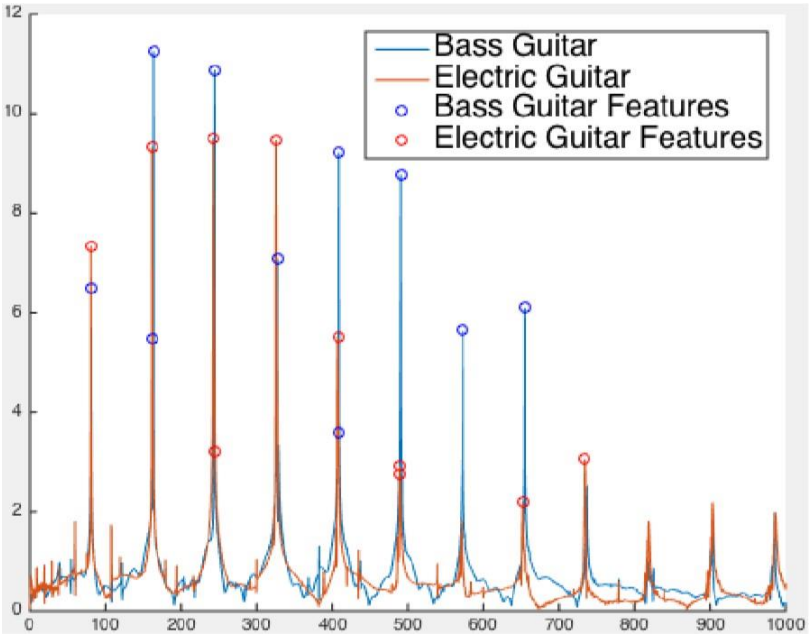
frame having Nm samples are converted into frequency domain. Fourier transformation is a fast algorithm to apply Discrete Fourier Transform (DFT), on the given set of Nm samples shown below:

$$D_k = \sum_{m=0}^{N_m-1} D_m \, e^{\frac{-j2\pi km}{Nm}}$$

Where k= 0, 1, 2 ….. N_m-1

Basically, the definition for FFT and DFT is same, which means that the output for the transformation will be the same; however, they differ in their computational complexity. In case of DFT, each frame with N-M samples directly will be used as a sequence for Fourier transformation. On another, in case of FFT this frame will be divided into small DFT's and then computation will be done on this divided small DFT's as individual sequence thus the computation will be faster and easier. Thus, it is in digital processing or other area instead of directly using DFT, FFT is used for applying DFT.

Commonly, Dk are the combination of real and imaginary numbers thus it represents the complex numbers but, merely absolute values (frequency magnitudes) are considered to carry out further process. The obtained sequence can be interpreted as positive frequencies $0 \leq f <$ Fs /2 correspond to values $0 \leq m \leq Nm/2 - 1$, while negative frequencies $-Fs/2 < f < 0$ correspond to values $Nm/2 + 1 \leq m \leq Nm - 1$, Fs is the sampling frequency. By calculating DFT we can obtain the magnitude spectrum.



Alternatively, we considered using the entire FFT as the feature vector for each sample. However, in order to avoid high variance, we chose to select a small number of discrete features relative to the size of our training set. This stemmed primarily from the difficulty of acquiring a large number of training examples. Furthermore, by selecting a small set of features shown to

be most relevant by our preliminary analysis, we were able to obtain reduced computational complexity and achieve run times of under a minute.

Normalization

In order to train a network on the co-relation between the amplitude tops of a sample and not the actual values, the data was normalized according to the feature scaling method, scaling the range of amplitudes between [0,1] (see eq.).

The normalization resulted in an emphasized co-relation of a sample's amplitudes across the frequency spectrum, allowing each sample to equally contribute to the training of the model

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

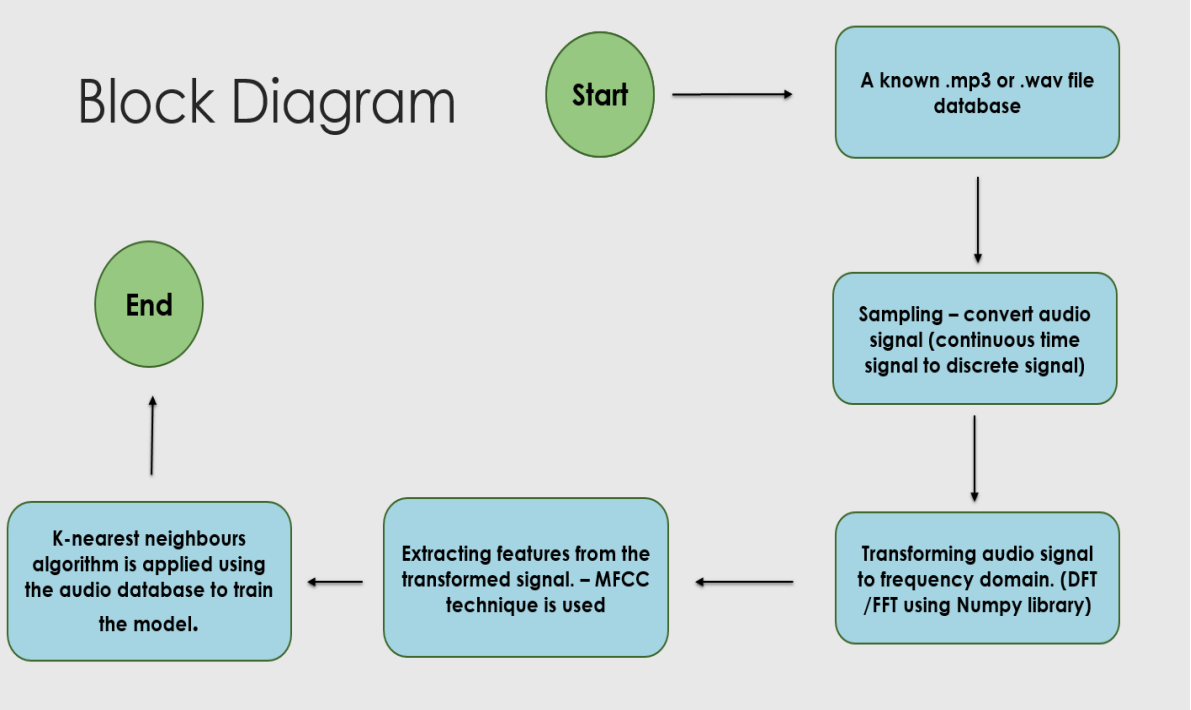
Machine Learning Technique (k-NN)

KNN is non-parametric learning algorithm. Non parametric means it does not make any assumption on the underlying data distribution. Its outstanding characteristic is that it does not require a training stage in the strict sense. The training samples are rather used directly by the classifier during the classification stage. The key idea behind this classifier is that, if we are given a test pattern (unknown feature vector), x, we first detect its k-nearest neighbors in the training set and count how many of those belong to each class. In the end, the feature vector is assigned to the class which has accumulated the highest number of neighbors.

Therefore, for the k-NN algorithm to operate, the following ingredients are required:

1. A dataset of labeled samples, i.e. a training set of feature vectors and respective class labels
2. An integer $k \geq 1$.
3. A distance (dissimilarity) measure.

FLOWCHART



DATASET

IRMAS: a dataset for instrument recognition in musical audio signals

600 audio files (mp3 files)

Link - https://philharmonia.co.uk/resources/sound-samples/

(PHILHARMONIA DATASET)

PYTHON CODE AND RESULT

(Code is written in Jupyter notebook, so the outputs are obtained after every block of code)

In [72]:
```
import warnings
warnings.filterwarnings('ignore')

#import data lib
import numpy as np
import pickle
import itertools
```

In [73]:
```
#import sys lib
import os, fnmatch
```

In [74]:
```
#import visual lib
import seaborn
import matplotlib.pyplot as plt
from IPython.core.display import HTML, display
```

In [75]:
```
#import ML lib
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import recall_score, precision_score, accuracy_score
from sklearn.metrics import confusion_matrix, f1_score, classification_report
from sklearn.svm import LinearSVC, SVC
from sklearn.externals import joblib
```

In [84]:
```
#random seed
from numpy.random import seed
seed(1)
```

In [94]:
```
#import audio lib
import librosa.display, librosa
```

In [99]:
```
#configurations
path = './audio'
```

In [100]:
```
#Get files

files = []
for root, dirnames, filenames in os.walk(path):
    for filename in fnmatch.filter(filenames, '*.mp3'):
        files.append(os.path.join(root, filename))

print("found {} audio files in {}".format(len(files), path))
```

```
found 4555 audio files in ./audio
```

In [102]:
```
#Get lables to identify number of different samples by listing
labels=[]
classes=['flute', 'sax', 'oboe', 'cello', 'trumpet', 'viola']
colour_dict={'cello':'blue', 'flute':'red', 'oboe':'green', 'trumpet':'black', 'sax':'magenta', 'viola':'yellow'}
colour_list=[]
for filename in files:
    for name in classes:
        if fnmatch.fnmatchcase(filename, '*'+name+'*'):
            labels.append(name)
            colour_list.append(colour_dict[name])
            break
        else:
            labels.append('other')
```

In [103]:
```
# Encode labels
labelencoder=LabelEncoder()
labelencoder.fit(labels)
print(len(labelencoder.classes_), "classes:", ", ".join(list(labelencoder.classes_)))
classes_num=labelencoder.transform(labels)
```

```
7 classes: cello, flute, oboe, other, sax, trumpet, viola
```

In [104]:
```
# Parameter for MFCC
fs=44100 #sampling frequency
n_fft=2048 #length of FFT window
hop_length=512 #Number of samples between successive frames
n_mels=128 #number of mel bands
n_mfcc=13 #number of MFCCs

#Machine learning parameters
testset_size=0.25 #Percentage of data for testing
n_neighbours=1 #number of neighbours for knn classifier(check this once)
```

In [105]:
```
#Defining funtion for calculating audio features
def get_features(y, sr=fs):
    mfcc=librosa.feature.mfcc(S=librosa.power_to_db(S), n_mfcc=n_mfcc)
    feature_vector=np.mean(mfcc, 1)
    #feature_vector=(feature_vector-np.mean(feature_vector))/np.std(feature_vector)
    return feature_vector
```

In [106]:
```
# LOADING THEM AND CREATING FEATURES VECTORS (FOR CONFUSION MATRIX)

feature_vectors = []
sound_paths = []
for i,f in enumerate(files):
    print("get %d of %d = %s"%(i+1, len(files), f))
    try:
        y, sr = librosa.load(f, sr = fs)
        y/=y.max() #Normalize
        if len(y) < 2:
            print("error loading %s" %f)
            continue
        feat = get_features(y, sr)
        feature_vectors.append(feat)
        sound_paths.append(f)
    except Exception as e:
        print("erroe loading %s. Error: %s" % (f,e))


print("Calculated %d feature vectors" %len(feature_vectors))
```

In [107]:
```
# NORMALIZATION OF DATASET

scaler = StandardScaler()
scaled_feature_vectors = scaler.fit_transform(np.array(feature_vectors))
print("Feature vectors shape:", scaled_feature_vectors.shape)
```

```
Feature vectors shape: (4553, 13)
```

In [108]:
```
#LOADING MFCC FEATURE

filename = "mfcc_feature_vectors.p1"
#scaled_feature_vectors = pickle.load( open(filename, "rb"))

# Save feature_vectors for future use
with open(filename, "wb") as f:
    pickle.dump( scaled_feature_vectors, f)
```

In [110]:
```
splitter = StratifiedShuffleSplit(n_splits=1, test_size=testset_size, random_state=0)
splits = splitter.split(scaled_feature_vectors, classes_num)
for train_index, test_index in splits:
    train_set = scaled_featurevectors[train_index]
    test_set = scaled_feature_vectors[test_index]
    train_classes = classes_num[train_index]
    test_classes = classes_num[test_index]
```

In [46]:
```
print("train_set shape: ", train_set.shape)
print("test_set shape: ", test_set.shape)
print("train_classes shape: ", train_classes.shape)
print("test_classes shape: ", test_classes.shape)
```

```
train_set shape:  (3414, 13)
test_set shape:  (1139, 13)
train_classes shape:  (3414,)
test_classes shape:  (1139,)
```

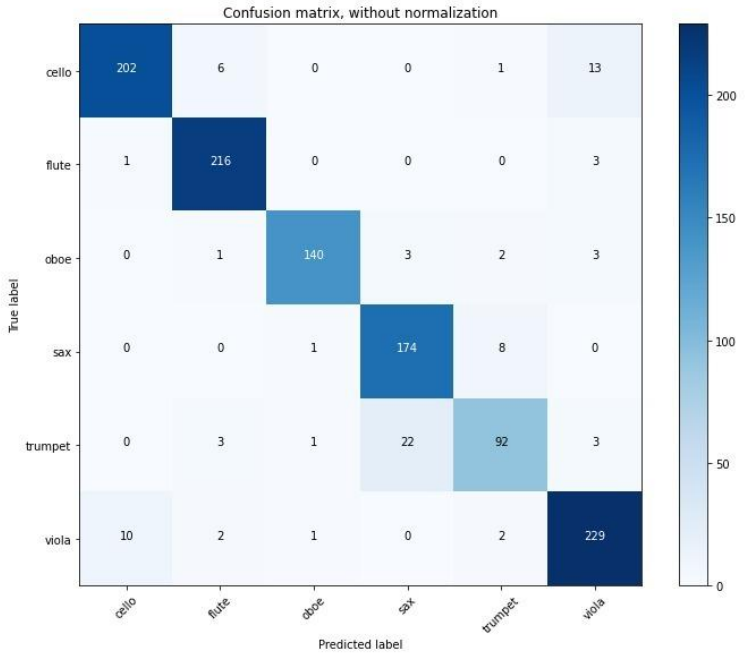
In [98]:
```
# with n = 1

n_neighbors = 1
model_knn = KNeighborsClassifier(n_neighbors = n_neighbors)
#KNN
model_knn.fit(train_set, train_classes);
#Predict using the test set
predicted_labels = model_knn.predict(test_set)

#model_knn.kneighbors_graph(train_set, n_neighbors, mode='connectivity')
```

In [99]:
```
#12) EVALUATION

#Recall the ability of the classifier to find all the positive samples
print("Recall: ", recall_score(test_classes, predicted_labels, average= None))

#Precision - The precision is intuitively the ability of the classifier not to label as positive a sample that is negative
print("Precision: ", precision_score(test_classes, predicted_labels, average=None))

#F1-Score- The F1 score can be interpreted as a weighted average of the precision and recall
print("F1-Score: ",f1_score(test_classes,predicted_labels,average=None))

#Accuracy - the number of correctly classified samples
print("Accuracy: %.2f ,"%accuracy_score(test_classes, predicted_labels,normalize = True), accuracy_score(test_classes, predicted
print("Number of samples: ", test_classes.shape[0])
```

```
Recall:  [0.91 0.98 0.94 0.95 0.76 0.94]
Precision:  [0.95 0.95 0.98 0.87 0.88 0.91]
F1-Score:  [0.93 0.96 0.96 0.91 0.81 0.93]
Accuracy: 0.92 , 1053
Number of samples:  1139
```

In [100]:
```
#13)VISUALIZATION USING CONFUSION MATRIZ

#Compute confusion matraix
cnf_matrix = confusion_matrix(test_classes, predicted_labels)
np.set_printoptions(precision=2)

#DEFINING FUNCTION FOR VISUALISING
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("Normalize confusion matrix")
    else:
    print('Confusion matrix, without normalizaton')
    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')
```

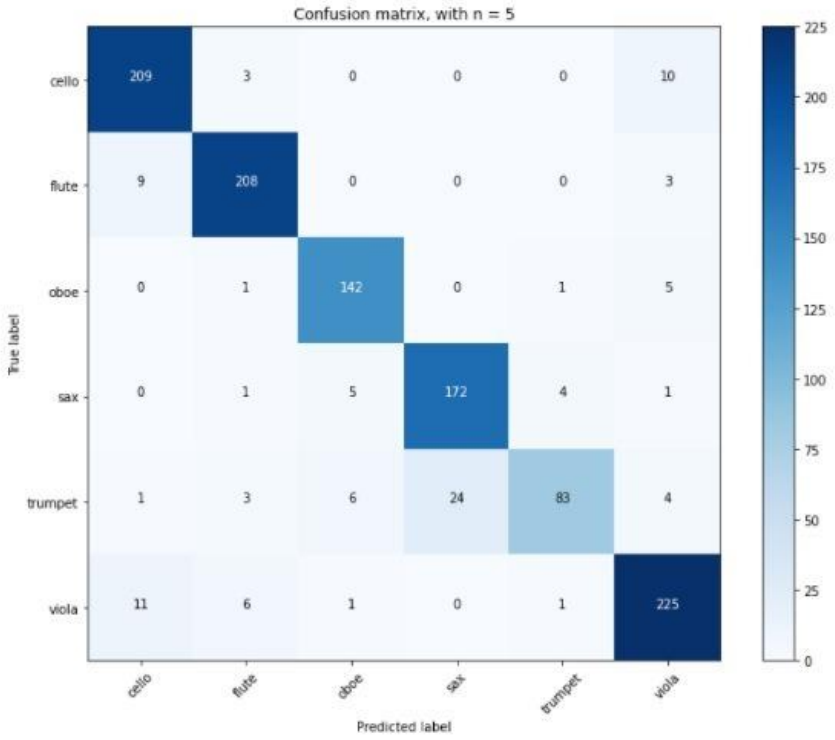
In [108]:
```
#PLOTING CONFUSION MATRIX

plt.figure(figsize = (10,8))
plot_confusion_matrix(cnf_matrix, classes = labelencoder.classes_, title="Confusion matrix, without normalization")
```



In [119]:

```
# with n = 5

model_knn = KNeighborsClassifier(n_neighbors = 5)
#KNN
model_knn.fit(train_set, train_classes);
#Predict using the test set
predicted_labels = model_knn.predict(test_set)

cnf_matrix = confusion_matrix(test_classes, predicted_labels)
np.set_printoptions(precision=2)

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("Normalize confusion matrix")
    else:
    print('Confusion matrix, without normalizaton')
    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')

#Recall the ability of the classifier to find all the positive samples
print("Recall: ", recall_score(test_classes, predicted_labels, average= None))

#Precision - The precision is intuitively the ability of the classifier not to label as positive a sample that is negative
print("Precision: ", precision_score(test_classes, predicted_labels, average=None))

#F1-Score- The F1 score can be interpreted as a weighted average of the precision and recall
print("F1-Score: ",f1_score(test_classes,predicted_labels,average=None))

#Accuracy - the number of correctly classified samples
print("Accuracy: %.2f ,"%accuracy_score(test_classes, predicted_labels,normalize = True), accuracy_score(test_classes, predicted
print("Number of samples: ", test_classes.shape[0])

plt.figure(figsize = (10,8))
plot_confusion_matrix(cnf_matrix, classes = labelencoder.classes_, title="Confusion matrix, with n = 5")
```

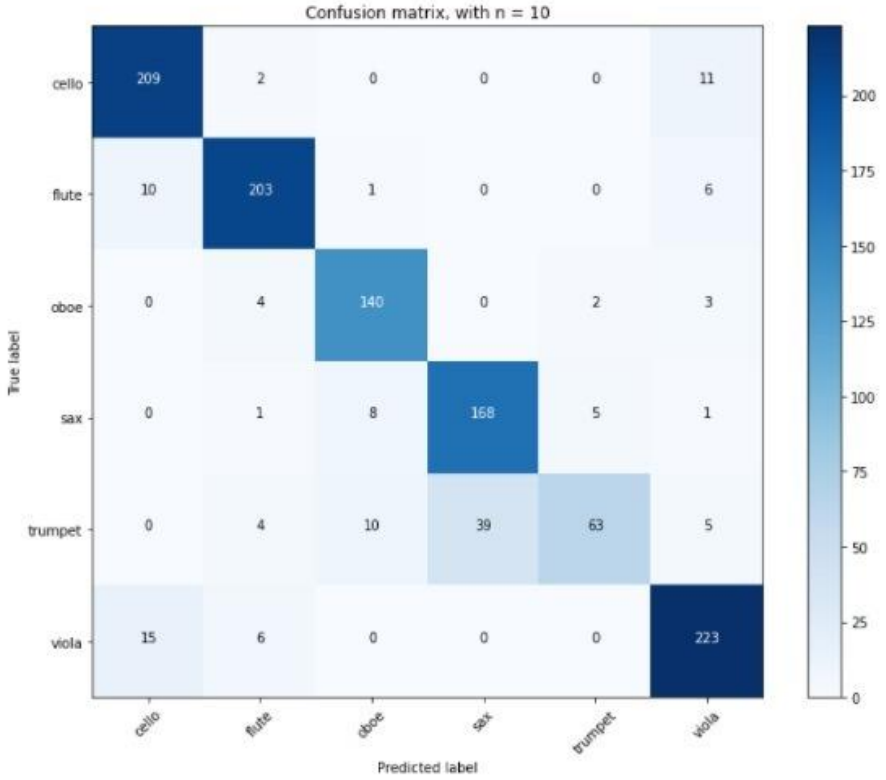
```
Recall:  [0.94 0.95 0.95 0.94 0.69 0.92]
Precision:  [0.91 0.94 0.92 0.88 0.93 0.91]
F1-Score:  [0.92 0.94 0.94 0.91 0.79 0.91]
Accuracy: 0.91 , 1039
Number of samples:  1139
```



In [143]:

```
# with n = 10

model_knn = KNeighborsClassifier(n_neighbors = 10)
#KNN
model_knn.fit(train_set, train_classes);
#Predict using the test set
predicted_labels = model_knn.predict(test_set)

#Recall the ability of the classifier to find all the positive samples
print("Recall: ", recall_score(test_classes, predicted_labels, average= None))

#Precision - The precision is intuitively the ability of the classifier not to label as positive a sample that is negative
print("Precision: ", precision_score(test_classes, predicted_labels, average=None))

#F1-Score- The F1 score can be interpreted as a weighted average of the precision and recall
print("F1-Score: ",f1_score(test_classes,predicted_labels,average=None))

#Accuracy - the number of correctly classified samples
print("Accuracy: %.2f ,"%accuracy_score(test_classes, predicted_labels,normalize = True), accuracy_score(test_classes, predicted
print("Number of samples: ", test_classes.shape[0])

cnf_matrix = confusion_matrix(test_classes, predicted_labels)
np.set_printoptions(precision=2)

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("Normalize confusion matrix")
    else:
    print('Confusion matrix, without normalizaton')
    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.figure(figsize = (10,8))
plot_confusion_matrix(cnf_matrix, classes = labelencoder.classes_, title="Confusion matrix, with n = 10")
```

```
Recall:  [0.94 0.92 0.94 0.92 0.52 0.91]
Precision:  [0.89 0.92 0.88 0.81 0.9  0.9 ]
F1-Score:  [0.92 0.92 0.91 0.86 0.66 0.9 ]
Accuracy: 0.88 , 1006
Number of samples:  1139
```



CONCLUSION

We have described a system that can listen the musical instrument tone and recognize it. The work began with reviewing Blind Source Separation and Musical Instrument Recognition. Features which make musical instrument distinct from each other are presented and discussed. The principle of classifier k-NN are described. Features are extracted from approximated sources and normalized to keep less complex. The k-NN classifier is used to assess the testing data on this identification system. To make truly naturalistic evaluations, the acoustic data would be needed is more.

From the above discussion we can say that, if we find accuracy of identification with consideration of respective family of an instrument, all feature provide more improved result than those are with instrument wise identification. The process is become very complicated if we try to find best feature value of an individual instrument.

REFERENCES

1. http://cs229.stanford.edu/proj2015/010_report.pdf
2. https://arxiv.org/pdf/1705.04971.pdf

3. http://aircconline.com/sipij/V4N4/4413sipij08.pdf
4. https://github.com/GuitarsAI/BasicsMusicalInstrumClassifi
5. https://philharmonia.co.uk/resources/sound-samples/
6. MUSICAL INSTRUMENT RECOGNITION USING MACHINE LEARNING TECHNIQUE, Sumit S. Bhojane1,Omkar G. Labhshetwar2,Kshitiz Anand3, Prof. S. R. Gulhane4
7. http://www.eecs.qmul.ac.uk/legacy/easaier/files/technical/identification/instrument_recognition.htm