

## Exercise 9: Sound and music description

With this exercise you will learn to describe sounds with simple machine learning methods. You will learn to use the Freesound API to load pre-computed sound descriptors from Freesound and to perform sound clustering and classification with them. You will work with instrumental sounds, thus learning what audio features are useful for characterizing them.

There are four parts in this exercise: 1) Download sounds and descriptors from Freesound, 2) Select two descriptors for a good sound clustering, 3) Cluster sounds using k-means, and 4) Classify sounds using k-NN.

We provide the code for each task, thus no need to get involved in much programming

### Relevant Concepts

#### Freesound API

With the Freesound API you can browse, search, and retrieve information from Freesound, such as automatically extracted features from audio files. You can also perform advanced queries combining content analysis features and other metadata (tags, etc...). With the API you can do text searches similar to what you can do from the advanced searches in the website <http://freesound.org/search/?q>, but implementing the queries in software. If you are interested in knowing more about the Freesound API, you can see examples of using it with python in: <https://github.com/MTG/freesound-python/blob/master/examples.py> and you can read the API documentation, <http://www.freesound.org/docs/apl/>

#### Sound descriptors

In this exercise, you will use sound descriptors that have been pre-computed with Essentia, <https://essentia.upf.edu> and are stored in the Freesound database together with the corresponding sounds. Many sound descriptors can be extracted using Essentia ([http://essentia.upf.edu/documentation/algorithms\\_reference.html](http://essentia.upf.edu/documentation/algorithms_reference.html)) and in Freesound, a number of them are used. In the Freesound API documentation there is specific information of the descriptors available in Freesound, [https://freesound.org/docs/apl/analysis\\_index.html](https://freesound.org/docs/apl/analysis_index.html).

#### Euclidian distance

The Euclidian distance is the straight-line distance between two points in an n-dimensional space, thus the distance between points  $p$  and  $q$  is the length of the line segment connecting them. If  $p = (p_1, p_2, \dots, p_n)$  and  $q = (q_1, q_2, \dots, q_n)$  are two points in Euclidean n-space, then the distance,  $d$ , from  $p$  to  $q$ , or from  $q$  to  $p$  is given by the Pythagorean formula:

$$d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

#### K-means clustering (k-means)

K-means clustering is a method of vector quantization that is popular for cluster analysis in data mining. K-means clustering aims to partition  $n$  observations into  $k$  clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. The problem is computationally difficult (NP-hard), however, efficient heuristic algorithms converge quickly to a local optimum.

Given a set of observations  $(x_1, x_2, \dots, x_n)$ , where each observation is a  $d$ -dimensional real vector, k-means clustering aims to partition the set into  $k$  ( $k \leq n$ ) sets  $S = S_1, S_2, \dots, S_k$  so as to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$$\arg \min_{S_i} \sum_{i=1}^k \sum_{x \in S_i} \|x - \mu_i\|^2 = \arg \min_{S_i} \sum_{i=1}^k |S_i| \text{Var } S_i, \text{ where } \mu_i \text{ is the mean of points in } S_i.$$

#### K-nearest neighbours classifier (k-NN)

K-nearest neighbours classification (k-NN) is a non-parametric method in which the input consists of the  $k$  closest training examples in the feature space. The output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its  $k$  nearest neighbors ( $k$  is a positive integer, typically small). If  $k = 1$ , then the object is simply assigned to the class of that single nearest neighbor.

## Part 1: Download sounds and descriptors from Freesound

Download a collection of instrumental sounds and their descriptors from Freesound using the Freesound API.

First get a Freesound API key from <http://www.freesound.org/apiv2/appl/> and create a directory in the workspace with the name `testDownload` to store the sounds and descriptors. You also need to install the python client for the freesound API. Do this by cloning the repository <https://github.com/MTG/freesound-python>, into a folder and installing it following the instructions. You will have to import the freesound module in the code.

You will be calling the function `download_sounds_freesound()` whose input parameters are:

- `queryText` (string): A single word or a string of words without spaces (use hyphens), typically the name of the instrument. e.g. (eg. "violin", "trumpet", "cello", "bassoon", etc.).
- `tag` (string): tag to be used for filtering the searched sounds (e.g., "multisample", "single-note", "velocity", "tenuato", etc.).
- `duration` (2 floating point numbers): min and max duration (seconds) of the sound to filter, e.g., (0.2,15).
- `API_Key` (string): your API key.
- `outputDir` (string): path to the directory where you want to store the sounds and their descriptors.
- `topNResults` (integer): number of results (sounds) that you want to download.
- `featureExt` (file extension): file extension for storing sound descriptor (.json, typically).

In the call to `download_sounds_freesound()` you will have to choose the appropriate `queryText`, `tag`, and `duration`, to return single notes/strokes of instrumental sounds. The first twenty results of the query should be "good". Note that the tag can be empty. Example of a query to obtain single notes of violin could be:

```
download_sounds_freesound(queryText='violin', API_Key=your key, outputDir='testDownload/', topNResults=20, duration=(0,8.5), tag='single-note')
```

This returns 20 single notes of violin sounds and the script stores them in the `testDownload` directory (the directory has to be created beforehand).

Download and store twenty sound examples of each instrument using the `download_sounds_freesound()` function given below. The examples need to be representative of the instrument and coherent, single notes (melodic instruments) or single strokes (percussion instruments), and shorter than 10 seconds. Specify a good query text, tag, and duration to query for the chosen instruments. Refine your search parameters until you get twenty adequate samples for each instrument. Select three instruments to be used of this set: violin, guitar, bassoon, trumpet, clarinet, cello, naabo (cymbals used in Beijing Opera). Listen to the sounds downloaded and look at the descriptor .json files.

Before using the API to download the sounds, we recommend to do the same query using the Freesound website and checking that the top 20 results are good.

```
In [1]: import os, sys
import json
import freesound as fs

descriptors = [ 'lowlevel_spectral_centroid.mean',
                'lowlevel_spectral_contrast.mean',
                'lowlevel_dissonance.mean',
                'lowlevel_hfc.mean',
                'lowlevel_mfcc.mean',
                'sfx_logattacktime.mean',
                'sfx_inharmonicity.mean' ]

In [2]: def download_sounds_freesound(queryText = "", tag=None, duration=None, API_Key = "", outputDir = "", topNResults = 5, featureExt = '.json'):
    """
    This function downloads sounds and their descriptors from freesound using the queryText and the
    tag specified in the input. Additionally, you can also specify the duration range to filter sounds
    based on duration.

    Inputs:
        (Input parameters marked with a * are optional)
        queryText (string): query text for the sounds (eg. "violin", "trumpet", "cello", "bassoon" etc.)
        tag* (string): tag to be used for filtering the searched sounds. (eg. "Multisample",
            "single-note" etc.)
        duration* (tuple): min and the max duration (seconds) of the sound to filter, eg. (0.2,15)
        API_Key (string): your api key, which you can obtain from : www.freesound.org/apiv2/appl/
        outputDir (string): path to the directory where you want to store the sounds and their
            descriptors
        topNResults (Integer): number of results(sounds) that you want to download
        featureExt (string): file extension for storing sound descriptors

    output:
        This function downloads sounds and descriptors, and then stores them in outputDir. In
        outputDir it creates a directory of the same name as that of the queryText. In this
        directory outputDir/queryText it creates a directory for every sound with the name
        of the directory as the sound id. Additionally, this function also dumps a text file
        containing sound-ids and freesound links for all the downloaded sounds in the outputDir.
        NOTE: If the directory outputDir/queryText exists, it deletes the existing contents
        and stores only the sounds from the current query.

    """

    # Checking for the compulsory input parameters
    if queryText == "":
        print("\n")
        print("Provide a query text for sounds")
        return -1

    if API_Key == "":
        print("\n")
        print("You need a valid freesound API key to be able to download sounds.")
        print("Please apply for one here: www.freesound.org/apiv2/appl/")
        print("\n")
        return -1

    if outputDir == "" or not os.path.exists(outputDir):
        print("\n")
        print("Please provide a valid output directory. This will be the root directory for storing sounds and descriptors")
        return -1

    # Setting up the freesound client and the authentication key
    fsCInt = fs.FreesoundClient()
    fsCInt.set_token(API_Key, "token")

    # Creating a duration filter string that the freesound API understands
    if duration and type(duration) == tuple:
        ftt_dur = " duration:[* + str(duration[0])+ " " T0 " +str(duration[1]) + "]"
    else:
        ftt_dur = ""

    if tag and type(tag) == str:
        ftt_tag = "tag:'*tag
    else:
        ftt_tag = ""

    # Querying Freesound
    page_size = 30
    if not ftt_tag + ftt_dur == "":
        qRes = fsCInt.text_search(query=queryText, filter = ftt_tag + ftt_dur, sort="score", fields="id,name,previews,url,analysis", descriptors=',')
    else:
        qRes = fsCInt.text_search(query=queryText, sort="score", fields="id,name,previews,url,analysis", descriptors=',').join(descriptors=',').join(descriptors=',')

    outDir2 = os.path.join(outputDir, queryText)
    if os.path.exists(outDir2):
        os.system("rm -r " + outDir2)
    os.mkdir(outDir2)

    # System quits after trying to download after 200 times
    totalSnds = min(qRes.count,200)

    # Creating directories to store output and downloading sounds and their descriptors
    downloadedSounds = []
    while(1):
        if indCnt >= totalSnds:
            print("Not able to download required number of sounds. Either there are not enough search results on freesound for your search query and filtering
            break
        sound = qRes[indCnt - ((pageNo-1)*page_size)]
        print("Downloading mp3 preview and descriptors for sound with id: %s"%str(sound.id))
        outDir1 = os.path.join(outputDir, queryText, str(sound.id))
        if os.path.exists(outDir1):
            os.system("rm -r " + outDir1)
            os.mkdir("mdir " + outDir1)

        mp3Path = os.path.join(outDir1, str(sound.previews.preview_lq_mp3.split("/")[-1]))
        ftrPath = mp3Path.replace('.',mp3',', featureExt)

        try:
            fs.FSRequest.retrieve(sound.previews.preview_lq_mp3, fsCInt, mp3Path)
        except:
            # Initialize a dictionary to store descriptors
            features = {}
            # Obtaining all the descriptors
            for desc in descriptors:
                features[desc]=[]
                features[desc].append(eval("sound.analysis."+desc))

            # Once we have all the descriptors, store them in a json file
            json.dump(features, open(ftrPath,'w'))
            sndCnt+=1
            downloadedSounds.append([str(sound.id), sound.url])

        except:
            if os.path.exists(outDir1):
                os.system("rm -r " + outDir1)

            indCnt +=1

        if indCnt==page_size==0:
            qRes = qRes_next_page()
            pageNo+=1

        if sndCnt==topNResults:
            break
        # Dump the list of files and Freesound links
        fId = open(os.path.join(outDir2, queryText+'SoundList.txt'),'w')
        for elem in downloadedSounds:
            fId.write('\t'.join(elem)+'\n')
        fId.close()

In [3]: # 1.1: call download_sounds_freesound for 3 instruments with parameters to obtain adequate sounds
#### your code here

#### explain the coherence of the sound collections obtained
"""
"""
```

## Part 2: Select two descriptors for a good sound clustering

Select two of the sound descriptors obtained from Task 1 in order to obtain a good clustering of the sounds of three instruments in a two dimensional space. By visualizing the descriptor values of the sounds in a 2D plot you can choose the features that can help to better cluster these instruments.

You take as inputs the downloaded sounds folder (`targetDir`) and the descriptor pair indices (`descInput`) (see mapping) to create a 2-D scatter plot of the descriptor pair. The data points, sounds, from different instruments are shown with different colors. In addition, you can also plot the Freesound ID of the sounds with the points. Only plot the sounds of the 3 instruments chosen. Make sure that in `targetDir` you only have the 3 instruments chosen.

Choose a good pair of descriptors for the sound separation of the 3 instruments you downloaded in Part 1. A good pair of descriptors leads to a point distribution where all the sounds of an instrument cluster together, with a good separation from the other instrument clusters. Try out different combinations of descriptor pairs. Write a short paragraph on the descriptor pairs you tried out, justifying your choices for selecting those particular descriptors. Based on the spectral and temporal features of the instruments and sounds, give an explanation of why (or why not) a good clustering is (or is not) achieved with the chosen pairs of descriptors.

From the code given you can generate a 2-D scatter plot of all sounds for the chosen descriptor pairs.

```
In [4]: import os, sys
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from scipy.cluster.vq import vq, kmeans

# Mapping of descriptors
descriptorMapping = { 0: 'lowlevel_spectral_centroid.mean',
                      1: 'lowlevel_dissonance.mean',
                      2: 'lowlevel_hfc.mean',
                      3: 'sfx_logattacktime.mean',
                      4: 'sfx_inharmonicity.mean',
                      5: 'lowlevel_spectral_contrast.mean.0',
                      6: 'lowlevel_spectral_contrast.mean.1',
                      7: 'lowlevel_spectral_contrast.mean.2',
                      8: 'lowlevel_spectral_contrast.mean.3',
                      9: 'lowlevel_spectral_contrast.mean.4',
                      10: 'lowlevel_spectral_contrast.mean.5',
                      11: 'lowlevel_mfcc.mean.0',
                      12: 'lowlevel_mfcc.mean.1',
                      13: 'lowlevel_mfcc.mean.2',
                      14: 'lowlevel_mfcc.mean.3',
                      15: 'lowlevel_mfcc.mean.4',
                      16: 'lowlevel_mfcc.mean.5'
                    }

In [5]: def convFtrDict2List(ftrDict):
    """
    This function converts descriptor dictionary to an np.array. The order in the numpy array (indices)
    are same as those mentioned in descriptorMapping dictionary.

    Input:
        ftrDict (dict): dictionary containing descriptors downloaded from the freesound
    Output:
        ftr (np.ndarray): Numpy array containing the descriptors for processing later on

    for key in range(len(descriptorMapping.keys())):
        try:
            fName, ind = ','.join(descriptorMapping[key].split(',')[:-1]), int(descriptorMapping[key].split(',')[-1])
            ftr.append(ftrDict[fName][0][ind])
        except:
            ftr.append(ftrDict[descriptorMapping[key]][0])
    return np.array(ftr)

def fetchDataDetails(inputDir, descExt = '.json'):
    """
    This function is used by other functions to obtain the information regarding the directory structure
    and the location of descriptor files for each sound

    """
    dataDetails = {}
    for path, npName, fNames in os.walk(inputDir):
        for fName in fNames:
            if descExt in fName.lower():
                remain, fName, cname, sname = path.split('/')[-3], path.split('/')[-2], path.split('/')[-1]
            if cname not in dataDetails:
                dataDetails[cname]={}
            fDict = json.load(open(os.path.join('/', join(remain), fName, cname, sname, fName, 'r'))
            dataDetails[cname][sname]=['file': fName, 'feature': fDict]
    return dataDetails

In [6]: # 2.2: Select the descriptors to plot of the three instruments chosen
InputDir = "testDownload/"
#### this is the main line to modify, select two descriptors, change the XX by a number from 0 to 16

descInput = (XX,XX)

# no need to change the code here
anoCnt = 0
dataDetails = fetchDataDetails(inputDir)
colors = ['r', 'g', 'c', 'b', 'k', 'm', 'y']

plt.figure(figsize=(15, 10))

legArray = []
catArray = []
for i4, category in enumerate(dataDetails.keys()):
    catArray.append(category)
    for soundId in dataDetails[category].keys():
        filepath = os.path.join(inputDir, category, soundId, dataDetails[category][soundId]['file'])
        descSound = convFtrDict2List(json.load(open(filepath, 'r')))
        x_cord = descSound[descInput[0]]
        y_cord = descSound[descInput[1]]
        plt.scatter(x_cord, y_cord, c = colors[i4], s=200, alpha=0.75)
        if anoCnt==1:
            plt.annotate(soundId, xy=(x_cord, y_cord), xytext=(x_cord, y_cord+10), markerfacecolor=colors[i4])
            legArray.append(circ)
        circ = Line2D([0], [0], linestyle="none", marker="o", alpha=0.75, markersize=10, markerfacecolor=colors[i4])
        legArray.append(circ)

plt.ylabel(descriptorMapping[descInput[1]])
plt.xlabel(descriptorMapping[descInput[0]])
plt.legend(legArray, catArray, numpoints=1, bbox_to_anchor=(0.,1.02,1,.,102), loc=3, ncol=len(catArray), mode="expand", borderaxespad=0.)
#### Explain the results
"""
"""
```

## Part 3: Cluster sounds using k-means

After visualizing the sound descriptors, you will now cluster the sounds using more than two descriptors. You can use as many descriptors as you need for the best clustering. Use the same set of sounds obtain in Part 1, starting from the descriptors that you found were good in Part 2, and then adding other descriptors that you feel can improve the means clustering of sounds. The function `cluster_sounds()` takes the sounds folder (`targetDir`), number of clusters (`nCluster`) and the descriptor indices (`descInput`) as inputs. It then performs a kmeans clustering using the selected descriptors. Make sure that in `targetDir` you only have the 3 instruments chosen.

For this part, you can use as many descriptors as you need to achieve good clustering and classification performance. However it is best to use as few descriptors as possible in order to make it easier to explain the contribution of each descriptor. Choose the number of clusters to be the same as the number of instruments (i.e., 3). Ideally in such a case, all the sounds of an instrument should go into a single cluster. In reality however, there might be sounds that are outliers and can go into a different cluster. The algorithm takes a majority vote on the sounds in each of the three clusters and assigns each cluster to an instrument. We compute the performance of the clustering by checking the number of points (sounds), that have been wrongly assigned to a cluster. The function `clusterSounds()` prints the clusters and the sounds assigned to each one. The function also prints the resulting classification obtained with the choice of descriptors you made.

Write a short paragraph explaining the descriptors you used, the resulting classification accuracy you obtained, and your observations on why you obtained (or not) those errors in clustering. Comment if you see any systematic errors (such as a consistent mix up of sounds from two instruments) and possible reasons for that. You should also try to cluster with different subsets of descriptors and mention the classification accuracy you obtain in each case.

Note: Since the cluster centers are randomly initialized every time in k-means, you might see different results every time you run the function. You can report the best result you obtained.

```
In [7]: def cluster_sounds(targetDir, nCluster = -1, descInput=[]):
    """
    This function clusters all the sounds in targetDir using kmeans clustering.

    Inputs:
        targetDir (string): Directory where sound descriptors are stored (all the sounds in this
            directory will be used for clustering)
        nCluster (int): Number of clusters to be used for kmeans clustering.
        descInput (list): List of indices of the descriptors to be used for similarity/distance
            computation (see descriptorMapping)

    Output:
        Prints the class of each cluster (computed by a majority vote), number of sounds in each
        cluster and information (sound-id, sound-class and classification decision) of the sounds
        in each cluster. Optionally, you can uncomment the return statement to return the same data.

    """
    dataDetails = fetchDataDetails(targetDir)

    ftrArr = []
    infoArr = []

    if nCluster ==-1:
        nCluster = len(dataDetails.keys())
    for cname in dataDetails.keys():
        #Iterating over sounds
        for sname in dataDetails[cname].keys():
            ftrArr.append(convFtrDict2List(dataDetails[cname][sname]['feature']))
            infoArr.append([sname, cname])

    ftrArr = np.array(ftrArr)
    infoArr = np.array(infoArr)

    centroids, distortion = kmeans(ftrArrWhite, nCluster)
    clusResults = [vnp.ones(ftrArrWhite.shape[0])

    for i1 in range(ftrArrWhite.shape[0]):
        diff = centroids - ftrArrWhite[i1,:]
        diff = np.sum(np.power(diff,2), axis = 1)
        indMin = np.argmin(diff)
        clusResults[i1] = indMin

    ClusterOut = []
    classCluster = []
    globalDecisions = []
    for i1 in range(nCluster):
        ind = np.where(clusResults==i1)[0]
        freqCnt = []
        for elem in infoArr[ind,1]:
            freqCnt.append(infoArr[ind,1].tolist().count(elem))
        indMax = np.argmax(freqCnt)
        classCluster.append(infoArr[ind,1][indMax])

    print("\nCluster: " + str(i1) + ") Using majority voting as a criterion this cluster belongs to " +
        "class: " + classCluster[i1])
    print ("Number of sounds in this cluster are: " + str(len(ind)))
    decisions = []
    for i1 in ind:
        if infoArr[i1,1] == classCluster[i1]:
            decisions.append(1)
        else:
            decisions.append(0)
    globalDecisions.extend(decisions)
    print ("Sound-Id, sound-class, classification decision")
    ClusterOut.append(np.hstack([infoArr[ind], np.array(decisions)].T))
    print (ClusterOut[i1])
    globalDecisions = np.array(globalDecisions)
    nIncorrectClassified = len(np.where(globalDecisions==0)[0])
    print("Out of %d sounds, %d sounds are incorrectly classified considering that one cluster should "
        "ideally contain sounds from only a single class"%(totalSounds, nIncorrectClassified))
    print("You obtain a classification based on obtained clusters and majority voting) accuracy "
        "%d %% (%.2f percentage)"%round(float(100.0*float(totalSounds-nIncorrectClassified)/totalSounds),2))
    return ClusterOut

In [8]: # 3.1: Run the function clusterSounds
#### your code here

Question E9 - 3.2:

Explain the results
```

## Part 4: Classify sounds with k-NN

Assign a sound different from the sounds of the 3 instruments chosen to one of the 3 instrumental classes you chose in Part 1, using the k-nearest neighbours classifier (k-NN).

Given a new sound (query sound) and its descriptors, use the function `classify_sound_kNN()` for doing a k-NN classification. It uses a distance measure based on Euclidian distance which is implemented in `compute_similar_sounds()`. `classify_sound_kNN()` returns the instrument class that the query sound is classified into.

The goal of the exercise is to experiment with the k-NN classifier and be able to understand the result by being able to explain why a particular query sound, that is not from any of the defined classes, is actually classified to one of those classes.

4.1 Get sounds from Freesound. To get query sounds and their descriptors, you can use `download_sounds_freesound()` function using different query texts (as you did in Part 1). Get sounds that are not from the 3 instruments you chose in Part 1, or at least that is none of the sounds you used to define the classes. If you use a sound from one of the three instruments make sure that is different enough from the existing sounds in your collections.

4.2 Perform 5 classifications. You can use as many descriptors as you need (the fewer you use, the easier it will be, to explain the result). k is usually chosen to be an odd positive integer. Try to work with different query sounds, different subsets of descriptors, and different values of k. Explain the reason for choosing the descriptors you used and the value of k you selected. Include cases where you think the classification is incorrect, and cases with a query sound of an instrument different from the starting classes. By trying out a sound from a different instrument, you are classifying the sound into the closest instrument class, which can tell you about similarity between instruments.

```
In [9]: def compute_similar_sounds(queryFile, targetDir, descInput = []):
    """
    This function returns similar sounds for a specific queryFile. Given a queryFile this function
    computes the distance of the query to all the sounds found in the targetDir and sorts them in
    the increasing order of the distance. This way we can obtain similar sounds to a query sound.

    Inputs:
        queryFile (string): Descriptor file (.json, unless changed)
        targetDir (string): Target directory to search for similar sounds (using their descriptor files)
        descInput (list): List of indices of the descriptors to be used for similarity/distance computation
            (see descriptorMapping)

    Output:
        List containing an ordered list of similar sounds.

    """
    dataDetails = fetchDataDetails(targetDir)

    #reading query feature dictionary
    qFtr = json.load(open(queryFile, 'r'))
    dist = []

    # Iterating over classes
    for cname in dataDetails.keys():
        # Iterating over sounds
        for sname in dataDetails[cname].keys():
            f1 = convFtrDict2List(qFtr)
            f2 = convFtrDict2List(dataDetails[cname][sname]['feature'])
            euclDist = np.sqrt(np.sum(np.power(np.array(f1[descInput]) - np.array(f2[descInput]), 2)))
            dist.append((euclDist, sname, cname))

    # Sorting the array based on the distance
    indSort = np.argsort(np.array(dist)[:,:0])
    return (np.array(dist)[indSort,:]).tolist()

def classify_sound_kNN(queryFile, targetDir, K, descInput = []):
    """
    This function performs the kNN classification of a sound. The nearest neighbors are chosen from
    the sounds in the targetDir.

    Input:
        queryFile (string): Descriptor file (.json, unless changed)
        targetDir (string): Target directory to search for similar sounds (using their descriptor files)
        K (int): Number of nearest neighbors to consider for kNN classification.
        descInput (list): List of indices of the descriptors to be used for similarity/distance computation
            (see descriptorMapping)

    Output:
        predClass (string): Predicted class of the query sound

    """
    distances = compute_similar_sounds(queryFile, targetDir, descInput)
    if len(np.where(np.array(distances)[:,:0].astype(np.float64)==0)[0])>0:
        print("Warning: We found an exact copy of the query file in the target directory. "
            "Beware of duplicates while doing kNN classification.")

    classes = (np.array(distances)[:,:2]).tolist()
    freqCnt = []
    for i1 in range(K):
        freqCnt.append(classes.count(classes[i1]))
    indMax = np.argmax(freqCnt)
    predClass = classes[indMax]
    print ("This sample belongs to class: " + str(predClass))
    return predClass

In [10]: # 4.1 Download one sound
#### your code here

Question E9 - 4.2:

Explain your choices
```

```
In [11]: # 4.3 Classify the downloaded sound
#### your code here

Question E9 - 4.4:

Explain the classification
```