ДОДАТОК А

Графічний матеріал атестаційної роботи

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

ФАКУЛЬТЕТ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА УПРАВЛІННЯ
КАФЕДРА КІТС

Інтелектуальна система класифікації стилів музики

Магістрант гр. КІТм-21-1 Ушаков М. Р.
Науковий керівник проф. Аксак Н. Г.

Харків 2022

Мета роботи

Метою даної роботи є розробка системи розпізнавання музичних стилів за допомогою нейронних мереж різних типів з метою отримання найкращого методу розпізнавання.

Для досягнення поставленої мети необхідно виконати наступні завдання:

- Дослідити існуючі способи розпізнавання музикальних стилів;
- Розробити систему яка зможе з найвищою точністю передбачити музикальний жанр;
- Проаналізувати результати, отримані під час моделювання;
- Створити систему використовуючи отримані дані.

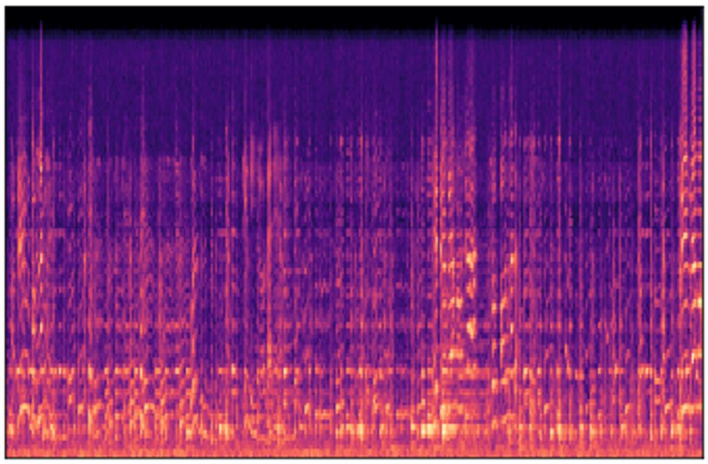
Основні методи класифікації музики

- Короткочасні
- Довгострокові
- Семантичничні
- Композиційні

Кепстральний коефіцієнт Мела

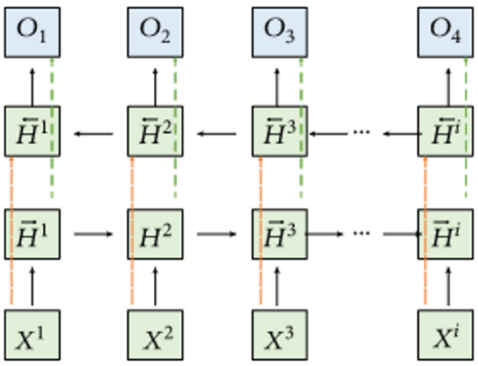
У обробці звуку кепстральний коефіцієнт Мела— це представлення короткочасного спектру потужності звуку на основі лінійного косинусного перетворення логарифмічного спектра потужності на нелінійній шкалі mel частоти.

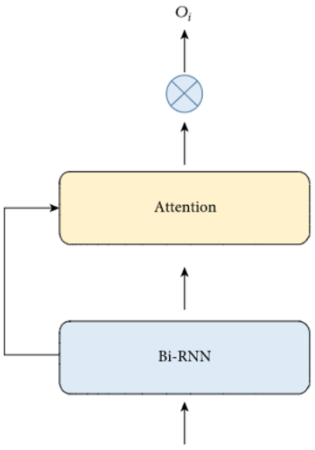
Вигляд Мел-спектрограми



5

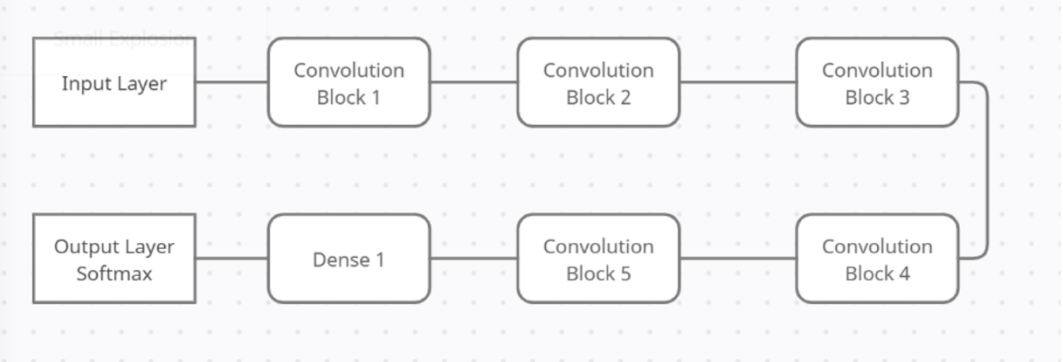
Структура рекурсивної нейронної мережі глибокого навчання





6

Опис згорткової нейронної мережі глибокого навчання



7

Порівняння результатів рекурсивної та згорткової нейронних мереж(1)

Таблиця 1 – Експерименти та результати моделі рекурсивної нейронної мережі.

№	Епохи	Точність	Втрата
1	30	74%	72%
2	30	67%	90%
3	30	74%	71%
4	100	75%	69%
5	30	90%	32%

Таблиця 2 – Експерименти та результати моделі згорткової нейронної мережі.

№	Епохи	Точність	Втрата
1	30	67%	88%
2	30	64%	98%
3	30	67%	88%
4	100	69%	85%
5	30	92%	23%

8

Порівняння результатів рекурсивної та згорткової нейронних мереж(2)

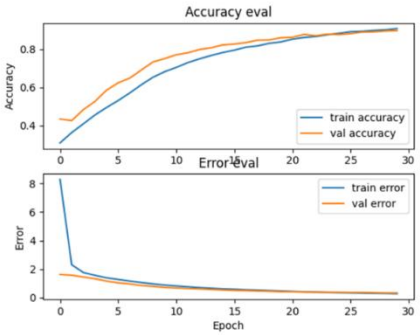


Рис 1 - Навчальний процес п’ятого експерименту для моделі RNN

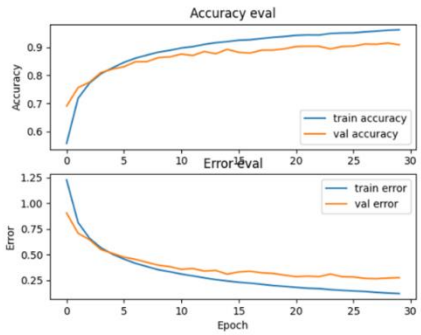
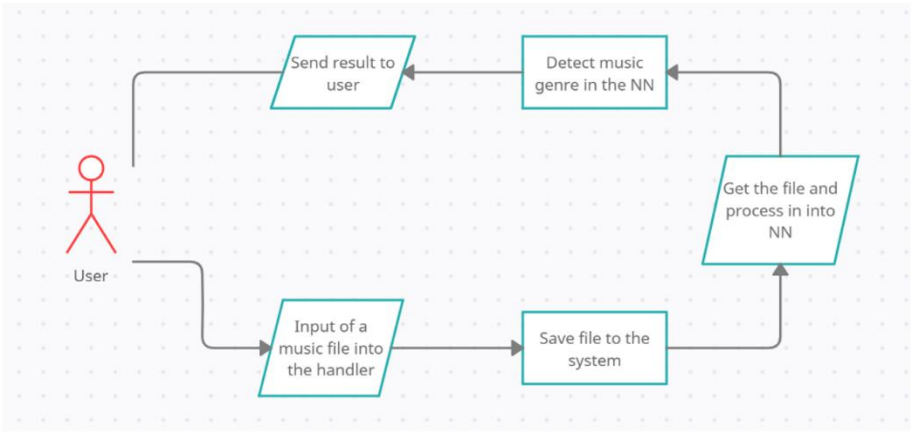


Рис 3.5 - П'ятий експериментальний навчальний процес для моделі CNN

Модель системи розпізнавання жанру музики



Результати роботи

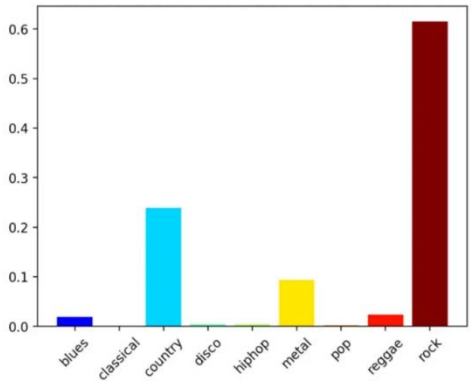


Рисунок 1 – Визначення жанру при використанні пісні Enter The Sandman - Metallica

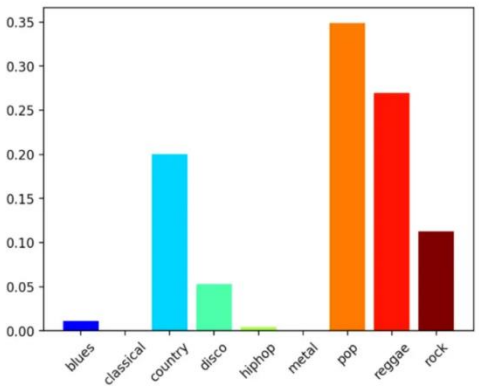


Рисунок 2 – Визначення жанру при використанні пісні Viva La Vida – Coldplay

Висновки

В ході дипломної роботи було розроблено та порівняно декілька систем для розв’язання поставленої проблеми, а саме для розпізнавання музикальних стилів. Для цього було розроблено систему рекурсивної нейронної мережі та нейронної мережі згорткового типу глибокого навчання. В результаті роботи було встановлено що використання згорткової мережі більш ефективне для розпізнавання стилів музики.

ДОДАТОК Б
КОД ПРОГРАММИ

Б.1 Код згорткової нейронної мережі

```
import math
import numpy as np
import h5py
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.python.framework import ops

def random_mini_batches(X, Y, mini_batch_size = 64, seed = 0):


    m = X.shape[1]                  # number of training examples
    mini_batches = []
    np.random.seed(seed)


    permutation = list(np.random.permutation(m))
    shuffled_X = X[:, permutation]
    shuffled_Y = Y[:, permutation].reshape((Y.shape[0],m))


    num_complete_minibatches = math.floor(m/mini_batch_size) # 
number of mini batches of size mini_batch_size in your partitionning
    for k in range(0, num_complete_minibatches):
        mini_batch_X = shuffled_X[:, k * mini_batch_size : k * 
mini_batch_size + mini_batch_size]
        mini_batch_Y = shuffled_Y[:, k * mini_batch_size : k * 
mini_batch_size + mini_batch_size]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    # (last mini-batch < mini_batch_size)
    if m % mini_batch_size != 0:
        mini_batch_X = shuffled_X[:, num_complete_minibatches * 
mini_batch_size : m]
        mini_batch_Y = shuffled_Y[:, num_complete_minibatches * 
mini_batch_size : m]
        mini_batch = (mini_batch_X, mini_batch_Y)
        mini_batches.append(mini_batch)

    return mini_batch


def convert_to_one_hot(Y, C):
    Y = np.eye(C)[Y.reshape(-1)].T
```

```
        return Y


    def predict(X, parameters):

        W1 = tf.convert_to_tensor(parameters["W1"])
        b1 = tf.convert_to_tensor(parameters["b1"])
        W2 = tf.convert_to_tensor(parameters["W2"])
        b2 = tf.convert_to_tensor(parameters["b2"])
        W3 = tf.convert_to_tensor(parameters["W3"])
        b3 = tf.convert_to_tensor(parameters["b3"])

        params = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2,
                  "W3": W3,
                  "b3": b3}

        x = tf.placeholder("float", [25, 1])

        z3 = forward_propagation_for_predict(x, params)
        p = tf.argmax(z3)

        sess = tf.Session()
        prediction = sess.run(p, feed_dict = {x: X})

        return prediction

    def forward_propagation_for_predict(X, parameters):

        # Retrieve the parameters from the dictionary "parameters"
        W1 = parameters['W1']
        b1 = parameters['b1']
        W2 = parameters['W2']
        b2 = parameters['b2']
        W3 = parameters['W3']
        b3 = parameters['b3']

        Z1 = tf.add(tf.matmul(W1, X), b1)
        A1 = tf.nn.relu(Z1)
        Z2 = tf.add(tf.matmul(W2, A1), b2)
        A2 = tf.nn.relu(Z2)
        Z3 = tf.add(tf.matmul(W3, A2), b3)

        return Z3


    # Loading the dataset
    #X_train_orig, Y_train_orig, X_test_orig, Y_test_orig, classes =
load_dataset()
    import pandas as pd
    from                sklearn.preprocessing                import
```

```
LabelEncoder,OneHotEncoder,StandardScaler
     from sklearn.model_selection import train_test_split
     #data = pd.read_csv('data.csv')
     features = pd.read_csv('features_3_sec.csv')
     #data = data.drop(['filename'],axis=1)
     features = features.drop('filename',axis=1)
     np.random.seed(10)
     permut = list(np.random.permutation(9900))

     genre_list = features.iloc[:, -1]
     encoder = LabelEncoder()
     y = encoder.fit_transform(genre_list)
     onehotencode = OneHotEncoder(sparse=False)
     #print(len(yl))
     y = y.reshape(len(y), 1)
     y = onehotencode.fit_transform(y).T
     print(y.shape)
     scaler = StandardScaler()
     X = scaler.fit_transform(np.array(features.iloc[:, :-1], dtype =
float))
     print(X.shape)
     X = X[permut,:]
     y = y[:,permut]
     x_train, x_test, y_train, y_test = train_test_split(X, y.T,
test_size=0.2,stratify = y.T
                                                         )

     #print(data.iloc[:,:-1].shape)
     #x_train = x_train.reshape((25,720))
     print(x_train.shape)

     x_train = x_train.T
     print(x_train.shape)
     y_train = y_train.T
     print(y_train.shape)
     x_test = x_test.T
     y_test = y_test.T

     features.head()

     def create_placeholders(n_x, n_y):

         X = tf.placeholder(tf.float32,shape = (n_x,None))
         Y =  tf.placeholder(tf.float32,shape = (n_y,None))


         return X, Y

     def initialize_parameters():


         tf.set_random_seed(1)
```

```
        W1      =      tf.get_variable("W1",[128,58],initializer      =
tf.contrib.layers.xavier_initializer(seed=1))
        b1      =      tf.get_variable("b1",[128,1],initializer      =
tf.zeros_initializer())
        W2      =      tf.get_variable("W2",[64,128],initializer      =
tf.contrib.layers.xavier_initializer(seed=1))
        b2      =      tf.get_variable("b2",[64,1],initializer      =
tf.zeros_initializer())
        W3      =      tf.get_variable("W3",[10,64],initializer      =
tf.contrib.layers.xavier_initializer(seed=1))
        b3      =      tf.get_variable("b3",[10,1],initializer      =
tf.zeros_initializer())

        print(W1)


        parameters = {"W1": W1,
                      "b1": b1,
                      "W2": W2,
                      "b2": b2,
                      "W3": W3,
                      "b3": b3
                     }


        return parameters

    def forward_propagation(X, parameters):

        tf.set_random_seed(2)


        W1 = parameters['W1']
        b1 = parameters['b1']
        W2 = parameters['W2']
        b2 = parameters['b2']
        W3 = parameters['W3']
        b3 = parameters['b3']



        Z1 = tf.add(tf.matmul(W1, X), b1)
        A1 = tf.nn.relu(Z1)
        A1 = tf.nn.dropout(A1,keep_prob = 0.9)

        Z2 = tf.add(tf.matmul(W2, A1), b2)
        A2 = tf.nn.relu(Z2)
        A2 = tf.nn.dropout(A2,keep_prob=0.9)

        Z3 = tf.add(tf.matmul(W3, A2), b3)

        return Z3
```

```
    def compute_cost(Z3, Y):

        logits = tf.transpose(Z3)
        labels = tf.transpose(Y)

        cost                                                    =
tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits         =
logits,labels =labels))

        return cost

    def  model(X_train,  Y_train,  X_test,  Y_test,  learning_rate  =
0.001,
              num_epochs  =  250,  minibatch_size  =  64,  print_cost  =
True):


        ops.reset_default_graph()
        tf.set_random_seed(1)
        seed = 3
        (n_x, m) = X_train.shape
        n_y = Y_train.shape[0]
        costs = []


        X, Y = create_placeholders(n_x,n_y)

        parameters = initialize_parameters()

        Z3= forward_propagation(X,parameters)

        cost = compute_cost(Z3,Y)

        optimizer     =     tf.train.AdamOptimizer(learning_rate     =
learning_rate,beta1=0.9,beta2 = 0.999,epsilon = 1e-08).minimize(cost)

        init = tf.global_variables_initializer()

        with tf.Session() as sess:

          sess.run(init)


          for epoch in range(num_epochs):

            epoch_cost = 0.
            num_minibatches = int(m / minibatch_size)
            seed = seed + 1
            minibatches   =   random_mini_batches(X_train,   Y_train,
minibatch_size, seed)

            for minibatch in minibatches:
```

```
                (minibatch_X, minibatch_Y) = minibatch


                _                ,              minibatch_cost              =
sess.run([optimizer,cost],feed_dict = {X:minibatch_X,Y:minibatch_Y}
                                                )


                epoch_cost += minibatch_cost / minibatch_size


            if print_cost == True and epoch % 10 == 0:
                print   ("Cost   after   epoch   %i:   %f"   %   (epoch,
epoch_cost))

            if print_cost == True and epoch % 5 == 0:
                costs.append(epoch_cost)


        plt.plot(np.squeeze(costs))
        plt.ylabel('cost')
        plt.xlabel('iterations (per fives)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()


        parameters = sess.run(parameters)
        print ("Parameters have been trained!")


        correct_prediction = tf.equal(tf.argmax(Z3), tf.argmax(Y))


        accuracy    =    tf.reduce_mean(tf.cast(correct_prediction,
"float"))

        print  ("Train  Accuracy:",  accuracy.eval({X:  X_train,  Y:
Y_train}))
        print   ("Test   Accuracy:",   accuracy.eval({X:   X_test,   Y:
Y_test}))

        return parameters


    parameters = model(x_train, y_train, x_test, y_test)
```

Б.2. Код рекурсивної нейронної мережі

Data Preprocessing.py

```
import librosa
import os
import math
import json


dataset_path = "genres"
jsonpath = "data_json"

sample_rate = 22050
samples_per_track = sample_rate * 30

#30 is the lenght of each dataset sound in seconds


##################################################################
###########

def
preprocess(dataset_path,json_path,num_mfcc=13,n_fft=2048,hop_length=51
2,num_segment=5):
        data = {
                "mapping": [],
                "labels": [],
                "mfcc": []
        }


        samples_per_segment = int(samples_per_track / num_segment)
        num_mfcc_vectors_per_segment  =  math.ceil(samples_per_segment
/ hop_length)

        for         i,         (dirpath,dirnames,filenames)         in
enumerate(os.walk(dataset_path)):

            if dirpath != dataset_path:

                #Adding all the labels
                label = str(dirpath).split('\\')[-1]
                data["mapping"].append(label)

                print("\nInside ",label)

                #Gping through each song within a label
                for f in filenames:
                    file_path = dataset_path +"/" + str(label) + "/"
+ str(f)
                    y, sr = librosa.load(file_path, sr = sample_rate)
```

```
                    #Cutting each song into 5 segments
                    for n in range(num_segment):
                        start = samples_per_segment * n
                        finish = start + samples_per_segment
                        #print(start,finish)
                        mfcc = librosa.feature.mfcc(y[start:finish],
sample_rate, n_mfcc = num_mfcc, n_fft = n_fft, hop_length =
hop_length)
                        mfcc = mfcc.T #259 x 13

                        #Making sure if
                        if len(mfcc) == num_mfcc_vectors_per_segment:
                            data["mfcc"].append(mfcc.tolist())
                            data["labels"].append(i-1)
                            print("Track Name ", file_path, n+1)

        with open(json_path, "w") as fp:
            json.dump(data, fp, indent = 4)


    if __name__ == "__main__":
        preprocess(dataset_path,jsonpath,num_segment=10)
```

Training_RNN.py

```
    import os
    import json
    import tensorflow as tf
    from sklearn.model_selection import train_test_split
    import numpy as np
    import matplotlib.pyplot as plt

    data_path = "data_json"

    def load_data(data_path):
        print("Data loading\n")
        with open(data_path, "r") as fp:
            data = json.load(fp)

        x = np.array(data["mfcc"])
        y = np.array(data["labels"])

        print("Loaded Data")

        return x, y


    def prepare_datasets(test_size,val_size):

        #load the data
        x, y = load_data(data_path)
```

```
        x_train,        x_test,        y_train,        y_test        =
train_test_split(x,y,test_size = test_size)
        x_train,        x_val,        y_train,        y_val        =
train_test_split(x_train,y_train,test_size = val_size)

        return x_train, x_val, x_test, y_train, y_val, y_test


    def build_model(input_shape):


        model = tf.keras.Sequential()

        model.add(tf.keras.layers.LSTM(64, input_shape = input_shape,
return_sequences = True))
        model.add(tf.keras.layers.LSTM(64))

        model.add(tf.keras.layers.Dense(64, activation="relu"))
        #model.add(tf.keras.layers.Dropout(0.3))

        model.add(tf.keras.layers.Dense(10,activation = "softmax"))

        return model

    if __name__ == "__main__":


        x_train,   x_val,   x_test,   y_train,   y_val,   y_test   =
prepare_datasets(0.25, 0.2)

        print(x_train.shape[0])

        input_shape = (x_train.shape[1],x_train.shape[2])
        model = build_model(input_shape)

        # compile model
        optimiser = tf.keras.optimizers.Adam(lr=0.001)
        model.compile(optimizer=optimiser,
                      loss='sparse_categorical_crossentropy',
                      metrics=['accuracy'])

        model.summary()

        """
        # train model
        history = model.fit(x_train, y_train, validation_data=(x_val,
y_val), batch_size=32, epochs=50)

        # plot accuracy/error for training and validation
        #plot_history(history)

        # evaluate model on test set
        test_loss,   test_acc   =   model.evaluate(x_test,   y_test,
```

```
verbose=2)
        print('\nTest accuracy:', test_acc)

        model.save("model_RNN_LSTM.h5")
        print("Saved model to disk")
        """

        model = tf.keras.models.load_model("model_RNN_LSTM.h5")
        print(model.predict(x_test[100]))
```

Predict_RNN_LSTM.py

```
    import tensorflow as tf
    import librosa
    from mp3towav import convert_to_wav

    ##################################################################
##########
    just_path = "genres/blues/"
    song_path = "genres/blues/1.wav"
    song_name = "1"
    ##################################################################
#########

    #Constants which depend on the model. If you train the model with
different values,
    #need to change those values here too
    num_mfcc = 13
    n_fft=2048
    hop_length = 512
    sample_rate = 22050
    samples_per_track = sample_rate * 30
    num_segment = 10
    ##################################################################
###########

    if __name__=="__main__":

        model = tf.keras.models.load_model("model_RNN_LSTM.h5")
        model.summary()

        classes = ["Blues","Classical","Country","Disco","Hiphop",
                   "Jazz","Metal","Pop","Reggae","Rock"]

        class_predictions = []

        samples_per_segment = int(samples_per_track / num_segment)


        if song_path.endswith('.mp3'):
            path_to_save = just_path + song_name+".wav"
            convert_to_wav(song_path,path_to_save)
            song_path = path_to_save
```

```
        else:
            pass

        #load the song
        x, sr = librosa.load(song_path, sr = sample_rate)
        song_length = int(librosa.get_duration(filename=song_path))

        prediction_per_part = []

        flag = 0
        if song_length > 30:
            print("Song is greater than 30 seconds")
            samples_per_track_30 = sample_rate * song_length
            parts = int(song_length/30)
            samples_per_segment_30   =   int(samples_per_track_30   /
(parts))
            flag = 1
            print("Song sliced into "+str(parts)+" parts")
        elif song_length == 30:
            parts = 1
            flag = 0
        else:
            print("Too  short,  enter  a  song  of  length  minimum  30
seconds")
            flag = 2

        for i in range(0,parts):
            if flag == 1:
                print("Song snippet ",i+1)
                start30 = samples_per_segment_30 * i
                finish30 = start30 + samples_per_segment_30
                y = x[start30:finish30]
                #print(len(y))
            elif flag == 0:
                print("Song is 30 seconds, no slicing")

            for n in range(num_segment):
                start = samples_per_segment * n
                finish = start + samples_per_segment
                #print(len(y[start:finish]))
                mfcc      =      librosa.feature.mfcc(y[start:finish],
sample_rate,  n_mfcc  =  num_mfcc,  n_fft  =  n_fft,  hop_length  =
hop_length)
                mfcc = mfcc.T
                #print(mfcc.shape)
                mfcc = mfcc.reshape(1, mfcc.shape[0], mfcc.shape[1])
                #print(mfcc.shape)
                array = model.predict(mfcc)*100
                array = array.tolist()

                #find maximum percentage class predicted

class_predictions.append(array[0].index(max(array[0])))
```

```
            occurence_dict = {}
            for i in class_predictions:
                if i not in occurence_dict:
                    occurence_dict[i] = 1
                else:
                    occurence_dict[i] +=1

            max_key = max(occurence_dict, key=occurence_dict.get)
            prediction_per_part.append(classes[max_key])

        #print(prediction_per_part)
        prediction    =    max(set(prediction_per_part),    key    =
prediction_per_part.count)
        print(prediction)
```