**Music Genre Classification**

1. **Installation of libraries and packages**

!pip install xgboost -U

!pip install librosa -U

# librosa is the package for music and audio analysis

!pip install hyperopt

# hyperopt is designed to accommodate Bayesian optimization algorithms based on Gaussian processes and regression tress.

1. **Import libraries**

import warnings

warnings.simplefilter("ignore", UserWarning)

import os

from tqdm import tqdm

# used to display progress bars which are used in this program

import pickle

# pickle is the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network.

import pandas as pd

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

%matplotlib inline

import librosa

from sklearn.preprocessing import StandardScaler

from sklearn.feature\_selection import RFECV,mutual\_info\_regression

from sklearn.metrics import confusion\_matrix, accuracy\_score,classification\_report

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

from sklearn.decomposition import PCA

from xgboost import XGBClassifier

# eXtreme Gradient Boosting (XGBoost) is a scalable and improved version of the gradient boosting algorithm designed for efficacy, computational speed and model performance. It is an open-source library and a part of the Distributed Machine Learning Community. XGBoost is a perfect blend of software and hardware capabilities designed to enhance existing boosting techniques with accuracy in the shortest amount of time.

from hyperopt import STATUS\_OK, Trials, fmin, hp, tpe

#For hyperparameter tuning

1. **Load Dataset**

df = pd.read\_csv('Data/features\_30\_sec.csv')

df.head()

# we are only considering 30 seconds files because it is still difficult to determine the genre of music based off of 3 seconds.

1. **Exploring the dataset**

df.label.value\_counts()

# to check how many genres are present and how many songs each genre has

df.info()

df.describe()

# Only 1000 songs are in the dataset with no null values

# length is not a relevant variable which needs to be dropped in model training phase

# There are some other music features that this dataset did not extract, for example, spectral contrast, spectral flatness, tonal centroid features etc. Given it is just a small dataset, we can extract them and put it in the dataframe for further analysis

1. **Extracting extra features and data cleaning**

songs\_path = 'Data/genres\_original'

# code to extract the 4 additional features from the 1000 music files present in the dataset.

def extract\_new\_features(song\_path, num\_files = 1000, num\_new\_features = 8):

data\_array = np.empty([num\_files, num\_new\_features])

counter = 0

for root, dirs, files in os.walk(songs\_path):

dirs.sort()

for file, i in zip(sorted(files), tqdm(range(num\_files))):

i = i + (counter-1)\*100

file\_path = os.path.join(root, file)

try:

#extract mean and variance of those 4 features

y, sr = librosa.load(os.fspath(file\_path))

chroma\_cens = librosa.feature.chroma\_cens(y=y, sr=sr)

spectral\_contrast = librosa.feature.spectral\_contrast(y=y, sr=sr)

spectral\_flatness = librosa.feature.spectral\_flatness(y=y)

tonnetz = librosa.feature.tonnetz(y=y, sr=sr)

data\_array[i,0] = np.mean(chroma\_cens)

data\_array[i,1] = np.var(chroma\_cens)

data\_array[i,2] = np.mean(spectral\_contrast)

data\_array[i,3] = np.var(spectral\_contrast)

data\_array[i,4] = np.mean(spectral\_flatness)

data\_array[i,5] = np.var(spectral\_flatness)

data\_array[i,6] = np.mean(tonnetz)

data\_array[i,7] = np.var(tonnetz)

# Set all values to zero for files with problems

except:

print(f'Problem file: {file\_path}')

for j in range(num\_new\_features):

data\_array[i, j] = 0

counter += 1

return data\_array

new\_features\_array = extract\_new\_features('Data/genres\_original')

1. **Adding the above extracted 4 features to the already selected 4 features which are present in the dataframe**

df['chroma\_cens\_mean'] = new\_features\_array[:,0]

df['chroma\_cens\_var'] = new\_features\_array[:,1]

df['spectral\_contrast\_mean'] = new\_features\_array[:,2]

df['spectral\_contrast\_var'] = new\_features\_array[:,3]

df['spectral\_flatness\_mean'] = new\_features\_array[:,4]

df['spectral\_flatness\_var'] = new\_features\_array[:,5]

df['tonnetz\_mean'] = new\_features\_array[:,6]

df['tonnetz\_var'] = new\_features\_array[:,7]

# Since there is only 1 row of missing value, now we'll just fill in the missing values of jazz.0054.wav as the mean of those of the jazz songs, which is at index 554

1. **Filtering out the jazz genre expect for jazz.0054**

for i in range(-8,0,1):

# Filter out the jazz genre except jazz.0054

df.iloc[554,i] = df[ df.label =='jazz'].iloc[np.r\_[np.arange(0,54),np.arange(55,100)],i].mean()

df.to\_csv('new\_csv',index=False)

# saving the index without the index column which comes by default

df = df.iloc[:,2:]

1. **EDA and Data Preprocessing**

corr = df.corr()

#Create a mask for the heatmap

mask = np.zeros\_like(corr)

mask[np.triu\_indices\_from(mask)] = True

plt.subplots(figsize=(20, 20))

sns.heatmap(corr, mask=mask, cmap="vlag")

# Most of the variables do not have a high correlation with other variables. Let's filter out the extremely highly correlated pairs and examine them.

sol = (corr.where(np.triu(np.ones(corr.shape), k=1).astype(np.bool))

.stack()

.sort\_values(ascending=False))

for index, value in sol.items():

if (value > 0.75) or (value < -0.75):

print(index, value)

# using this code we are able to extract the features which are correlated with each other and also by how much from the above map

1. **Splitting train and test data**

# Given the small number of training data, I set 90% as training data and 10% as testing data. For hyparameter tuning, given the small dataset, we will use the same train dataset to tune For the split of data, I make sure every class has the same number of data to train and test.

y = df.label

X = df

#Use `label` to split data evenly and drop `label` column after split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.1, stratify=df.label, random\_state=77)

X\_train.drop('label',axis=1,inplace=True)

X\_test.drop('label',axis=1,inplace=True)

1. **Normalize the data**

sc = StandardScaler()

X\_train\_scaled = sc.fit\_transform(X\_train)

X\_train = pd.DataFrame(X\_train\_scaled, index=X\_train.index, columns=X\_train.columns)

X\_test\_scaled = sc.transform(X\_test)

X\_test = pd.DataFrame(X\_test\_scaled, index=X\_test.index, columns=X\_test.columns)

1. **Initial model fitting and recursive feature elimination (cell 16 ask swaraj once!!!!!!)**

# Now we will fit our training data to xgboost classifier first, and then we’ll do RFECV to check which variables can be eliminated

# RFECV stands for Recursive Feature Elimination and Cross-Validation Selection which does Backward selection in Python: Scikit-Learn.

estimator = XGBClassifier(eval\_metric=’merror’)

rfecv = RFECV(estimator, step=1, cv=5,scoring=’accuracy’,verbose=1)

rfecv.fit(X\_train, y\_train)

# if step > 1 then step corresponds to the (integer) number of features to remove at each iteration.

1. **Checking which features can be further deleted.**

features\_drop\_array = list(np.where(rfecv.support\_ == False)[0])

X\_train.columns[features\_drop\_array]

X\_train.drop(X\_train.columns[features\_drop\_array], axis=1, inplace=True)

X\_test.drop(X\_test.columns[features\_drop\_array], axis=1, inplace=True)

# this code determines which columns can be eliminated and drops them, this happens in both train and test data

1. **Model training**

model = XGBClassifier(n\_estimators=1000)

model.fit(X\_train,y\_train,eval\_metric='merror')

# eval\_metric is the evaluation metric for validation data, a default metric will be assigned according to objective (rmse for regression, and logloss for classification) but here we use rmse -> root mean square error for better yielding results

1. **Getting the accuracy for both train and test data**

y\_pred\_train = model.predict(X\_train)

y\_pred\_test = model.predict(X\_test)

target\_names = sorted(set(y))

print(f'Training accuracy: {accuracy\_score(y\_train,y\_pred\_train)}')

print(f'Training:\n {classification\_report(y\_train, y\_pred\_train, labels=target\_names)}')

print(f'Testing accuracy: {accuracy\_score(y\_test,y\_pred\_test)}')

print(f'Testing:\n {classification\_report(y\_test, y\_pred\_test, labels=target\_names)}')

#Confusion matrix of the test data

cm = confusion\_matrix(y\_test, y\_pred\_test)

plt.figure(figsize = (16, 9))

sns.heatmap(cm,cmap="Blues", annot=True, xticklabels = target\_names, yticklabels = target\_names )

# this will display training accuracy as 0.99 while test data is 0.78 which indicates that the model has been overfit. Hence, we will add regularization parameters and tune other parameters to reduce this problem.

1. **Hyperparameter tuning**

# When creating a machine learning model, you'll be presented with design choices as to how to define your model architecture. Often times, we don't immediately know what the optimal model architecture should be for a given model, and thus we'd like to be able to explore a range of possibilities. In true machine learning fashion, we'll ideally ask the machine to perform this exploration and select the optimal model architecture automatically. Parameters which define the model architecture are referred to as hyperparameters and thus this process of searching for the ideal model architecture is referred to as hyperparameter tuning.

# In the following, we'll use hyperopt library to help tuning the parameters. The parameters may vary for each run

space={

'n\_estimators': hp.quniform('n\_estimators', 0,3000,1),

'reg\_lambda' : hp.quniform('reg\_lambda', 0,500,1),

}

def objective(space):

clf=XGBClassifier(

n\_estimators =int(space['n\_estimators']),

reg\_lambda = int(space['reg\_lambda']),

)

evaluation = [( X\_train, y\_train), ( X\_test, y\_test)]

clf.fit(X\_train, y\_train,

eval\_set=evaluation, eval\_metric="auc",

early\_stopping\_rounds=10,verbose=False)

pred = clf.predict(X\_test)

accuracy = accuracy\_score(y\_test, pred)

return {'loss': -accuracy, 'status': STATUS\_OK }

trials = Trials()

best\_hyperparams = fmin(fn = objective,

space = space,

algo = tpe.suggest,

max\_evals = 100,

trials = trials)

print(f"best params: {best\_hyperparams}")

# this will output the best n\_estimators and reg\_lambda values to be used for the model

1. **Running the model again after hyperparameter tuning**

model1 = XGBClassifier(n\_estimators=304, reg\_lambda=25)

model1.fit(X\_train,y\_train,eval\_metric='merror')

y\_pred\_test1 = model1.predict(X\_test)

print(f"accuracy: {accuracy\_score(y\_test,y\_pred\_test1)}")

print(f'New tuned model:\n {classification\_report(y\_test, y\_pred\_test1, labels=target\_names)}')

1. **Save the model and the preprocessing**

pickle.dump(sc, open('sc.pkl','wb'))

pickle.dump(model1, open('model.pkl', 'wb'))