Automobile Classification from Engine Sound using Learning Vector Quantization

A PROJECT REPORT

TEAM MEMBERS

SAHIL ASHVINKUMAR TRIVEDI	16BIT0098
SOMDYUTI DAS ADHIKARY	18BEE0112
AJEYA SHYAM BHATT	18BEE0233

Under the guidance of

PROF. MATHEW NOEL M.
SCHOOL OF ELECTRICAL ENGINEERING,
VELLORE INSTITUTE OF TECHNOLOGY,
VELLORE - 632014

ABSTRACT

Sound classification has become increasingly accurate in past decade. In the 1950s the general consensus among computer scientists was that sound waves needed to be split into discrete, phonetic ‘units’ which could be categorised and be used to classify sounds. The first ever speech classifier was called ‘Audrey’, developed by Bell Labs in 1952. It was built with analog electronic circuits and could recognise spoken numbers between 0 and 9. Later DARPA created a system called ‘Harpy’. It consisted of 15,000 interconnected nodes, and each node represented all the possible utterances in a domain. It could recognise complete sentences and was utilised brute force to map the speech to the nodes.

Deep Blue, developed by IBM in 1980, used Hidden Markov Models (HMMs) and obtained better results. HMMs represented utterances as states and probabilistically determined what the word was. Certain words when spoken for different durations have different meanings. HMMs display significant plasticity for context mapping because of their probabilistic approach. The current state of the art was achieved by Georffrey Hinton at the University of Toronto, where he used a Convolutional Neural Networks do leverage the abundance of data and computing power.

METHODOLOGY

Any of the techniques that use subword features will not be able to be used for non-speech sound identification. This is because environmental sounds lack the phonetic structure that speech does. There is no set “alphabet” that certain slices of non-speech sound can be split into, and therefore subword features (and the related techniques) cannot be used. Due to the lack of an environmental sound alphabet, all of the Hidden Markov Model (HMM) based techniques that are shown in the table cannot be used. Since HMM techniques are the main techniques now used in speech and speaker recognition, this leaves only a few other techniques.

An LVQ system is represented by prototypes which are defined in the feature space of observed data. In winner-take-all training algorithms one determines, for each data point, the prototype which is closest to the input according to a given distance measure. The position of this so-called winner prototype is then adapted, i.e. the winner is moved closer if it correctly classifies the data point or moved away if it classifies the data point incorrectly. An advantage of LVQ is that it creates prototypes that are easy to interpret for experts in the respective application domain. LVQ systems can be applied to multi-class classification problems in a natural way. It is used in a variety of practical applications.

Once the training converges we test the model on newly recorded samples to obtain the testing accuracy. Typical ways of regularization include adding some form of magnitude measurement of weights to the loss function. It is accepted that one of the main advantages of LVQ over ANN’s is its ability to correctly classify results even where classes are similar.

DATASET

There are several use cases where identifying the type of automobile using engine sound would be required, and these use cases affect the nature that has to be collected. Since this is a supervised model, we need to carefully pick the audio samples for further processing. We decided to collect audio data from manually by recording the stationary engine noise of household cars.

We collected a total of 180 audio samples per vehicle, each sample being 1 second long. These samples are converted from .aac to .wav

The data we collected has been uploaded on github:
https://github.com/sahil3Vedi/Engine-Sound-Classification/tree/master/Audio%20Files

IMPLEMENTATION

Using the .wav samples obtained from recording the engine sounds we can obtain Fourier Transforms. Each audio clip, 5 seconds in length, should have a corresponding spectrogram.

To generate spectrograms we utilise Fast Fourier Transforms (FFTs). This algorithm computes the Discrete Fourier Transform (DFT) of a sequence.This can be implemented on python.

STEP 1: Import Dependencies

```
from google.colab import drive
drive.mount('/content/drive')
import os
from pydub import AudioSegment
import matplotlib.pyplot as plt
from scipy.io import wavfile as wav
from scipy.fftpack import fft
import numpy as np
import pandas as pd

from sklearn.tree import DecisionTreeClassifier, export_graphviz
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn import model_selection, metrics
from sklearn.pipeline import Pipeline
from sklearn.metrics import roc_auc_score , classification_report, mean_squared_error, r2_score
from sklearn.metrics import precision_score, recall_score, accuracy_score, classification_report
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.feature_selection import *
from sklearn import metrics
from sklearn.utils import shuffle
from scipy.spatial import distance
```

STEP 2: Load Data Into The Project

```
DATA_DIR = 'drive/My Drive/Engine Data'
number_samples = 60
df_list = []
categories = []

for each_name in os.listdir(DATA_DIR):
  AUDIO_FILE = os.path.join(DATA_DIR,each_name)
  rate, data = wav.read(AUDIO_FILE)
  categories.append(each_name[:-4])

  for counter in range(number_samples):
    tempdict = {}
    newAudio = data[counter*500:(counter+1)*500]
    fft_out = np.abs(fft(newAudio), dtype = float)
    norm= np.linalg.norm(fft_out)
    fft_norm= fft_out/norm
    fft_norm1 = np.mean(fft_out)
    fft_norm2 = np.std(fft_out)
    fft_label = each_name[:-4]
    tempdict["FFT"] = fft_norm1
    tempdict["Norm"] = fft_norm2
    tempdict["label"] = fft_label
    df_list.append(tempdict)

df = pd.DataFrame(df_list)
df.head()
```

	FFT	Norm	label
0	1.001177e+06	5.416206e+05	Skoda_Yeti
1	1.284359e+08	1.042747e+08	Skoda_Yeti
2	3.069598e+08	2.412794e+08	Skoda_Yeti
3	4.867208e+08	4.421068e+08	Skoda_Yeti
4	6.626031e+08	4.884486e+08	Skoda_Yeti

STEP 3: Generate Training and Testing Set

```
df = shuffle(df)
df.reset_index(inplace=True, drop=True)

features=list(df.columns)
features.remove('label')
X = df[features]
y = df['label']

# split dataset to 60% training and 40% testing
X_train, X_test, y_train, y_test = model_selection.train_test_split(X,y, test_size=0.4,
random_state=0)
```

STEP 4: Model Implementation

```
#obtaining initial weights
weights = []
category_names = []
temp_categories = categories
for i, j in df.iterrows():
if(len(temp_categories)>0):
    category_label = j['label']
    if category_label in temp_categories:
      temp_categories.remove(category_label)
      sample_FFT = j['FFT']
      sample_Norm = j['Norm']
      temp_weight = [sample_FFT,sample_Norm]
      weights.append(temp_weight)
      category_names.append(category_label)
  else:
    continue

print(weights)
```

STEP 5: Training

```
#training the weight vectors iteratively
total_samples = (len(category_names)*number_samples)-len(category_names)
training_sample_max_index = int(np.floor(total_samples*0.6))
learning_rate = 0.1
epochs = 50
for epoch_counter in range(epochs):
  for k in range(0, training_sample_max_index):
    selected_sample = df.loc[k]
    samp_FFT = selected_sample['FFT']
    samp_Norm = selected_sample['Norm']
    samp_label = selected_sample['label']
    sample_weight = [samp_FFT,samp_Norm]
    distances = []
    for each_weight in weights:
      eucl_dist = distance.euclidean(each_weight, sample_weight)
      distances.append(eucl_dist)
    min_distance = min(distances)
    sample_min_dist_index = distances.index(min_distance)
    weight_index = category_names.index(samp_label)
    W = weights[weight_index]
    Q = np.subtract(sample_weight,W)
    R = np.multiply(learning_rate,Q)
    if(sample_min_dist_index==weight_index):
    weights[weight_index] = np.add(W,R) else:
      weights[weight_index] = np.subtract(W,R)

print(weights)
```

OUTPUT

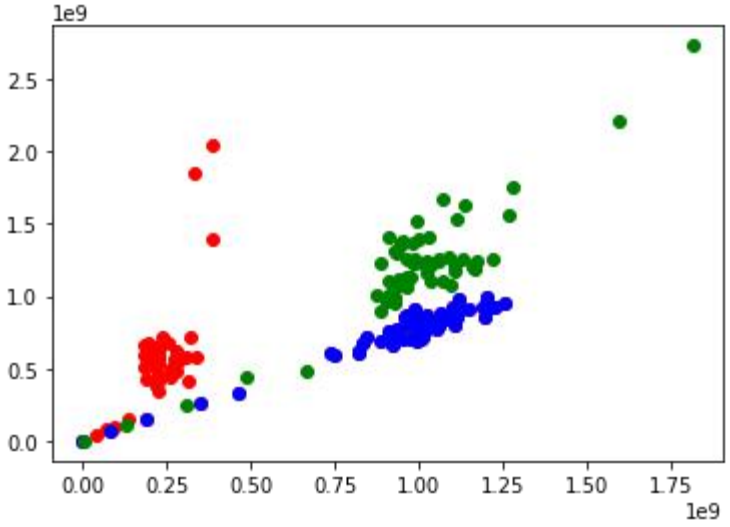

Fig: A Plot Diagram of samples from different categories

CONCLUSION

In the research reported in this paper, we studied the efficacy of the LVQ algorithm in classifying engine noises from household cars. We note that the accuracy rate of LVQ is only satisfactory and there is still a considerable amount of tweaking that the model requires before it is implementable for any of its use cases (toll plazas, car parkings, etc). One prominent restriction is collecting high quality audio samples of engine noise from regular street cars. To automate it is difficult and to do so manually is time intensive. This research suggests that while the LVQ algorithm has a lot of potential for classifying time based signals there is a long way to go before State of the Art (SotA) automobile engine noise classification is achieved using this technique.