AUDIO CLASSIFICATION MODEL

-By Anjali Yadav

1. Methodology (Steps) used

Data Collection: The first step in building a binary classification model for dog and cat audio signals is to collect a data set of audio files. This is done by using a pre-existing dataset **"Audio Cats and Dogs"** dataset from Kaggle. It is important to ensure that the dataset is representative of the problem you are trying to solve and that there is enough data to build a robust model.

Data preprocessing: Once you have a dataset of audio files, the next step is to preprocess the data to extract useful features that can be used as input to a MLP (Multilayered Perceptron) model. This usually involves converting the raw audio files into a format that can be fed into a MSP, such as spectrograms or Mel frequency cepstral coefficients (MFCC). In this model, I have used MFCC for feature extraction along with PSD and Spectral Centroid.

Before feature extraction, the audio signal files present with the .wav extension are digitized using the Librosa library and converted into waveforms. These waveforms can then be used to plot Spectrogram of each audio file. These waveform data is then normalized and resampling is done. The output of this stage is then used for feature extraction. The output is a feature matrix.

Data Splitting: After preprocessing, the output feature is encoded into 0 and 1, 1 being cat and 0 being dog. This data is then divided into dependent and independent features. These are then split into Train and test set using stratified method.

Modeling: After the data is preprocessed and augmented, the next step is to create a MSP model. Since we have already obtained a feature matrix, we will be directly feeding the Feature matrix into the fully-connected layers and the output of the last layer will be passes through sigmoid function to generate class output.
Popular libraries for creating MSPs in Python include Keras and TensorFlow.

Model Training: After building a CNN model, the next step is to train it on the preprocessed data using appropriate loss functions and optimization algorithms. Common loss functions for binary classification include **binary cross-entropy** and categorical cross-entropy. I have binary cross entropy for my model. The optimization algorithm used to train the model can be selected from a range of options, such as Stochastic Gradient Descent (SGD) or **Adam**. For simple binary classification models, Adam is preferred.

Model evaluation: Once the model is trained, the next step is to evaluate its performance on the held validation set. It involves calculating metrics such as precision, accuracy, recall, and F1 score to determine the model's ability to classify an audio file as a dog or a cat. The model can be further evaluated on the test set to assess its generalization performance.

2. Packages used

a. **Os:**
   It is a python built-in library which helps us to interact with our operating system. In our project, we have used ‘os’ library to accessing and manipulating files and directories.
   Some functions we used are os.path.join() and os.listdir().

b. **Pandas:**
It is a python library used for manipulating datasets. It helps us perform various operations on these datasets like data cleaning, transformation, and analysis. In this project, we used pandas to create a DataFrame containing waveform data of our audio signals.

c. **Matplotlib:**
This library is used for plotting and visualization. We have used Matplotlib to create Histograms to check whether our dataset is balanced or not.

d. **Numpy:**
It is a python library for numerical purposes and allows us to manipulate and access arrays and matrices. Some functions we used in our array are np.array(), np.mean(), etc.

e. **Plotly:**
It is a python library used for interactive visualizations, I have used this library to plot bar plots and gain information about the exact counts of each output class from the graph which was not provided by matplotlib library.

f. **Librosa:**
It is a Python library for analyzing and processing audio signals. It is specifically designed for music and speech processing, and provides a wide range of functions for tasks such as loading and saving audio files, extracting features from audio signals, and visualizing audio data. I used Librosa library to perform various tasks such as Librosa.load() for converting my audio signals to waveform data

g. **Scikit-learn:**
It helps to build machine learning models, including algorithms for classification, regression, clustering, and dimensionality reduction, as well as tools for data preprocessing, model selection, and evaluation. We used this library for various functions. We used 'LabelEncoder' class for encoding our output classes into 0 and 1. We also used 'train_test_split' to split our training data into train set and validation set.

h. **TensorFlow:**
TensorFlow is a low-level framework for building and training machine learning models, developed by Google. It provides a wide range of tools and functions for building and training different types of neural networks, including convolutional neural networks (CNNs), recurrent neural networks (RNNs), and deep neural networks (DNNs). In this project, we use it for MSP model.

i. **Keras:**
Keras provides a wide range of building blocks for building neural networks, such as layers, activations, loss functions, and optimizers, which can be combined to build complex models.

3. Data Pre-Processing implemented

Loading the data:

Loading audio data is the first step in the data preprocessing pipeline. Librosa is a Python package for audio signal analysis and processing, which provides useful tools for loading and processing audio data. When loading data with Librosa, the audio data is usually loaded as a time series waveform representing the amplitude of the sound wave at each instant. I loaded the data from the "Audio Cats and Dogs" dataset from Kaggle and unloaded the dataset into the waveform format by using Librosa.load().

Normalization:

Normalization is an important step in the data preprocessing pipeline because it ensures that the amplitude of sound waves remains the same across all samples. This is important because it ensures that the machine learning model does not learn from differences in hardness between samples. Various normalization methods can be used, but the most common is to scale the signal amplitude so that it has a maximum value of 1. I performed normalization to bring all the audio signals to a consistent level before resampling. Normalization is done by:

normalization_factor = desired_max_amplitude / max_amplitude

This normalization factor is calculated for each sample and then multiplied with the corresponding sample.

Resampling:

Resampling is changing the sampling rate of an audio signal. There are many reasons for this, such as reducing the data size or changing the frequency resolution of the signal. Resampling is changing the number of samples in a signal without changing the amplitude or shape of the signal. A lower sampling rate can decrease processing cost and time but it may distort the quality of our audio and hence decreasing the ultimate accuracy of our model.

Our dataset had sample rate of “22050” which is standard. We have tried moving the sample rate a little high to “24000” as our dataset isn’t complex.

Feature Extraction:

Audio signals can be analyzed by extracting relevant features such as pitch, loudness, and spectral content. Feature extraction can help to identify patterns and structures in the data. As we know each method of feature extraction captures different aspects, so I have used three methods to better caption multiple aspects of characteristics of my features.

MFCC for capturing information about the spectral envelope of the audio signal. PSD is a measure of the distribution of power across the frequency spectrum of an audio signal. Spectral Centroid can be used to identify the dominant frequency range of the signal and to distinguish between different types of sounds based on their spectral characteristics.

Encoding:

Deep learning models are typically designed to take numerical inputs and outputs, and they require the output classes to be represented as numeric values. This is because deep learning models use optimization algorithms to minimize a loss function, which is computed based on the difference between the predicted output and the ground truth output. Therefore, I performed encoding using the LabelEncoder class from scikit-learn.

4. Code Explanation

I have divided my code into 6 sections:

- Importing the Dataset
- Data preprocessing
- Splitting the dataset

- Modeling and Training
- Evaluation of the Model and Accuracy Visualization

In the first, **"Importing the dataset",** I have defined the paths for the train folder which contains the dataset of audio signals of cats and dogs in separate folders. I have also imported several libraries which will be used for accessing and manipulating this dataset. These files are in .wav format.

Then, in **"Data Preprocessing",** I have created 3 empty lists called audio_file, file_names and file_path to store various data about each audio file. "audio_file" contains the waveform data of each file, "file_names" contains the respective name of each file in .wav format. "file_path" contains the path of each audio file which trace back to the folder stored in my Drive. After I stored all the information in the respective lists, I have converted each of them into a single DataFrame which now contains 3 features. Then, I have used the feature "file_names" to extract the labels and created another feature column in our DataFrame. This feature will be our output Feature and it contains two classes namely 'Cat' and 'Dog'.

I then created a Spectrogram to visually represent the frequency content of my sound signals over time. Next, I performed normalization on my data. This was does by simply define the formula:

normalization_factor = desired_max_amplitude / max_amplitude

and then multiplying each waveform data sample with the normalization_factor. I defined my desired_max_amplitude as 1, which is the value used in most cases. I also observed that max_amplitude of my samples was either 1 or very close to 1 in most of the cases. This shows that the frequencies across the dataset were already very much consistent.

After Normalization, I increased the sampling rate of my data from 22050 to 24000 Hz. This will result in a higher frequency range being captured in the audio signal, a larger file size, and potentially improved audio quality.

For feature extraction, I first defined the window size and hop length. The choice was made according to the sample rate of the data. A default value is 25 ms and 10 ms for a sr of 40 Hz.

Then we created a feature_matrix to store all the feature vectors corrosponsing to each feature. For Feature extraction, I used MFCCS (Mel Frequency Cepstral Coefficients), PSD (Power Spectral Density), and Spectral Centroid. Using these three methods together instead of just one can increase feature dimensionality, create a robust to noise model and capture multiple aspects of features more efficiently. Using all three instead of just mfcc can give our model a beter discriminating power and ,make it more robust to noise.

Also, since we are applying deep learning models, we will need to convert our output labels into numeric values since loss functions calculate difference in predicted and actual output probabilities. Since our output variable do not have any heirarchial relationship, I will use LabelEncoder for encoding.

Next, we **"Split the dataset"** to train the model. Before splitting the dataset I checked if we have unbalanced dataset. As we see, the dataset is slightly unbalanced, using a Stratified method for splitting the data will be a better fit as it will divide equal parts of both class labels into the training and testing datasets.

For **"Building the model",** I use TensorFlow and Keras libraries. I built a three layered fully connected network and the output of the last layer of this network was fed into Sigmoid activation function. Sigmoid was preferred as our task was to perform binary classification between cats and dogs sounds. I used other non-linear functions at each layer of our fully-connected network. The choice of this non-linear function was made to be 'ReLU' to prevent any vanishing gradient problem. Also, at the end of each layer, a dropout layer was built which added a given amount of error before passing on the data to the next layer. This was done to prevent overfitting.

After modeling was done, I compiled my model. I used 'Adam' as the optimizer because it prevents being stuck in local optima and is relatively easy to use. It requires very little hypertuning. Binary_crossentropy was chosen as our loss function because, as the name suggests, it's main goal is to predict binary labels. It also is very effected when we have unbalanced data.

As the dataset was not balanced, We could not use Accuracy as our performance metric, and F1-score and Binary Accuracy were the best choice for this task. I chose Binary Accuracy as it was designed for binary class predictions. It doesn't get much affected by class imbalance as it takes into account both true positives and true negatives.

Then our model is trained on the training set and evaluated on the validation set after each epoch. The goal is to find the best set of hyperparameters that minimize the loss on the validation set while avoiding overfitting to the training set. I chose the number of epochs to be 100 as our dataset isn't very big and task at hand is quite simple.

Next, we **"Visualize and Evaluate our model".**

5. Results

We got a Training accuracy of **96.27%**

And testing Accuracy of **97.01%**