Speech Recognition (Report)

Name: Doaa Samir **ID:** 2103114 **Department:** AI

Importing the necessary libraries

There are different tools and libraries that I used in the code, which are used for handling data, performing calculations, and visualizing results like:

os: Helps with interacting with the operating system, such as managing files and directories.

numpy: Provides support for handling numerical data and performing mathematical operations efficiently.

seaborn: A tool for creating visually appealing charts and graphs to analyze and understand data patterns.

pandas: Used for organizing and manipulating structured data, such as spreadsheets or tables.

librosa: A library specifically designed for analyzing audio, which helps with tasks like extracting features from sound files.

matplotlib.pyplot: A tool for creating a variety of plots and visualizations to display data in a graphical format.

tensorflow.keras: A library used for building and training machine learning models. It provides tools for creating and optimizing models for tasks like classification or prediction.

glob: Helps in finding and managing files and directories based on specific patterns.

sklearn.preprocessing: Contains tools for preparing and transforming data before using it in machine learning models, such as scaling features and encoding categorical values.

sklearn.metrics: Provides methods to evaluate the performance of machine learning models by comparing predictions to actual results.

sklearn.model_selection: Offers tools for splitting data into training and testing sets, which is essential for evaluating how well a model performs on new, unseen data.

I start by defining where the audio files for both training and testing are stored. These are organized into separate directories, one for training and another for testing. Each directory contains subfolders for 'cat' and 'dog' sounds. Then prepare two lists: one to hold the labels (either 'cat' or 'dog'), and another to hold the file paths (locations of the audio files). These lists will help us keep track of what each audio file represents and where it is stored.

A function is used to process each directory. It scans through the files in the 'cat' and 'dog' folders within both the training and testing directories. For each file, the function records its label (either 'cat' or 'dog') and its full file path. This ensures that we can easily access and identify each audio file later.

After processing all the files, the collected labels and file paths are combined into a table (DataFrame). This table provides a structured way to view and analyze the data. Then we get to look at both the first few and the last few rows of the DataFrame to get an initial understanding of how the data is organized.

A summary of the DataFrame is generated to provide an overview of the data. This includes details like the total number of entries, unique labels, and any missing data points. Finally, we will count the number of instances for each label (cat and dog). This helps us understand the balance of our dataset—whether we have an equal or unequal number of cat and dog sounds, which can impact model training.

The processed dataset was exported and saved as a CSV file named ‘train_test.csv’ in the Downloads folder. The data was saved without including any additional index column, ensuring that only the relevant information was stored.

Data Visualization

First, the dataset was carefully filtered to remove any entries where the label was marked as 'Unknown'. This step ensured that the analysis only included data clearly identified as either "cat" or "dog."

A bar chart was created to display the number of entries labeled as "cat" compared to those labeled as "dog." This visual representation allowed us to easily compare the quantities of data available for each category. The x-axis of the chart represents the two categories: "Cat" and "Dog." The y-axis shows the count of how many entries are labeled as "cat" or "dog."

There are 3 types of visualization: Waveform, Spectrogram, Mel Spectrogram.

- **Waveform Visualization:** The first technique involves creating a **waveform plot**. A waveform is a visual representation of the audio signal, showing how the loudness (or amplitude) of the sound changes over time. This visualization allows us to see the overall shape of the sound, such as when it becomes louder or quieter, and can give us an idea of the audio's dynamics.
- **Spectrogram Visualization:** A spectrogram provides a detailed view of the audio's frequency content over time. It breaks the sound into short segments and analyzes which pitches (or frequencies) are present in each segment. This method allows us to see how the different frequencies in the sound vary as the audio progresses. The spectrogram is particularly useful for identifying patterns, such as the presence of specific notes in music or the features of speech.
- **Mel Spectrogram Visualization**: It is a refined version of the standard spectrogram. The Mel spectrogram adjusts the frequency scale to better match human hearing, emphasizing the frequencies we are more sensitive to and downplaying those we hear less well. This visualization makes it easier to understand how the sound would be perceived by a human listener, focusing on the most relevant parts of the audio.

We began by selecting the second audio file from our dataset. This file contains the sound labeled as "cat." The chosen audio file was loaded into our system for further analysis. During this process, the audio data was prepared along with information about how it was recorded (specifically, the rate at which the audio was sampled).

For the Waveform plot I created a visual representation of the audio's waveform. This plot shows how the sound waves (volume changes) vary over time, providing insight into the structure and dynamics of the sound. Next, I generated a spectrogram, which is a visual representation showing how the different frequencies in the audio change over time. This helps us understand the frequency content and how it evolves throughout the audio clip. Then I made a Mel spectrogram. This is a more refined version of the spectrogram, adjusted to align with how humans perceive sound. It emphasizes the frequencies that our ears are most sensitive to, making it particularly useful for analyzing speech and other audio signals.

Data Augumentation

To improve the robustness and accuracy of an audio classification model by applying various data augmentation techniques:

Noise: It is used to simulate real-world conditions where audio signals may contain background noise. This technique makes the model more resilient to variations in audio quality. Random noise is added to the original audio signal. This noise is subtle, ensuring the main characteristics of the audio are preserved while introducing slight variations.

Time Stretching: It alters the speed of the audio without changing its pitch. This method helps the model recognize sounds at different speeds, making it more versatile in handling variations in speaking or playing speeds. The audio is either slowed down or sped up by a specific rate. For instance, if the audio is stretched, it will play slower, allowing the model to learn from different tempo variations.

Time Shifting: Time shifting involves shifting the audio forward or backward slightly. This simulates the effect of misalignment or delay in starting or stopping the recording, helping the model learn to recognize sounds even when they appear at different times. The audio data is shifted by a random amount, effectively moving the start of the sound earlier or later in the recording.

Feature Extraction

I use specialized techniques to extract important features from audio recordings. These features help us understand the unique characteristics of the sound. First I extract two key audio features: MFCC (Mel-Frequency Cepstral Coefficients) and the Mel Spectrogram from the audio data. These features help capture the characteristics of the sound, such as pitch and tone, in a way that a model can understand.

I load a small segment (2.5 seconds) of the audio file. This segment is used as the base for extracting features. To make the model strong, the code creates a noisy version of the original audio. It then extracts features from this noisy version, helping the model learn to recognize the sound even when there is background noise. It creates a time-stretched version of the audio, which means it slows down the audio slightly. This helps the model learn to identify the sound even when it's played at a different speed.

All the extracted features (from the original, noisy, and time-stretched audio) are combined into a final set of data. This combined data is what will be fed into the model for training.

The function get_features starts by loading an audio file from a specified location (path). If a sample rate (the speed at which the audio is sampled) is provided, it uses that rate; otherwise, it uses the default sample rate. The core feature being extracted is called Mel-Frequency Cepstral Coefficients (MFCCs)**.** These are numerical values that capture important characteristics of the audio signal, focusing on how different frequencies contribute to the sound. In this function, it calculates 13 different MFCCs.

After extracting the MFCCs, the function processes these values by taking the average of each coefficient across the entire audio clip. This results in a single set of 13 values that represent the overall audio characteristics. Finally, the function returns this processed set of features. These values can then be used for further analysis or as input for model.

Two empty lists, X and Y, are created. X will hold the features extracted from each audio file, while Y will store the labels associated with those features. The program goes through each audio file listed in the data, along with its label. For each file, it extracts important characteristics (features) that represent the audio content. As each audio file is processed, the program provides updates every 500 files, showing the number of files that have been processed so far. For each extracted feature, the program adds the feature to the X list and the corresponding label to the Y list. This creates a complete set of features and labels for all audio files. After processing all the audio files, the program confirms that all files have been processed and provides a final update.

Data Preparation

The dataset named Features is divided into two separate components. The first component, represented by X, includes all the columns of the dataset except for the last one. This subset contains the features or input variables used for analysis. The second component, represented by Y, consists of only the values from the column labeled 'labels', which indicates the target variable or the output we are trying to predict. Essentially, X holds the input data, while Y holds the corresponding labels or categories for each data entry.

Splitting the Dataset

I apply a technique called one-hot encoding. This involves transforming each categorical label into a vector of numbers where only one position is marked as 1, and all other positions are 0. For example, if we have three categories, the labels will be converted into three-dimensional vectors where only one of

the dimensions is 1 depending on the category, and the rest are 0. This transformation makes it easier for machine learning models to understand and process the categorical data.

The dataset has been divided into training and testing sets using a process called "train-test split." Specifically, 80% of the data is used for training the model, while 20% is reserved for testing its performance. This division helps ensure that the model is trained on one part of the data and evaluated on a separate part to assess how well it generalizes to new, unseen data. The shapes of the resulting data sets are then displayed, showing the number of samples and features in the training and testing sets.

A StandardScaler is used to normalize the data. First, the StandardScaler is applied to the training data to adjust its scale so that the features have a mean of 0 and a standard deviation of 1. This process helps in improving the performance and stability of machine learning models. The same scaling is then applied to the test data to ensure consistency, meaning that the test data is scaled in the same way as the training data. The shapes of the training and test datasets are then printed out, showing how many samples and features are in each dataset.

The input data x_train and x_test are modified to include an additional dimension. This is done using the np.expand_dims function, which adds a new axis to each dataset, changing their shape. Specifically, a new dimension is added along axis 2. This adjustment is often necessary when preparing data for models that expect a certain input shape, such as convolutional neural networks. After this adjustment, the shapes of the training and test datasets are displayed, showing how the data has been transformed to meet the requirements of the model.

MLP Model

In the model of this MLP model, a sequential architecture was employed, consisting of multiple fully connected (dense) layers. The model begins with an input layer that corresponds to the features of the training data. Following this, there are several dense layers with decreasing numbers of neurons (512,

256, 128, and 64, respectively). Each of these layers uses the ReLU (Rectified Linear Unit) activation function to introduce non-linearity, which helps the model capture complex patterns in the data.

To improve the model's training stability and prevent overfitting, each dense layer is followed by batch normalization, which normalizes the outputs of the previous layer, and dropout, which randomly sets a portion of the neurons to zero during each training step. This dropout is applied with a 50% probability, ensuring that the model does not rely too heavily on any single neuron, promoting more generalized learning.

The model concludes with an output layer that uses the softmax activation function, suitable for multi-class classification tasks. The model is compiled using the Adam optimizer with a very low learning rate of 0.00001, aiming for slow and steady convergence during training. The loss function used is categorical crossentropy, appropriate for classification tasks, and accuracy is monitored as the performance metric.

Model Evaluation

We used to evaluate the performance of a trained model. Specifically, it calculates how accurately the model predicts the correct labels on a set of test data, which it hasn't seen before. The result is expressed as a percentage, indicating the model's accuracy. The higher the percentage, the better the model is at making correct predictions. This accuracy score is a key metric to assess how well the model is likely to perform on new, unseen data.

The trained machine learning model is used to make predictions on the test data (x_test). These predictions are initially in a numerical format that represents the model's output. Next, the predicted numerical values are converted back into their original categorical form using a method called inverse_transform. This step is important because it translates the model's numeric predictions into labels that are understandable and meaningful (e.g., class names or categories). Finally, the true labels for the test data (y_test) are also converted back into their original categorical form using the same

method, allowing for a direct comparison between the predicted labels and the actual labels to evaluate the model's performance.

Confusion Matrix

We analyze the performance of our model by using a confusion matrix to compare the predicted labels with the actual labels from the test data. The confusion matrix is a table that shows the number of correct and incorrect predictions made by the model, with each row representing the actual class and each column representing the predicted class.

To make this comparison clear and easy to interpret, we visualize the confusion matrix as a heatmap. This heatmap uses different shades of blue to represent the number of instances for each combination of predicted and actual labels, with darker shades indicating higher counts. The matrix also includes labels on both axes to indicate which classes are being compared.