Bird Sound Classification using Support Vector Machines

Problem Formulation

At the onset of this project, our primary objective is to classify audio recordings of bird sounds from different species using supervised machine learning, specifically employing Support Vector Machines (SVM). This classification problem utilizes audio recordings as data points, featuring five common birds in Helsinki: Hooded Crow, Whooper Swan, Barnacle Goose, Oriental Cuckoo, and Eurasian Penduline Tit. Each bird species is represented by approximately 300 seconds of combined sound clips, forming our dataset.

These recordings, along with their corresponding species names, are obtained from the Xeno-canto database via GBIF [1]. The audio files come in different formats but all are converted to WAV files with a 44.1 kHz sample rate. To transform this raw audio data into a format suitable for machine learning, we extract Mel-frequency cepstral coefficients (MFCC) ast the features. MFCC captures the spectral envelope of sound whereby our multidimensional features are simplified to concise, segmented parts which could be trained for classification. In this supervised learning setup, the name of each species serves as the label for the corresponding audio samples.

Given that these audios were recorded in natural environments, background noises are inevitable, sometimes even containing sounds of other bird species. To mitigate these effects, only high-quality sound clips are selected. Furthermore, we employ a triple threshold segmentation method based on short-time energy and zero-crossing rate to isolate bird vocalizations similar to the segmentation method used in this paper[5]. This approach uses three thresholds: T1 and T2 for identifying high-energy segments by calculating short-time energy in a small time window, T3 for refining the segment boundary based on zero-crossing rate. This segmentation technique effectively removes noisy data, enhancing the quality of our input data for subsequent feature extraction through segmentation revolving around T1,T2,T3 threshold criterion.[Appendix B]

Methodology: Feature Selection

Mel-frequency cepstral coefficients(MFCC)essentially break down the audio signal into its basic components, similar to how we might represent a musical note in terms of its pitch and tone. They focus on important frequencies in audio by mapping them to the Mel scale, which reflects how the human ear perceives sound. To put it simple, it is an array of numbers that represents the frequency and amplitude of an audio sample.

MFCC is chosen as our feature representation due to its relatively strong robustness to noise while concurrently reducing dimensionality for ease of computation as many already established libraries demonstrate with a similar implementation [2]. This feature selection process is crucial in our supervised learning approach, as it directly influences the input space of our SVM model.

The multidimensional nature of our input data is inherent in its time-frequency representation. Each audio sample can be viewed as a spectrogram or MFCC matrix[Appendix D], where rows represent short time frames (e.g., 10ms segments) and columns represent frequency bins or MFCC coefficients, whereby we deliberately select a combination of {n_mfcc, n_fft, hop_length, n_mels}[6] for fine-tuning the input variables to achieve the best results for model training. By looping through different values of parameters, we identified optimal MFCC parameters (n_mfcc=20, n_fft=128, hop_length=64, n_mels=64) that maximized our model's performance. More precisely, there are 20 MFCC coefficients for each sample of the recording. For a multi-sample recording, we take the mean and standard deviation value for each coefficients in all the samples, yielding a 40 - dimension feature for each data point, thereby creating a matrix of data for each audio sample, capturing both temporal and spectral characteristics of the bird sound [Appendix C].

Model Selection: Support Vector Machines

We utilized a Support Vector Machines for the classification of bird sounds. SVMs are particularly suitable for our audio classification task due to their efficacy in high-dimensional spaces. For this project, we opt for a multi-class SVM approach using the one-vs-one strategy, whereby we train multiple binary classifiers, each distinguishing one class from the rest.

Within our multidimensional problem with K=5 classes (bird species), we train K(K−1) / 2 binary SVM classifiers. Each classifier is trained to distinguish between two specific classes, ignoring the other K - 2 classes.During the classification of a new instance, each of the binary classifiers predicts a class. The final prediction is made using a voting mechanism, where each classifier "votes" for one of the two classes it was trained on. The class with the most votes across all classifiers is selected as the final prediction for the instance. The decision function for the k-th classifier is formulated as [3]:

$$f_k(x) = w_k^T \phi(x) + b_k$$

Where phi (x) represents the feature map induced by the kernel function. We employ a Radial Basis Function (RBF) kernel, which is adept at reducing complex dimensionalities and encompass nonlinear behaviors within the feature space and allow it to map back to the kernel, which is most appropriate for us to examine intricacies in audio data.

In our SVM implementation, the kernel choice significantly impacts model performance. We opted for the Radial Basis Function (RBF) kernel, which measures similarity between data points and allows for non-linear decision boundaries. This characteristic is crucial for our bird sound classification task, where audio feature relationships are often non-linear. The RBF kernel's ability to implicitly map data to a higher-dimensional space enables the capture of complex patterns in bird vocalizations that a linear kernel might overlook. The fine tuning process reflects this fact as well: we utilize grid search with 5 fold cross-validation, identified optimal hyperparameters (C=10, kernel='rbf', gamma=0.01, degree=2, class_weight='balanced') from different combination of parameter values. Experimentally, the RBF kernel with a gamma value of 0.01 yielded a test accuracy of 98.68% compared to the linear model with an accuracy of 96.98%[Appendix E]. Moreover, our 5-fold cross-validation process for the fine tuned kernel, RBF produced a mean validation score of 98.3% from the grid search[Appendix E]. closely aligning with the test set performance. Such consistency across multiple data subsets underscores the robustness of our model.The essence of our SVM implementation revolves around the squared hinge loss function, defined mathematically as [3]:

$$L = \sum_{i=1}^{n} \max(0, 1 - y_i f(x_i))^2$$

For average loss we get:

$$L = \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i f(x_i))^2$$

Where y represents the true label (either +1 or -1) and f(x) denotes the predicted score from the SVM model, L is the total loss, n is the number of samples trained and the max function is the squared hinge loss for a single instance[3]. This loss function is pivotal in our approach for several reasons. It facilitates margin maximization, encouraging a larger separation between classes, which often leads to superior generalization on unseen data. Furthermore, the method promotes sparsity by focusing solely on support vectors, thereby reducing computational complexity. With the two benefits aforementioned, the method is robust to outliers even at the face of noisy audio data. Finally, the squared term provides a gradual, smoother gradient compared to standard hinge loss, thereby reducing over-fitting, and enhances stability in our optimization process.

Random Forest

To provide a broader perspective on our classification task, we also implemented a Random Forest classifier. Random Forest is a popular ensemble learning method known for its robustness and ability to handle high-dimensional data. By comparing SVM with Random Forest, further insight into how different types of classifiers perform on our bird sound dataset could be achieved.

Gini impurity to evaluate the quality of splits when constructing decision trees. Equation wise, the Gini impurity is expressed as [3]:

$$I_G(p) = \sum_{i=1}^{J} p_i(1 - p_i) = 1 - \sum_{i=1}^{J} p_i^2$$

Where $I_{G(p)}$ is the Gini impurity, J is the number of classes, and p_i is the proportion of items in class i [3]. This measure quantifies the probability of misclassification for a randomly chosen element. The algorithm selects splits that minimize the weighted average of Gini impurities in the resulting subsets, effectively maximizing information gain at each node. Such a process ensures that the Random Forest creates decision trees that optimally separate the classes based on the available features.

Our Random Forest implementation also underwent rigorous hyperparameter tuning using grid search with 5-fold cross-validation. The basic default model has a test accuracy of 97.17%. Whereas the fine tuned configuration (100 estimators, 'sqrt' for max features, min samples split of 10, min samples leaf of 1, and 'balanced_subsample' for class weight) achieved a cross-validation mean accuracy of 97.92% and a test set accuracy of 98.3% [Appendix F].

Final Comparison

While both SVM and Random Forest excelled in our bird sound classification task, they differ fundamentally in their approach. SVM seeks an optimal hyperplane to maximize class separation, whereas Random Forest aggregates multiple decision trees, reducing overfitting and handling nonlinear relationships differently, yet effectively. The fine tuned SVM model slightly outperformed Random Forest (98.68% vs. 98.3% test accuracy and 98.3% vs. 97.9% mean validation score) and is chosen as the final method of this project. However, it is worth noting that the SVM model with RBF kernel demonstrated sensitivity to hyperparameter tuning despite the Random Forest demonstrated strong performance even with default parameters.

This comparison highlights important considerations for model selection. While our tuned SVM achieved the highest accuracy, Random Forest's robust intrinsic performance makes it a practical choice for larger datasets or scenarios with limited computational resources. Our results emphasize the need to balance raw performance metrics with practicality and scalability when choosing a classification approach for bird sound analysis if the study is further enhanced.

Comparison to Literature Values and Analysis of Limitations:

Interestingly, our results diverge significantly from those reported in the literature. For instance, Xie et al. [4] report much lower accuracies for SVM (64.51%) and Random Forest (85.71%) in their bird sound classification task. They also present results for a more complex model, MDF-Net, which achieved 97.29% accuracy. The discrepancy, however, raises concerns for some limitations in our project such as: bias of the dataset, and inherent segment size differences that stem from recording environment constraints.

The discrepancy also sheds light on the delicate balance in the "bias-variance trade-off". Our relatively simple SVM implementation achieved surprisingly high accuracy, which could potentially indicate a risk of overfitting to our specific dataset. Whereas more complex models like MDF-Net in the literature achieve high accuracy by leveraging intricate architectures and multiple feature types. These differences could stem from various factors such as dataset characteristics, feature extraction methods, and

experimental setup. Most notably, there are inherent limitations in the setup of our system. The class imbalance in our dataset, with support values ranging from 21 to 157 samples per class[Appendix E,F], may have biased our model towards majority classes which is possibly the reason why the class 3 Whooper Swan has the lowest recall score of 0.92, contributing most to the misclassification. Additionally, our selection of high-quality audio clips, while conducive for feature extraction, might have created an environment bias that doesn't reflect real-world constraints where bird sounds are often mixed with various noises.

A balanced approach would involve controlled model training and validation, perhaps using ensemble methods to mix and match the best models appropriate for the task and acquiring a more balanced dataset. Through the iterative process of model validation, one could observe the optimum balance of theory and pragmatic approaches. The contrasting results between our implementation and the literature emphasize the importance of context in model selection. While our SVM model accuracy sails through our specific dataset, the literature suggests that for different datasets or in broader applications, more complex models like MDF-Net or ensemble methods might be necessary to achieve comparable performance.

Thus, the high accuracy achieved by our SVM model with RBF kernel in bird sound classification, compared to Random Forest and other approaches in the literature, gives insight to both limitation and potential for our method. For the limited scope of our project it is evident in its overall F1 score of 0.99, demonstrating both high precision and recall across bird species. However, one should also consider the difficulty in segmentation and preprocessing to train such a model. The model achieved perfect classification for most species. While the Whooper Swan posed a slight challenge with a few misclassifications, the overall performance remains optimal for the scope of our research. These results highlight our model's capability to delineate signature bird acoustics within our specific frame of 5 species. Future research could focus on fine-tuning the model for species such as the Whooper Swan, potentially through advanced feature extraction techniques or targeted ensemble methods in order to hone the model better suited for scaling.

References

[1] W. Vellinga, "Xeno-canto - Bird sounds from around the world," Xeno-canto Foundation for Nature Sounds, 2024. Occurrence dataset https://doi.org/10.15468/qv0ksn accessed via GBIF.org on 2024-09-18.

[2] B. McFee et al., "librosa: Audio and Music Signal Analysis in Python," in Proceedings of the 14th Python in Science Conference, 2015, pp. 18-24.

[3] C. M. Bishop, "Pattern Recognition and Machine Learning," New York: Springer, 2006, pp. 338, 353.

[4] S. Xie et al., "Bird sound classification based on multi-view features with dual-attention mechanism," Applied Acoustics, vol. 225, 2024.

[5] X. Han and J. Peng, "Bird sound classification based on ECOC-SVM," Applied Acoustics, vol. 204, p. 109245, Mar. 2023, doi: 10.1016/j.apacoust.2023.109245.

[6] “librosa.feature.mfcc — librosa 0.10.2 documentation,” *Librosa.org*, 2023. https://librosa.org/doc/latest/generated/librosa.feature.mfcc.html

[Appendix] Table1. Number of audio data and segments for 5 species of birds.

name	length_in_seconds	segments_count	segments_total_length(i

			n second)
Barnacle Goose	375	702	140.248526
Eurasian Penduline Tit	470	728	178.631111
Hooded Crow	316	136	39.253333
Oriental Cuckoo	309	711	85.553923
Whooper Swan	388	372	107.357460

[Appendix A]: Code for data preprocessing

Part of original data:

occurrence.txt

gbifID	accessRights	bibliographicCitation	language	license	modified	publisher	references	rightsHolder	type	institutionID	collectionID	datasetID	institutionCode	collectionCode	datasetName
4940940689		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC932235				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940679		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC932114				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940659		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931935				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940658		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931924				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940639		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931674				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940638		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931663				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940616		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931429				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940582		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC931083				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940553		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930824				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940552		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930813				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940551		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930802				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940504		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930299				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940503		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930291				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940502		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930288				Johannes Worona			Wildlife sounds - Birds		MACHINE_OBSERVATION		"{""recordingDev
4940940481		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930101				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi
4940940480		CC_BY_NC_4_0		https://data.biodiversitydata.nl/xeno-canto/observation/XC930093				Johannes Worona			Wildlife sounds - Birds		HUMAN_OBSERVATION		"{""recordingDevi

Processed data for feature extraction:

: 0	en	file	file-name	type	length	date	smp	file_path	converted_file_path	segments	segments_count	audio	integer_code
0	Hooded Crow	https://xeno-canto.org/931935/download	XC931935-20240904_001_H1e_08_06_29.wav	call	0.07	2024-09-04	44100	AudioFile\XC931935-20240904_001_H1e_08_06_29.wav	audio_files/XC931935-20240904_001_H1e_08_06_29.wav	[((0.5688888888888889), (1.1145578231292517)), ((2.670294784580499), (3.1695238095238096)), ((5.804888662131519), (6.315827664399093)), ((6.826666666666667), (7.325895691609977))]	4	[0.00447083 0.00540161 0.00520325 ... 0.00210571 0.00285339 0.00167847]	1
1	Hooded Crow	https://xeno-canto.org/930288/download	XC930288-20240827_001_H1e_08_05_48.wav	call	0.04	2024-08-27	44100	AudioFile\XC930288-20240827_001_H1e_08_05_48.wav	audio_files/XC930288-20240827_001_H1e_08_05_48.wav	[((0.7314285714285714), (1.0448979591836736)), ((1.4628571428571429), (1.7298866213151927)), ((1.9620861678004535), (2.0085260770975055)), ((2.182675736961451), (2.507755102040816)), ((2.5774149659863945), (2.9141043083900023)), ((3.2624036281179136), (3.645532879818594)), ((3.982222222222222), (4.376861451247165))]	7	[0.00367737 0.0058286 0.00637817 ... -0.00810242 -0.01062012 -0.00566101]	1
2	Hooded Crow	https://xeno-canto.org/930082/download	XC930082-20240826_001_H1e_09_03_53.wav	call	0.22	2024-08-26	48000	AudioFile\XC930082-20240826_001_H1e_09_03_53.wav	audio_files/XC930082-20240826_001_H1e_09_03_53.wav	[((2.020136054421769), (2.6006349206349206)), ((2.995374149659864), (3.575873015873016)), ((16.393287981859941), (16.973786848072564)), ((17.414965986394556), (17.983854875283445))]	4	[0.00406191 0.00460359 0.00391816 ... -0.00047314 -0.00070009 -0.00039059]	1
3	Hooded Crow	https://xeno-canto.org/929213/download	XC929213-20240822_LSP5_0183_10_02_48.wav	call	0.08	2024-08-22	48000	AudioFile\XC929213-20240822_LSP5_0183_10_02_48.wav	audio_files/XC929213-20240822_LSP5_0183_10_02_48.wav	[((5.921088435374149), (6.42031746031746))]	1	[0.00065025 0.0059313 0.00057572 ... 0.00078842 0.00101328 0.]	1

Code for preprocessing [**(1)On local computer:]**

```
39
40 # Save the updated DataFrame back to CSV
41 output_csv_file_path = "finalRawData.csv"  # Replace with
   your desired output CSV file path
42 df.to_csv(output_csv_file_path, index=False)
43
44 print(f"Downloaded files and updated CSV saved at: {
   output_csv_file_path}")
45
```

File - C:\Users\cleaver\Desktop\programming hw\python\MLProject\gettingData.py

```
 1
 2 import pandas as pd
 3 import requests
 4 pd.set_option('display.max_columns', None)   # Display all
    columns
 5 pd.set_option('display.max_colwidth', None)  # Display the
    full width of each column
 6 occurrenceFile = 'occurrence.txt'
 7 rowDf = pd.read_csv(occurrenceFile, delimiter='\t')
 8 df = rowDf[['catalogNumber', 'vernacularName']].copy()
 9 df.loc[:, 'recording_id'] = df['catalogNumber'].str.
   extract(r'(\d+)$')
10 print(df.head(5))
11 target_species = df['vernacularName'].unique()  # Replace
   with actual species names
12 print(target_species)
13 target_duration = 300  # 300 seconds per species
14 species_duration = {species: 0 for species in
   target_species}  # Initialize accumulated duration for
   each species
15 # Create an empty DataFrame to store the results
16 filtered_results = pd.DataFrame()
17 def get_recording_data(recording_id):
18     url = f'https://www.xeno-canto.org/api/2/recordings?
   query=nr:{recording_id}'
19     response = requests.get(url)
20
21     if response.status_code == 200:
22         data = response.json()
23         if data['recordings']:
24             return data['recordings'][0]  # Return the
   first recording found
25     return None
26
27 for index, row in df.iterrows():
28     species = row['vernacularName']
29     recording_id = row['recording_id']
30
31     # Skip if we've reached the 300-second target
32     if species_duration[species] >= target_duration:
33         continue
34
35     # Fetch recording data
36     recording_data = get_recording_data(recording_id)
```

```
37
38      if recording_data and recording_data['q'] == 'A' and
    int(recording_data['smp']) >= 44100:  # Check for quality
     'A' and sample rate
39          length_seconds = float(recording_data['length'].
    split(":")[0]) * 60 + float(recording_data['length'].split
    (":")[1])
40          species_duration[species] += length_seconds
41          row_data = pd.json_normalize(recording_data)
42          filtered_results = pd.concat([filtered_results,
    row_data], ignore_index=True)
43          print(str(species) + ', ' + str(species_duration[
    species]))
44      if all(duration >= target_duration for duration in
    species_duration.values()):
45          print("Target durations reached for all species.")
46          break
47
48 print(filtered_results.head())
49 filtered_results.to_csv('filered_results.csv')
```

File - C:\Users\cleaver\Desktop\programming hw\python\MLProject\convertingFiles.py

```python
import pandas as pd
import os
from pydub import AudioSegment

# Read the CSV file
csv_file_path = 'finalRawData.csv'
df = pd.read_csv(csv_file_path)

# Specify the directory for converted WAV files
converted_directory = "AudioFilesInWav"
os.makedirs(converted_directory, exist_ok=True)

# Function to convert audio files to WAV format
def convert_to_wav(file_path, output_directory):
    try:
        # Load the audio file
        audio = AudioSegment.from_file(file_path)
        # Create the output file path
        file_name_without_ext = os.path.splitext(os.path.basename(file_path))[0]
        wav_file_path = os.path.join(output_directory, f"{file_name_without_ext}.wav")
        # Export as WAV
        audio.export(wav_file_path, format='wav')
        return wav_file_path
    except Exception as e:
        print(f"Error converting {file_path}: {e}")
        return None

# Convert downloaded files to WAV and store paths
df['converted_file_path'] = df['file_path'].apply(lambda path: convert_to_wav(path, converted_directory))

# Save the updated DataFrame back to CSV
output_csv_file_path = "finalRawData.csv"
df.to_csv(output_csv_file_path, index=False)

print(f"Downloaded files, converted to WAV, and updated CSV saved at: {output_csv_file_path}")

```

File - C:\Users\cleaver\Desktop\programming hw\python\MLProject\dowloadingFiles.py

```python
import pandas as pd
import os
import requests

# Read the CSV file
csv_file_path = "filered_results.csv"
df = pd.read_csv(csv_file_path)[['en', 'file', 'file-name', 'type', 'length', 'date', 'smp']].copy()

# Specify the column containing download links
url_column = "file"
filename_column = 'file-name'

# Specify the directory to save downloaded files
download_directory = "AudioFile"
os.makedirs(download_directory, exist_ok=True)

# Function to download a file
def download_file(url, download_dir, fileName):
    try:
        response = requests.get(url, stream=True)
        if response.status_code == 200:
            file_name = fileName
            file_path = os.path.join(download_dir, file_name)
            print('downloading')
            with open(file_path, 'wb') as file:
                for chunk in response.iter_content(chunk_size=8192):
                    file.write(chunk)
            return file_path
        else:
            print(f"Failed to download: {url} (Status code: {response.status_code})")
            return None
    except Exception as e:
        print(f"Error downloading {url}: {e}")
        return None

# Download files and add file paths to the DataFrame
df['file_path'] = df.apply(lambda row: download_file(row[url_column], download_directory, row[filename_column]), axis=1)

```

(2) imported to google colab:

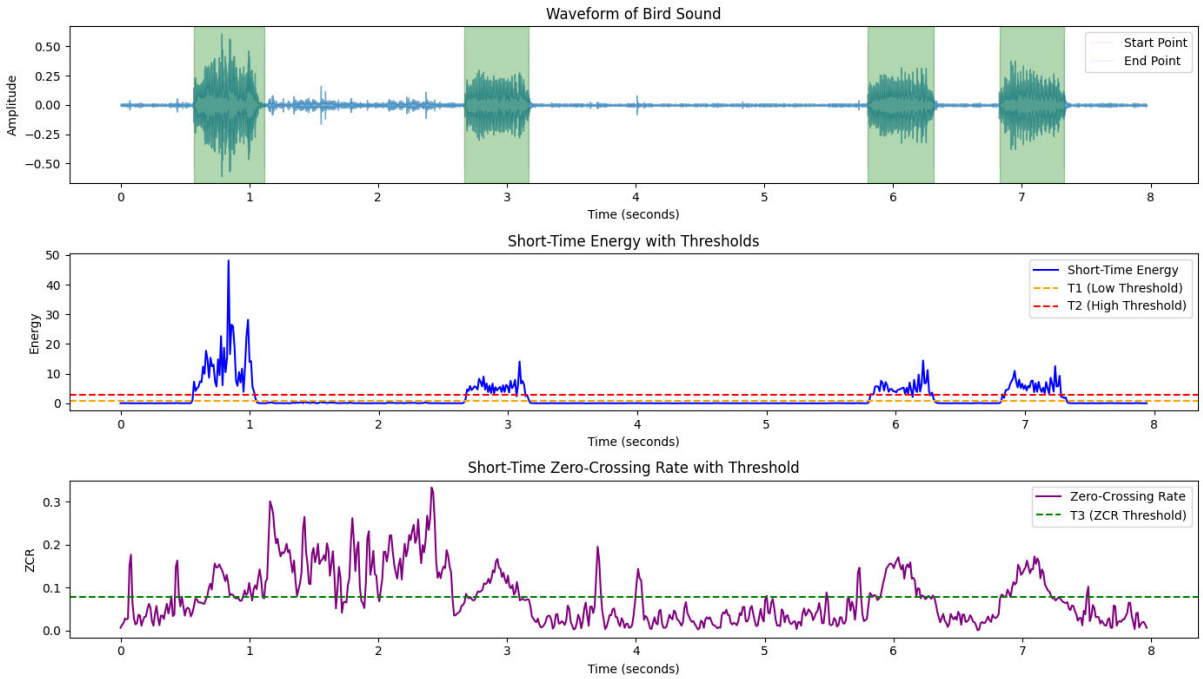
```
[48]
currentPath = '/content/drive/MyDrive/birdsound/birdClassification'
csv_path = os.path.join(currentPath, 'segmentedData.csv')
df = pd.read_csv(csv_path)
df['audio'] = df['converted_file_path'].apply(lambda x: librosa.load(os.path.join(currentPath,x), sr=44100)[0]) # interpret the WAV file
```

```
Start coding or generate with AI.
```

Show hidden output

```
[ ]
#code to convert the original dataframe to the ideal format and shape in google colab, only use once
df['converted_file_path'] = df['converted_file_path'].apply(lambda x: x.replace('\\', '/').replace('AudioFilesInWav', 'audio_files'))

labels = df['en'].unique()
# 1. Integer Encoding
label_encoder = LabelEncoder()
integer_encoded = label_encoder.fit_transform(labels)

species_dict = {}
for i, species in enumerate(label_encoder.classes_):
  species_dict[species] = integer_encoded[i]
print(species_dict)
df['integer_code'] = df['en'].map(species_dict)
df['segments'] = df['segments'].str.replace('np.float64', '')
#df.to_csv('segmentedData.csv')
```

[Appendix B segmentation of T1,T2,T3 visualization and code][5]



```
 1 import librosa
 2 import numpy as np
 3 import pandas as pd
 4 import matplotlib.pyplot as plt
 5
 6 df = pd.read_csv('finalRawData.csv')
 7
 8 # Load the audio file
 9 audio_paths = df['converted_file_path']
10
11 def gettingSegments(audio_path):
12     y, sr = librosa.load(audio_path, sr=44100)
13
14     # Normalize the audio signal
15     y = y / np.max(np.abs(y))
16
17     # Define frame length and hop length
18     frame_length = 1024
19     hop_length = 512
20
21     # Apply a window function (Hamming window)
22     window = np.hamming(frame_length)
23
24     # Calculate STE
25     energy = np.array([
26         np.sum((y[i:i + frame_length] * window) ** 2)
27         for i in range(0, len(y) - frame_length + 1,
   hop_length)
28     ])
29     #print(energy)
30
31
32     # Define T2 and T1 for the double threshold
33     T2 = np.mean(energy) * 1.5
34     T1 = np.mean(energy) * 0.5
35     #print((T1, T2))
36     # Find segments above T2
37     segments = np.where(energy > T2)[0]
38     #print(segments)
39     # Find start (A) and end points (B) using T1
40     start_end_points = []
41     start = None
42
43     # Iterate over all frames of energy, not just the
```

```
43 segments
44     for i in range(len(energy)):
45         # Detect the start of a sound segment when energy
   crosses above T2
46         if start is None and energy[i] > T2:
47             start = i
48 
49         # Detect the end of a sound segment when energy
   crosses below T1 after starting
50         if start is not None and energy[i] < T1:
51             end = i
52             start_end_points.append((start, end))
53             start = None
54 
55     # Handle the case where a segment reaches the end of
   the signal
56     if start is not None:
57         start_end_points.append((start, len(energy) - 1))
58 
59     #print("Start and end points (in frame indices):",
   start_end_points)
60 
61     # Calculate Zero-Crossing Rate (ZCR)
62     zcr = librosa.feature.zero_crossing_rate(y,
   frame_length=frame_length, hop_length=hop_length)[0]
63 
64     # Define T3 as the mean ZCR
65     T3 = np.mean(zcr)
66 
67     # Apply ZCR refinement to find the final segments
68     final_segments = []
69     for (start, end) in start_end_points:
70         # Search outward from start and end points until
   ZCR is below T3
71         while start > 0 and zcr[start] > T3:
72             start -= 1
73         while end < len(zcr) and zcr[end] > T3:
74             end += 1
75         final_segments.append((start, end))
76 
77     # Convert frame indices to time
78     final_segments_time = [(librosa.frames_to_time(s, sr=
   sr, hop_length=hop_length),
79                             librosa.frames_to_time(e, sr=sr
```

```
 79 , hop_length=hop_length))
 80                             for (s, e) in final_segments]
 81     return final_segments_time
 82
 83 df['segments'] = audio_paths.apply(lambda path:
    gettingSegments(path))
 84 df['segments_count'] = df['segments'].apply(lambda seg:
    len(seg))
 85 #df.to_csv('segmentedData.csv')
 86
 87
 88 #print("Final detected segments (start, end in seconds
    ):")
 89 #to plot a example segmentation
 90 def gettingSegmentsAndPlot(audio_path):
 91     y, sr = librosa.load(audio_path, sr=44100)
 92
 93     # Normalize the audio signal
 94     y = y / np.max(np.abs(y))
 95
 96     # Define frame length and hop length
 97     frame_length = 1024
 98     hop_length = 512
 99
100     # Apply a window function (Hamming window)
101     window = np.hamming(frame_length)
102
103     # Calculate STE
104     energy = np.array([
105         np.sum((y[i:i + frame_length] * window) ** 2)
106         for i in range(0, len(y) - frame_length + 1,
    hop_length)
107     ])
108     #print(energy)
109
110
111     # Define T2 and T1 for the double threshold
112     T2 = np.mean(energy) * 1.5
113     T1 = np.mean(energy) * 0.5
114     #print((T1, T2))
115     # Find segments above T2
116     segments = np.where(energy > T2)[0]
117     #print(segments)
118     # Find start (A) and end points (B) using T1
```

```
119     start_end_points = []
120     start = None
121
122     # Iterate over all frames of energy, not just the
    segments
123     for i in range(len(energy)):
124         # Detect the start of a sound segment when energy
     crosses above T2
125         if start is None and energy[i] > T2:
126             start = i
127
128         # Detect the end of a sound segment when energy
    crosses below T1 after starting
129         if start is not None and energy[i] < T1:
130             end = i
131             start_end_points.append((start, end))
132             start = None
133
134     # Handle the case where a segment reaches the end of
    the signal
135     if start is not None:
136         start_end_points.append((start, len(energy) - 1))
137
138     #print("Start and end points (in frame indices):",
    start_end_points)
139
140     # Calculate Zero-Crossing Rate (ZCR)
141     zcr = librosa.feature.zero_crossing_rate(y,
    frame_length=frame_length, hop_length=hop_length)[0]
142
143     # Define T3 as the mean ZCR
144     T3 = np.mean(zcr)
145
146     # Apply ZCR refinement to find the final segments
147     final_segments = []
148     for (start, end) in start_end_points:
149         # Search outward from start and end points until
    ZCR is below T3
150         while start > 0 and zcr[start] > T3:
151             start -= 1
152         while end < len(zcr) and zcr[end] > T3:
153             end += 1
154         final_segments.append((start, end))
155
```

```
156     # Convert frame indices to time
157     final_segments_time = [(librosa.frames_to_time(s, sr=
    sr, hop_length=hop_length),
158                             librosa.frames_to_time(e, sr=
    sr, hop_length=hop_length))
159                            for (s, e) in final_segments]
160
161
162     for segment in final_segments_time:
163         print(segment)
164     y, sr = librosa.load(df['converted_file_path'].iloc[0
    ], sr=44100)
165     # Plotting
166     plt.figure(figsize=(14, 8))
167
168     # Plot the waveform
169     plt.subplot(3, 1, 1)
170     librosa.display.waveshow(y, sr=sr, alpha=0.6)
171     plt.title("Waveform of Bird Sound")
172     plt.xlabel('Time (seconds)')
173     plt.ylabel('Amplitude')
174
175     # Mark final segments start and end points
176     for start, end in final_segments_time:
177         plt.axvline(x=start, color='red', linestyle='--'
    , label='Start Point' if start == final_segments_time[0][
    0] else "",
178                     linewidth=0.1, ymin=0.1, ymax=0.9)
179         plt.axvline(x=end, color='blue', linestyle='--',
    label='End Point' if end == final_segments_time[0][1]
    else "",
180                     linewidth=0.1, ymin=0.1, ymax=0.9)
181     plt.legend()
182
183     # Mark final segments
184     for start, end in final_segments_time:
185         plt.axvspan(start, end, color='green', alpha=0.3
    , label='Detected Segment')
186
187     # Plot Short-Time Energy (STE)
188     plt.subplot(3, 1, 2)
189     frames = range(len(energy))
190     t = librosa.frames_to_time(frames, sr=sr, hop_length=
    hop_length)
```

```
191     plt.plot(t, energy, label='Short-Time Energy', color=
    'blue')
192     plt.axhline(T1, color='orange', linestyle='--', label
    ='T1 (Low Threshold)')
193     plt.axhline(T2, color='red', linestyle='--', label='
    T2 (High Threshold)')
194     plt.title("Short-Time Energy with Thresholds")
195     plt.xlabel('Time (seconds)')
196     plt.ylabel('Energy')
197     plt.legend(loc='upper right')
198 
199     # Plot Short-Time Zero-Crossing Rate (ZCR)
200     frames_zcr = range(len(zcr))  # Number of frames
    corresponding to ZCR
201     t_zcr = librosa.frames_to_time(frames_zcr, sr=sr,
    hop_length=hop_length)
202     plt.subplot(3, 1, 3)
203     plt.plot(t_zcr, zcr, label='Zero-Crossing Rate',
    color='purple')
204     plt.axhline(T3, color='green', linestyle='--', label=
    'T3 (ZCR Threshold)')
205     plt.title("Short-Time Zero-Crossing Rate with
    Threshold")
206     plt.xlabel('Time (seconds)')
207     plt.ylabel('ZCR')
208     plt.legend(loc='upper right')
209 
210     plt.tight_layout()
211     plt.show()
212     return final_segments_time
213 
214 gettingSegmentsAndPlot(audio_paths.iloc[0])
215 
```

[Appendix C] : Feature parameter finetuning and extraction:

```
#to test out the best parameters to use for mfccs extraction, only run once to get the result, the result is then stored
sr = 44100
# Define parameter arrays
n_mfcc_vals = [13, 20]
n_fft_vals = [64, 128, 256]
hop_length_vals = [32, 64, 128]
n_mels_vals = [20, 32, 40, 64]
lifter_vals = [0, 22]
results = []
for n_mfcc in n_mfcc_vals:
    for n_fft in n_fft_vals:
        for hop_length in hop_length_vals:
            for n_mels in n_mels_vals:
                for lifter in lifter_vals:
                  X = []
                  y = []
                  for index, row in df.iterrows():
                    # Access data from each row using row['column_name']
                    audio_data = row['audio']
                    species = row['en']
                    integer_code = row['integer_code']
                    segments = ast.literal_eval(row['segments'])
                    for segment in segments:
                        start, end = segment

                        # Extract the segment from the audio
                        segment_audio = audio_data[int(start * sr):int(end * sr)]

                        # Extract MFCC features for the segment
                        mfccs = librosa.feature.mfcc(y=segment_audio, sr=sr, n_mfcc=n_mfcc, n_fft=n_fft,n_mels = n_mels, hop_length = hop_length, lifter = lifter)
                        mfccs_mean = np.mean(mfccs, axis=1)
                        mfccs_std = np.std(mfccs, axis=1)
                        mfccs_features = np.concatenate((mfccs_mean, mfccs_std))
                        X.append(mfccs_features)
                        y.append(integer_code)


                  X = np.array(X).reshape(np.array(X).shape[0], -1)
```

✓ 0s completed at 6:14 PM

+ Text

```
X = np.array(X).reshape(np.array(X).shape[0], -1)
y = np.array(y)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the SVM classifier
svm = SVC(kernel='linear', C=1)
svm.fit(X_train, y_train)

# Predict on the test set
y_pred = svm.predict(X_test)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)

# Set up k-fold cross-validation (k = 5)
kf = KFold(n_splits=5, shuffle=True, random_state=42)
X_scaled = scaler.fit_transform(X)
# Perform cross-validation and get accuracy scores

validation_scores = cross_val_score(svm, X_scaled, y, cv=kf, scoring='accuracy')
#print("Accuracy for each fold:", validation_scores)
#print("Mean accuracy:", validation_scores.mean())
result = {
      'n_mfcc': n_mfcc,
      'n_fft': n_fft,
      'hop_length': hop_length,
      'n_mels': n_mels,
      'lifter': lifter,
      'accuracy': accuracy,
      'validation_scores': validation_scores.mean()
  }
results.append(result)
```

✓ 0s completed at 6:14 PM

Feature extraction with fine-tuned parameters:

```
                    'n_fft': n_fft,
                    'hop_length': hop_length,
                    'n_mels': n_mels,
                    'lifter': lifter,
                    'accuracy': accuracy,
                    'validation_scores': validation_scores.mean()
                }
            results.append(result)
df_results = pd.DataFrame(results)
df_results = df_results.drop_duplicates().sort_values(by='validation_scores', ascending=False)
df_results.to_csv('results.csv', index=False)
max_row = df_results.head(20)
print(max_row)
```

```
#the best parameter combos
mfcc_csv_path = os.path.join(currentPath, 'top20mfccParams.csv')
mfccParams = pd.read_csv(mfcc_csv_path)
print(mfccParams)
```

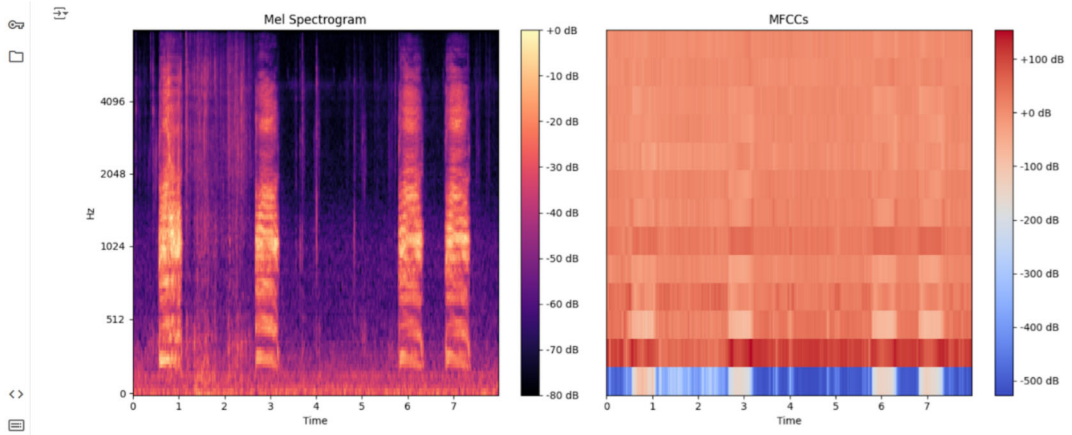
```
    n_mfcc  n_fft  hop_length  n_mels  accuracy  validation_scores
0       20    128          64      64  0.966038           0.968673
1       20    128          32      64  0.964151           0.968669
2       20     64          32      40  0.956604           0.966029
3       20     64          32      64  0.949057           0.964519
4       20     64          32      32  0.950943           0.964139
5       20    256          64      64  0.956604           0.963764
6       20     64          64      64  0.954717           0.963388
7       20    256          32      64  0.954717           0.963010
8       20    256          32      32  0.949057           0.963009
9       20     64          64      40  0.956604           0.961499
10      20    256          64      32  0.950943           0.961499
11      20    256         128      64  0.954717           0.961123
12      20     64          64      40  0.956604           0.961121
13      20    256         128      32  0.939623           0.960747
```

```
#use the best paramter combo to extract the features
sr = 44100
X = []
y = []
for index, row in df.iterrows():
  # Access data from each row using row['column_name']
    audio_data = row['audio']
    species = row['en']
    integer_code = row['integer_code']
    segments = ast.literal_eval(row['segments'])
    for segment in segments:
        start, end = segment

        # Extract the segment from the audio
        segment_audio = audio_data[int(start * sr):int(end * sr)]

        # Extract MFCC features for the segment
        mfccs = librosa.feature.mfcc(y=segment_audio, sr=sr, n_mfcc=20, n_fft=128,n_mels = 64, hop_length = 64)

        mfccs_mean = np.mean(mfccs, axis=1)
        mfccs_std = np.std(mfccs, axis=1)
        mfccs_features = np.concatenate((mfccs_mean, mfccs_std))
        X.append(mfccs_features)
        y.append(integer_code)

X = np.array(X).reshape(np.array(X).shape[0], -1)
y = np.array(y)
#print(X.shape)
#print(y.shape)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

✓ 0s completed at 6:14 PM

[Appendix D]MFCCs and Mel-spectrogram visualization of a sound-clip of hooded crow

```
#example mel_sepctrogram and mfccs figure
example_audio = df['audio'].iloc[0]
# Extract MFCC features
mfccs = librosa.feature.mfcc(y=example_audio, sr=sr, n_mfcc=13)

# Extract Mel Spectrogram
mel_spectrogram = librosa.feature.melspectrogram(y=example_audio, sr=sr, n_mels=128)
mel_spectrogram_db = librosa.power_to_db(mel_spectrogram, ref=np.max)  # Convert to dB for better visualization

# Create subplots
plt.figure(figsize=(14, 6))

# Plot Mel Spectrogram
plt.subplot(1, 2, 1)
librosa.display.specshow(mel_spectrogram_db, x_axis='time', y_axis='mel', sr=sr, fmax=8000)
plt.colorbar(format='%+2.0f dB')
plt.title('Mel Spectrogram')

# Plot MFCCs
plt.subplot(1, 2, 2)
librosa.display.specshow(mfccs, x_axis='time', sr=sr)
plt.colorbar(format='%+2.0f dB')
plt.title('MFCCs')

# Display the plots
plt.tight_layout()
plt.show()
```



[Appendix E]Training and Fine-Tuning SVM model:

```
#results with default parameters for SVM
svm_basic = SVC(random_state=42)
svm_basic.fit(X_train, y_train)
y_pred = svm_basic.predict(X_test)
# Evaluate accuracy
accuracy_basic = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy_basic * 100:.2f}%")
```

```python
#searching for the best hyperparameters
param_grid = {
    'C': [0.1, 1, 10, 100],              # Regularization parameter
    'kernel': ['linear', 'rbf', 'poly'], # Different kernels
    'gamma': [0.001, 0.01, 0.1, 1],     # Kernel coefficient (for rbf, poly)
    'degree': [2, 3, 4],                 # Degree of the polynomial (for poly kernel)
    'class_weight': ['balanced']  # Class weights for imbalanced classes
}
# Train the SVM classifier

svm = SVC()

# Use GridSearchCV to tune hyperparameters
grid_search = GridSearchCV(svm, param_grid, cv=5, scoring='accuracy', verbose=1, n_jobs=-1)
grid_search.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 144 candidates, totalling 720 fits
/usr/local/lib/python3.10/dist-packages/numpy/ma/core.py:2820: RuntimeWarning: invalid value encountered in cast
  _data = np.array(data, dtype=dtype, copy=copy,
```

▸ GridSearchCV

▸ best_estimator_: SVC

▸ SVC

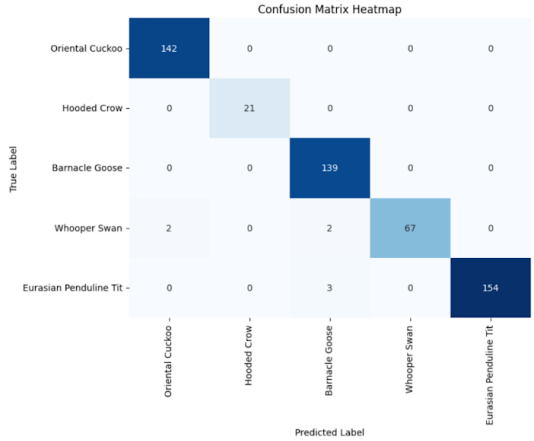
```python
[40] # Get the best hyperparameters from the grid search
best_params = grid_search.best_params_
best_accuracy = grid_search.best_score_

print("Best Hyperparameters:", best_params)
print("Best Accuracy:", best_accuracy) #best mean cross validation scores
#use the best parameters to train the data
svm = SVC(
    C=best_params['C'],
    kernel=best_params['kernel'],
    gamma=best_params['gamma'],  # gamma is not relevant for 'linear' kernel
    degree=best_params['degree'],      # degree only relevant for 'poly' kernel
    class_weight=best_params['class_weight'],       # degree only relevant for 'poly' kernel
    random_state = 42
)
svm.fit(X_train, y_train)

# Predict on the test set
y_pred = svm.predict(X_test)
```

```
Best Hyperparameters: {'C': 10, 'class_weight': 'balanced', 'degree': 2, 'gamma': 0.01, 'kernel': 'rbf'}
Best Accuracy: 0.9830144074222759
```

Fine Tuned SVM model results:

Accuracy: 98.68%

	precision	recall	f1-score	support
0	0.99	1.00	0.99	142
1	1.00	1.00	1.00	21
2	0.97	1.00	0.98	139
3	1.00	0.94	0.97	71
4	1.00	0.98	0.99	157
accuracy			0.99	530
macro avg	0.99	0.98	0.99	530
weighted avg	0.99	0.99	0.99	530



[Appendix F]Training and fine tuning Random Forest model:

```
[ ] #get the results for Random forest use mostly the default parameters
    # Initialize Random Forest
    rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
    # Train the model
    rf_model.fit(X_train, y_train)
    y_pred_rf = rf_model.predict(X_test)
    # Evaluate the model performance
    print(classification_report(y_test, y_pred_rf))
    accuracy = accuracy_score(y_test, y_pred_rf)
    print(f"Accuracy: {accuracy * 100:.2f}%")
```

```
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       142
           1       0.95      0.90      0.93        21
           2       0.96      0.99      0.98       139
           3       0.98      0.89      0.93        71
           4       0.99      0.98      0.98       157

    accuracy                           0.97       530
   macro avg       0.97      0.95      0.96       530
weighted avg       0.97      0.97      0.97       530

Accuracy: 97.17%
```

```
#fine_tuning the parameters
# Define the parameter grid
param_grid_rf = {

    'n_estimators': [100, 200, 300],
    'min_samples_split': [10, 20, 30],
    'min_samples_leaf': [1, 2],
    'max_features': ['sqrt',0.5, None],
    'class_weight': ['balanced', 'balanced_subsample'],


}

# Initialize the RandomForestClassifier
rf = RandomForestClassifier(random_state=42)

# Grid search with cross-validation (cv=5)
grid_search_rf = GridSearchCV(estimator=rf, param_grid=param_grid_rf, cv=5, scoring='accuracy', verbose=1, n_jobs=-1)
grid_search_rf.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 108 candidates, totalling 540 fits
▸ GridSearchCV
▸ best_estimator_: RandomForestClassifier
    ▸ RandomForestClassifier
```

✓ 0s completed at 6:14 PM

```
[ ]
    # Get the best hyperparameters
    best_params_rf = grid_search_rf.best_params_
    best_accuracy_rf = grid_search_rf.best_score_
    print("Best hyperparameters: ", best_params_rf)
    print("Best Accuracy:", best_accuracy_rf)
```

```
Best hyperparameters:  {'class_weight': 'balanced_subsample', 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 10, 'n_estimators': 100}
Best Accuracy: 0.9792330166376733
```

```
rf_model = RandomForestClassifier(
    n_estimators=best_params_rf['n_estimators'],
    min_samples_split=best_params_rf['min_samples_split'],
    min_samples_leaf= best_params_rf['min_samples_leaf'],
    max_features= best_params_rf['max_features'],
    class_weight= best_params_rf['class_weight'],
    random_state = 42
)

# Train the model
rf_model.fit(X_train, y_train)
y_pred_rf = rf_model.predict(X_test)
# Evaluate the model performance
print(classification_report(y_test, y_pred_rf))
accuracy = accuracy_score(y_test, y_pred_rf)
print(f"Accuracy: {accuracy * 100:.2f}%")

cm_rf = confusion_matrix(y_test, y_pred_rf)
# Create a heatmap using seaborn
plt.figure(figsize=(8, 6))
sns.heatmap(cm_rf, annot=True, fmt='g', cmap='Blues', cbar=False,
            xticklabels=labels, yticklabels=labels)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix Heatmap')
plt.show()
```

✓ 0s completed at 6:14 PM

Random Forest Results:

	precision	recall	f1-score	support
0	0.99	1.00	1.00	142
1	1.00	1.00	1.00	21
2	0.97	1.00	0.98	139
3	0.98	0.92	0.95	71
4	0.99	0.98	0.98	157
accuracy			0.98	530
macro avg	0.99	0.98	0.98	530
weighted avg	0.98	0.98	0.98	530

Accuracy: 98.30%