USING WING FLAP SOUNDS TO DISTINGUISH INDIVIDUAL BIRDS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Thinh Phan

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Department:
Electrical and Computer Engineering

February 2024

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

USING WING FLAP SOUNDS TO DISTINGUISH INDIVIDUAL BIRDS

By

Thinh Phan

The Supervisory Committee certifies that this thesis complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Roger Green

Chair

Dr. Jacob Glower

Dr. Erin Gillam

Approved:

March 4, 2024

Date

Dr. Benjamin Braaten

Department Chair

ABSTRACT

Ornithologist often need to recognize individual birds, but traditional invasive methods, such as capturing, marking, releasing, and recapturing of birds, have limitations. To overcome this, researchers use non-invasive alternatives, such as bird vocalizations. In our study, we used wing flap sounds of three male Zebra Finch birds for individual recognition. We achieved identification accuracies ranging from 55% to 100% by using a combination of Principal Component Analysis-K-Nearest Neighbor (PCA-KNN) and Cross-Correlation method on training data and testing data. PCA-KNN allows for dimensionality reduction and pattern recognition, while the Cross-Correlation method bases data analysis on shifting data elements. Our approach can be applied to other bird species and is becoming more accessible due to technological advancements. Non-invasive methods for bird identification are becoming increasingly popular, and our study demonstrates the potential for using wing flap sounds to recognize individual birds.

ACKNOWLEDGEMENTS

I would like to express my profound and heartfelt gratitude to my advisor, Dr. Roger Green, for his unwavering guidance, invaluable advice, and continuous support throughout the course of my Master's research study. His enthusiastic encouragement and insightful feedback have been pivotal in navigating the complex challenges of my thesis work, inspiring me to delve deeper into my field and continually strive for excellence. Dr. Green's dedication to mentoring has not only shaped my research journey but also profoundly influenced my personal and professional growth.

Furthermore, I am immensely thankful to Dr. Jacob Glower for his invaluable contributions to my thesis. His unique perspectives and methodical approach to problem-solving have significantly broadened my analytical skills and enhanced my ability to tackle technical challenges with innovative solutions. Dr. Glower's constructive critiques and positive reinforcement have been instrumental in refining my work and ensuring its success.

I would also like to extend my heartfelt thanks to Dr. Erin Gillam for her invaluable participation on my committee. Her readiness to delve into a field beyond her primary area of expertise has been a testament to the spirit of interdisciplinary scholarship and a significant source of inspiration for me. Dr. Gillam's unique perspective and open–minded approach to my research area have encouraged me to think more expansively and creatively. Her involvement has been pivotal in fostering a broader understanding of my work, allowing me to incorporate diverse methodologies and viewpoints. This enriching experience has significantly contributed to the depth and breadth of my academic journey, for which I am profoundly grateful.

The mentorship and contributions of Dr. Green, Dr. Glower, and Dr. Gillam have been fundamental to the completion of this thesis. Their collective commitment to excellence, combined with their genuine interest in my research, have been continual sources of motivation and inspiration. I am deeply appreciative of their guidance and support, which have been vital in bringing my research to fruition and achieving this significant milestone in my academic career.

In addition, special thanks are due to Troy Reisenauer and Drew Taylor, Graduate Consultants, who have provided invaluable assistance with the English aspects of my thesis. Their keen insights and meticulous attention to detail have greatly enhanced the clarity and coherence of my writing. Their support has not only helped in polishing my thesis but also in improving my academic writing skills more broadly. I am truly grateful for their guidance and expertise, which have been instrumental in the successful completion of this work.

I would also like to acknowledge the assistance provided by ChatGPT in refining the English language used throughout my thesis. The immediate and insightful support from ChatGPT played a significant role in enhancing the overall quality of my manuscript, allowing me to express my research findings more clearly and effectively. This tool has been a valuable resource, aiding not just in language improvement but also in facilitating a deeper understanding of my subject matter through engaging discussions. My thanks to ChatGPT for its contribution to the success of my thesis.

DEDICATION

This thesis is dedicated to my family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION AND THESIS STATEMENT

1.1. Introduction

There have been studies conducted on how birds, both male and female, attend nests during the breeding season. To monitor bird nest attendance, researchers often capture the birds and place bands on their legs (Boulton et al., 2010; Cantarero et al., 2016; Sandell et al., 1996; Wierucka et al., 2016). These bands can be made of metal or plastic and are fitted onto the bird's leg. However, the process of capturing and marking birds can have negative impacts on their well-being. Capturing birds can cause stress, leading to changes in behavior and potentially affect study results (Linhart et al., 2012). Furthermore, metal and plastic rings can cause injury to the birds. Plastic rings may cause inflammation and infection, and the buildup of substances under the ring can lead to physical damage. Metal rings, on the other hand, can be sharp and cause injuries (Amat, 1999; Griesser et al., 2012).

Efficient and accurate methods for individual bird identification are crucial to the study and management of birds. Invasive methods, which generally involve banding individual birds, can be highly accurate (approaching 100%) but are usually expensive, pose a risk of bird injury and can produce undesirable behavior changes: birds may become more alert to their surroundings to avoid being recaptured.

Non-invasive, or near non-invasive, techniques are therefore gaining increased interest. Methods based on human observation can be extremely accurate and provide nuanced insights but are generally prohibitively expensive, difficult to achieve continuous monitoring, and challenging to scale up (Hałupka, 1994). Camera or video systems can be highly accurate but are expensive and complex (Ferreira et al., 2020).

Identification based on bird calls (Bedoya and Molles, 2021; Galeotti and Pavan, 1991; Pedroza et al., 2020; Rogers and Paton, 2005) is typically less expensive and less likely to induce behavior changes but is better suited for species-level rather than individual identification. Such systems, of course, cannot work in the absence of bird vocalizations and are generally less accurate than invasive approaches. Additionally, mimicry by other birds can affect the accuracy of the results.

This thesis propose a new approach: the use of wing flap sounds to identify individual birds, well-suited for many bird studies that involve monitoring ingress and egress at nests or feeding sites. This method is similar to gait detection methods used to identify humans (AbdAlKader, 2019; Zhang et al., 2004). Birds have unique wing flap sounds that can be used to identify individual birds without causing any stress or harm to them. This approach is well-suited to many bird studies that involve monitoring nest or feeding site ingress and egress.

This method is completely non-invasive and can be hidden so as to avoid inducing changes in bird behaviors. While not as accurate as some other identification systems, we demonstrate reasonable accuracy is possible, and accuracy rates are sure to improve as algorithm development and refinement continue.

1.2. Thesis Statement

This study focuses on the individual identification of three male Zebra Finch birds based on the differences in the sound of their wing flaps. The Zebra Finches were selected based on their availability and active behavior, and wing flap audio plots were used to analyze their flight sounds. The analysis revealed that each bird has unique acoustic features associated with its flight, and that the sound waveforms differ consistently between birds.

To classify the birds based on their wing flap sounds, we used a combination of Principal Component Analysis (PCA) and K-Nearest Neighbor (KNN) algorithms (PCA-KNN), as well as the Cross-Correlation method. The PCA-KNN techniques allowed for dimensionality reduction and pattern recognition, while the Cross-Correlation technique provided an additional method that potentially gives more accurate data analysis by shifting data elements, although this method does require relatively more computer time and power.

Our results demonstrate that the PCA-KNN and Cross-Correlation methods are highly effective in identifying individual birds based on their wing flap sounds. We observed identification accuracies ranging from 55% to 100%. The most frequent identification accuracy recorded across our tests was 85%. This approach offers a non-invasive alternative to traditional bird capturing and marking techniques. Moreover, its feasibility is increasingly enhanced by ongoing technological advancements.

Overall, our study highlights the potential of using wing flap sounds for individual bird identification and demonstrates the effectiveness of the PCA-KNN and Cross-Correlation methods in this context. Further research can improve the accuracy of the identification technique and expand its applicability to other bird species.

1.2.1. Introduction to Distance Metrics

Distance metrics, sometimes referred to as similarity or dissimilarity measures, play a foundational role in various fields such as machine learning (Hoi et al., 2006; Yang and Jin, 2006), statistics (Cohen et al., 2003), and data mining (Singh et al., 2013). They essentially quantify the “distance” or difference between two data points and are instrumental for tasks ranging from clustering (where the aim is to group similar data points together) (Singh et al., 2013) to classification (where the goal is to assign a data point to a predefined group)

(Singh et al., 2013), recommendation systems (recommending items based on similarity to user preferences) (Kumar et al., 2015; Yu et al., 2017), and nearest neighbor searches (finding the closest data point or points to a given point) (Brin, 1995; Ruan et al., 2021).

It's important to note that the concept of "distance" in this context does not always pertain to physical distance. Rather, it serves as an abstract representation of how different two sets of data are from each other. A few key aspects to consider when discussing distance metrics include the type of data being worked with; the nature of your data (be it categorical, continuous, binary, etc.) can often dictate the best distance metric to use. The specific problem at hand also matters; depending on the problem, some metrics might be more appropriate than others (Singh et al., 2013). Additionally, robustness is a crucial factor as some metrics are more sensitive to outliers or variations in the data (Singh et al., 2013). Finally, computational efficiency is a concern, especially in high-dimensional spaces or with large datasets. In such cases, computing distances can become computationally intensive, making some metrics more efficient than others (Suárez et al., 2021).

1.2.2. Method 1: PCA-KNN for Distinguishing Individual Birds

In this study, one of the methods employed for distinguishing individual birds is PCA-KNN, a hybrid of Principal Component Analysis (PCA) and K-Nearest Neighbors (KNN). PCA is a dimensionality reduction technique that transforms the original variables into a new set of uncorrelated variables, known as principal components. These principal components retain most of the original data's variance, often allowing for a reduction in the dimensionality of the data without significant loss of information (Jolliffe, 2002; Shlens, 2014; Wold et al., 1987). KNN is a simple yet effective algorithm for classification, where an unknown sample is classified based on the majority class of its 'K' nearest neighbors in the feature space (Kramer and Kramer, 2013; Laaksonen and Oja, 1996; Mucherino et al., 2009).

By combining PCA and KNN, we can achieve efficient and accurate classification. PCA aids in reducing the dimensionality of the dataset, making the KNN algorithm faster and more effective(Borman et al., 2021; Haque et al., 2019; Xia et al., 2021). In the context of distinguishing individual birds based on wing flap sounds, PCA is first applied to reduce the dimensionality of the sound feature space. The KNN algorithm then uses this reduced feature space for classification, employing various distance metrics like Euclidean, Cityblock, Correlation, and Cosine distance for enhanced accuracy.

1.2.2.1. Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a statistical technique used to reduce the complexity of high-dimensional datasets while retaining most of the important information. The fundamental idea behind PCA is to transform the original variables into a new set of variables, called principal components, that are uncorrelated with each other and capture the maximum amount of variation in the data. In essence, PCA seeks to summarize the most important features or patterns in the data, while removing redundant or irrelevant information (Daffertshofer et al., 2004).

The process of performing PCA involves several steps (Karim et al., 2010). The first step is to standardize the data by subtracting the mean and dividing by the standard deviation of each variable. This ensures that each variable is on the same scale and prevents one variable from dominating the analysis. Next, the correlation matrix is calculated, which shows the linear relationship between each pair of variables. The eigenvectors and eigenvalues of the correlation matrix are then computed, with the eigenvectors representing the directions in the dataset that have the most variation. The eigenvalues correspond to the amount of variance explained by each eigenvector, with larger eigenvalues indicating more important principal components.

The principal components are then extracted by multiplying the original dataset by the eigenvectors. The first principal component captures the largest amount of variance in the data, while each subsequent principal component captures as much of the remaining variation as possible, subject to the constraint of being orthogonal to all preceding components. The principal components can be used for data visualization, feature extraction, and data compression. The extracted principal components are typically ordered by decreasing eigenvalue, so that the first few components capture most of the variation in the data.

One of the benefits of PCA is that it can be used to reduce the number of dimensions in a dataset without losing too much information. This is useful for visualizing high-dimensional data, as well as for machine learning and predictive modeling applications. Principal Component Analysis can also be used to identify the most important variables in a dataset, which can help to simplify the analysis and improve the accuracy of the results (Wold et al., 1987).

However, there are also some potential limitations and challenges associated with PCA. One of the challenges is interpreting the results of the analysis. While the principal components capture the most important patterns in the data, it can be difficult to understand what these patterns represent, especially when dealing with large or complex datasets. Additionally, PCA assumes that the data is linearly related, and may not work well with categorical data (Wold et al., 1987).

In the end, PCA is a powerful and widely used technique for reducing the complexity of high-dimensional datasets. By transforming the data into a new set of variables that capture the most important patterns, PCA can help to improve the accuracy of analysis and modeling, as well as simplify the visualization of complex data. However, it is important to use PCA appropriately and to interpret the results carefully, as well as to be aware of its potential limitations and challenges.

1.2.2.2. K-Nearest Neighbor (KNN)

K-Nearest Neighbor (KNN) is a popular machine learning algorithm used for classification problems (Bukhari et al., 2020; Jutzi and Gross, 2009; Lakshmi and Prabakaran, 2014; Ramteke and Monali, 2012). The KNN learning algorithm does not build a model beforehand, it relies on the data itself to make predictions. KNN is based on the idea that similar objects belong to the same class as objects that are close to each other.

In a KNN classification problem, a set of labeled training data is used to make predictions on new, unseen data. The training data is composed of various classes that have been classified differently, as their unique features set them apart. To make a prediction, the KNN calculates the distances between the test data and all training data. The algorithm then selects the K nearest neighbors (based on the distance metric used) and assigns the test data to the class that is majority represented among the K nearest neighbors. The value of K is a user-defined parameter that determines the number of neighbors to consider for each prediction (Ali et al., 2019; Kataria and Singh, 2013).

There are several distance metrics that can be used with KNN, including Euclidean distance, Cityblock distance, Cosine distance, and Correlation distance. Euclidean distance is the most used distance metric, and is simply the straight-line distance between two points in space. Cityblock distance is the sum of the absolute differences between the coordinates of two points. The choice of distance metric depends on the characteristics of the data and the specific requirements of the problem(Ali et al., 2019; Kataria and Singh, 2013).

One of the main advantages of the KNN algorithm is its simplicity. It is easy to understand and implement and does not require a lot of data preparation. In addition, KNN is flexible, as it can handle both continuous and categorical data. (Ali et al., 2019; Kataria and Singh, 2013).

Another advantage of KNN is that it is computationally efficient. Unlike other algorithms, KNN does not require training of a model. Instead, it relies on the training data to make predictions, which makes it fast and easy to use for large data sets. KNN is also adaptable, as the value of K can be adjusted to control the trade-off between overfitting and underfitting(Ali et al., 2019; Kataria and Singh, 2013).

However, there are also some disadvantages of KNN (Ali et al., 2019; Kataria and Singh, 2013). One of the main limitations is that it can be sensitive to the choice of distance metric. The choice of distance metric can greatly impact the performance of the algorithm, and it can be difficult to determine the optimal metric for a particular data set. In addition, KNN can be computationally expensive for large data sets, as the distance between the test point and all the points in the training data must be calculated. Finally, KNN is susceptible to the curse of dimensionality, meaning that it can perform poorly in high-dimensional spaces.

Figure 1.1 demonstrates the operation of KNN. The illustration includes two training classes, Class A and Class B, and an unclassified triangle. If K is set to 1, the KNN will assign the triangle to Class A due to its proximity to that class. If K is set to 2, there is a tie between Class A and Class B, and one possible solution is to classify the triangle based on the closest class. If K is set to 3, the KNN will assign the triangle to Class B, as it is the class represented by the majority of the three nearest neighbors (2 belonging to Class B and 1 belonging to Class A).

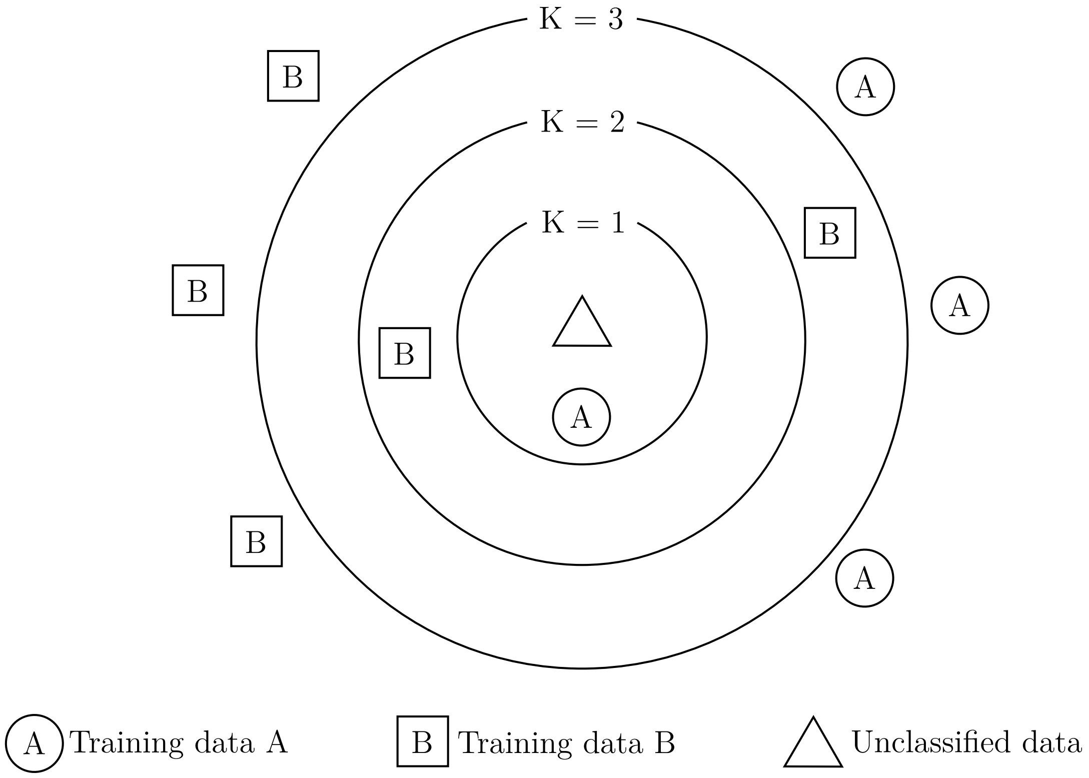


Figure 1.1. Example of the K values in KNN classification.
© 2020 IEEE.

In Figure 1.2 and Figure 1.3, close-up views of specific waveforms are provided. Each waveform is depicted with its sampled values highlighted. These sampled values, essential for signal analysis, can be systematically arranged into arrays.

The sample values of the waveforms x_1, x_2, and x_3 presented in Figure 1.2 are organized into arrays, as illustrated in Figure 1.4. Similarly, the sampled values of the waveforms y_1, y_2, and y_3 in Figure 1.3 are arranged into arrays, as shown in Figure 1.5.

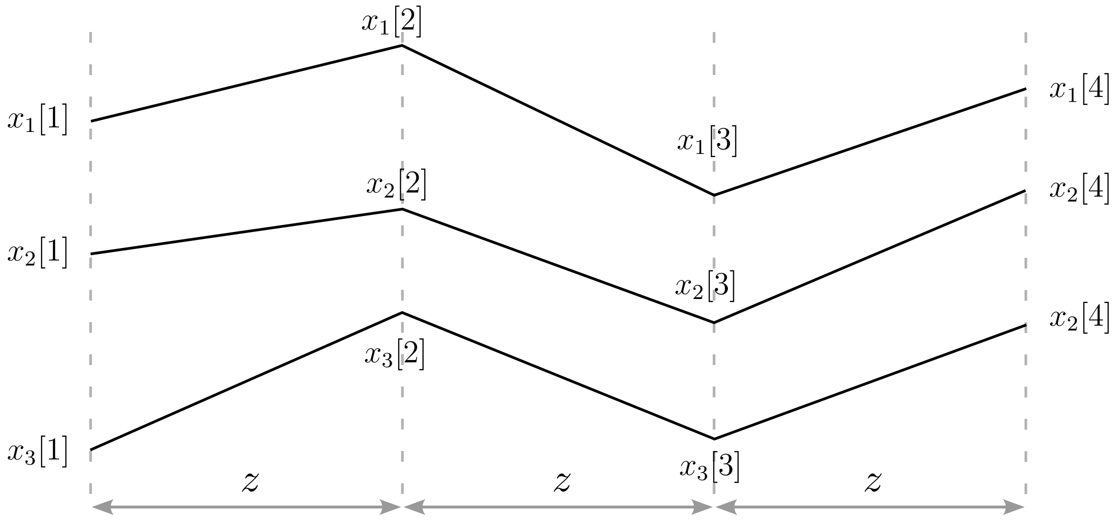


Figure 1.2. Zoomed-in view of the x waveforms showing specific sampled values.

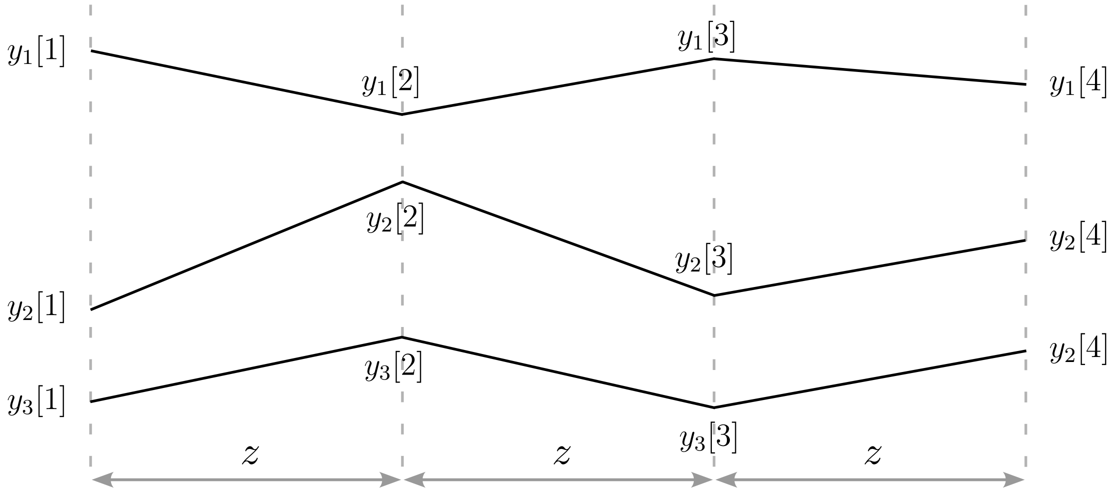


Figure 1.3. Zoomed-in view of the y waveforms showing specific sampled values.

$$
\begin{aligned}
x_1 &= [x_1[1], x_1[2], x_1[3], x_1[4]] \\
x_2 &= [x_2[1], x_2[2], x_2[3], x_2[4]] \\
x_3 &= [x_3[1], x_3[2], x_3[3], x_3[4]] \\
x_4 &= [x_4[1], x_4[2], x_4[3], x_4[4]]
\end{aligned}
$$

Figure 1.4. Arrays representation of the sampled values from the x waveform.

$$\begin{aligned}
y_1 &= [y_1[1], y_1[2], y_1[3], y_1[4]] \\
y_2 &= [y_2[1], y_2[2], y_2[3], y_2[4]] \\
y_3 &= [y_3[1], y_3[2], y_3[3], y_3[4]] \\
y_4 &= [y_4[1], y_4[2], y_4[3], y_4[4]]
\end{aligned}$$

Figure 1.5. Arrays representation of the sampled values from the y waveform.

1.2.2.3. Distance Formulas Used in PCA-KNN

In the pursuit of classifying individual bird, a range of distance metrics, including Cityblock, Euclidean, Correlation, and Cosine, is employed.

- Cosine distance:

$$d = 1 - \frac{\sum_{n=1}^{N} (x_{j[n]})(y_{i[n]})}{\sqrt{\sum_{n=1}^{N} (x_{j[n]})^2}\sqrt{\sum_{n=1}^{N} (y_{i[n]})^2}} \tag{1.1}$$

- Correlation distance:

$$d = 1 - \frac{\sum_{n=1}^{N} (y_{i[n]} - \overline{y}_i)(x_{j[n]} - \overline{x}_j)}{\sqrt{\sum_{n=1}^{N} (y_{i[n]} - \overline{y}_i)^2}\sqrt{\sum_{n=1}^{N} (x_{j[n]} - \overline{x}_j)^2}} \tag{1.2}$$

$$\text{where } \overline{x}_j = \frac{1}{N}\sum_{n=1}^{N} x_{j[n]}$$

$$\text{and } \overline{y}_i = \frac{1}{N}\sum_{n=1}^{N} y_{i[n]}$$

- Cityblock distance:

$$d = \sum_{n=1}^{N} \mid y_{i[n]} - x_{j[n]} \mid \tag{1.3}$$

- Euclidean distance:

$$d = \sqrt{\sum_{n=1}^{N} (y_{i[n]} - x_{j[n]})^2} \tag{1.4}$$

As an example, the Euclidean distance defined in Equation 1.8 can be used to compute the distance between y_3 from Figure 1.5 and x_2 from Figure 1.4.

An example using the Euclidean distance in Equation 1.8 to calculate the distance between y_3 in Figure 1.5 and x_2 in Figure 1.4:

$$d_{y_3-x_2} = \sqrt{\sum_{n=1}^{4} (y_3[n] - x_2[n])^2}$$

1.2.2.4. PCA-KNN

Figure 1.6(a) displays a tabular representation of a dataset poised for dimensionality reduction via PCA. Two primary columns, x and y, represent the features of the dataset across four distinct observations. These observations are categorized into two training classes: “training class 1” encapsulates observations 1 and 2, while “training class 2” contains observations 3 and 4. Subsequently, the dataset is concisely encoded as matrix A in Figure 1.6(b). Each row in matrix A aligns with an observation vector consisting of x and y feature values. This matrix construct serves as a standardized mathematical model for datasets undergoing

PCA, facilitating the transformation from a 2-dimensional feature space to a 1-dimensional feature space.

	x	y
Observation 1, training class 1	x_1	y_1
Observation 2, training class 1	x_2	y_2
Observation 3, training class 2	x_3	y_3
Observation 4, training class 2	x_4	y_4

(a)

$$A = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{bmatrix}$$

(b)

Figure 1.6. Example of PCA - (a) Data information. (b) The matrix contains only the features of the data.

Proceeding to Figure 1.7, the data encapsulated in Figure 1.6 is visualized in a Cartesian plane, with each point representing an individual observation's features x's and y's. This scatter plot delineates the observations according to their respective training classes. The eigenvector of the covariance matrix A, labeled E, represents the first principal component in PCA and indicates the direction of maximum data variance. The features of the training classes are projected onto this eigenvector, facilitating a dimensionality reduction while retaining significant data characteristics.

The same projection process is applied to the features of the unknown class. Subsequently, KNN classification is carried out by computing the distances between the projected features of the unknown class and those of the training classes along the eigenvector. This method allows for the classification of the unknown data point by identifying its nearest neighbors in the feature space. The implementation of this classification strategy is further illustrated in Figure 1.8.

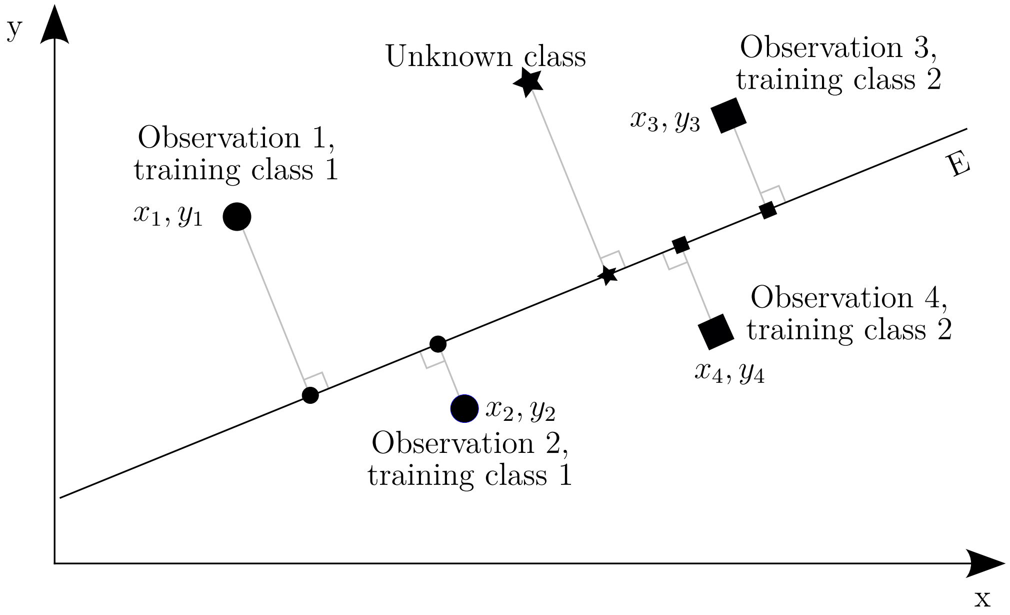


Figure 1.7. An example of using PCA to reduce the dimensionality from two to one. The data, originally in two dimensions, are now in one dimension after being projected onto the eigenvector.

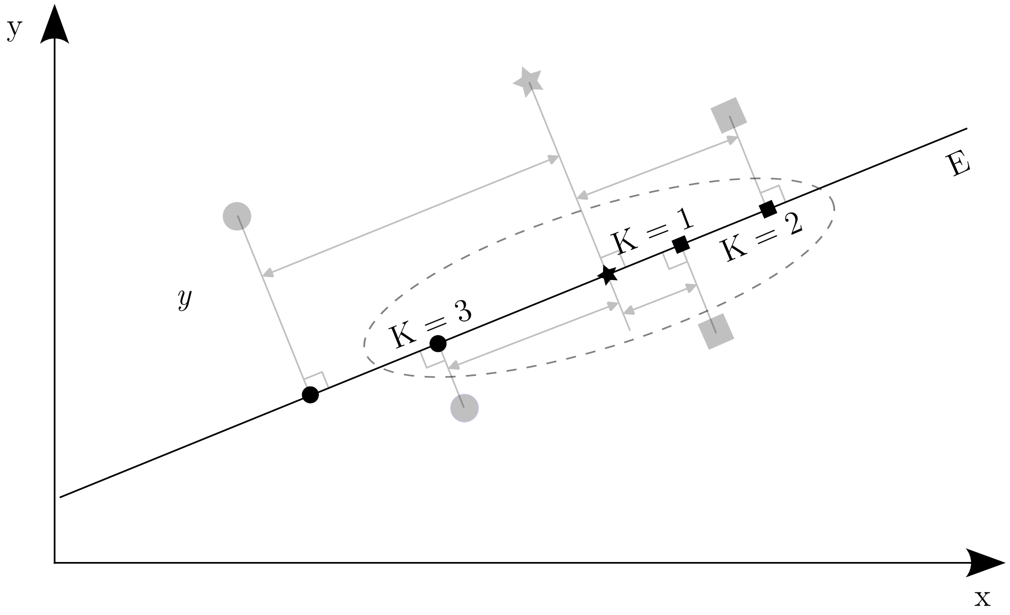


Figure 1.8. After the data dimensions are reduced from two to one using the eigenvector, KNN is utilized along the same eigenvector for classification.

1.2.3. Method 2: Cross-Correlation for Distinguishing Individual Birds

In the PCA-KNN method, manual alignment of each flight waveform is required, involving meticulous adjustment to ensure that all start points match. Conversely, the Cross-Correlation method eliminates the need for this manual alignment. This technique offers an efficient solution, particularly useful in situations where the data is not easily separable or when the features of the dataset are not well-defined.

1.2.3.1. Cross-Correlation

The Cross-Correlation method aims to identify the closest match using distance metric formulas. This involves padding zeros to the training dataset and padding zeros to the testing dataset. The calculation to find the closest match involves shifting elements, which are the sample values, within the testing dataset. For each shift, the distance metric formula is used to compute the distance between the testing data and all the training data.

Once the smallest distance is found, the testing data is assigned to the class with the shortest distance to the training data that belongs to that class. The shortest distance represents the closest match between the two datasets.

The downside to the Cross-Correlation method is that it requires a computer with fast computation capabilities. Because the method involves shifting and comparing large arrays of data, it can be computationally intensive and requires significant processing power. Nonetheless, the benefits of the method in terms of accuracy and reliability make it a worthwhile investment in cases where precise data matching is critical.

Figures 1.9, 1.10, 1.11, 1.12, and 1.13 provide a visual illustration of the concept. The figures depict two waves, namely Wave A in Figure 1.9 and Wave B in Figure 1.10. To determine the closest match between the two waves, Wave A is initially placed on the left side of Wave B, as shown in Figure 1.11. This position results in a low amount of similarity between the two waves.

Next, Wave A is slid towards the right. As Wave A is moved to the right, the degree of similarity between Wave A and Wave B increases. When Wave A reaches the position that best matches Wave B, as seen in Figure 1.12, this relative position results in the highest level of similarity between the two waves. In Figure 1.13, Wave A is placed to the right of Wave B, resulting in the least similarity between the two waves.

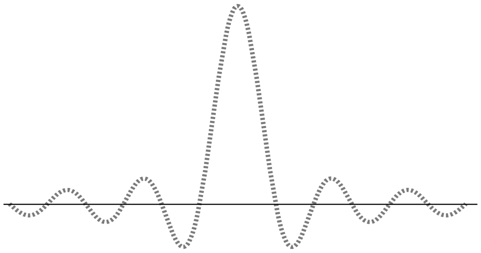


Figure 1.9. Wave A.

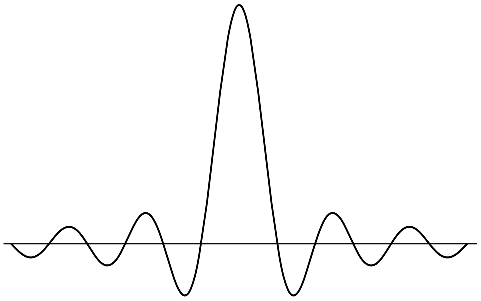


Figure 1.10. Wave B.

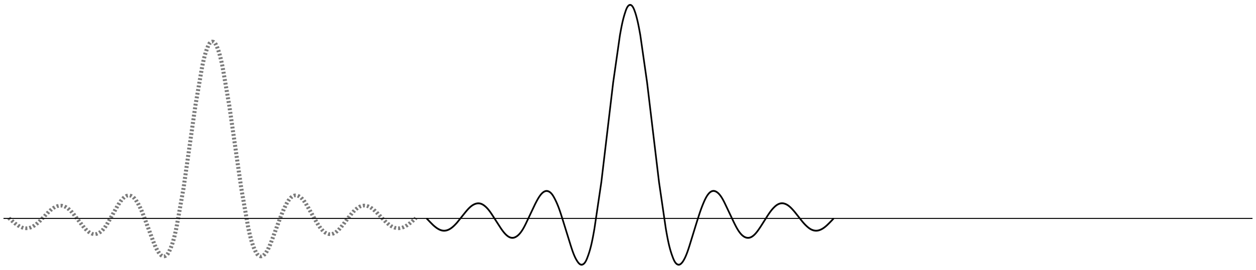


Figure 1.11. Wave A is on the left of wave B. This relative position gives the least similarity.

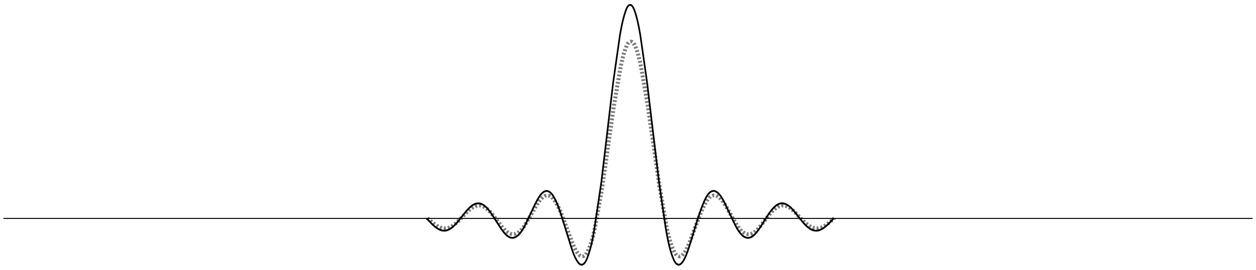


Figure 1.12. Wave A overlap wave B. This relative position gives the most similarity.

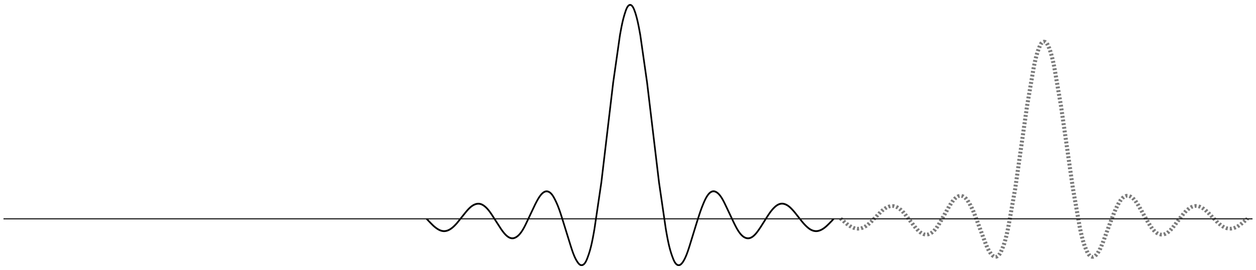


Figure 1.13. Wave A is on the right of wave B. This relative position gives the least similarity.

1.2.3.2. Distance Formulas Used in Cross-Correlation

- Cosine distance:

$$d_p = 1 - \frac{\sum_{n=1}^{N} (y_{i[n-Mp]})(x_{j[n]})}{\sqrt{\sum_{n=1}^{N} (y_{i[n-Mp]})^2} \sqrt{\sum_{n=1}^{N} (x_{j[n]})^2}} \tag{1.5}$$

- Correlation distance:

$$d_p = 1 - \frac{\sum_{n=1}^{N} (y_{i[n-Mp]} - \overline{y}_i)(x_{j[n]} - \overline{x}_j)}{\sqrt{\sum_{n=1}^{N} (y_{i[n-Mp]} - \overline{y}_i)^2} \sqrt{\sum_{n=1}^{N} (x_{j[n]} - \overline{x}_j)^2}} \tag{1.6}$$

$$\text{where } \overline{x}_j = \frac{1}{N}\sum_{n=1}^{N} x_{j[n]}$$

$$\text{and } \overline{y}_i = \frac{1}{N}\sum_{n=1}^{N} y_{i[n-Mp]}$$

- Cityblock distance:

$$d_p = \sum_{n=1}^{N} \mid y_{i[n-Mp]} - x_{j[n]} \mid \tag{1.7}$$

- Euclidean distance:

$$d_p = \sqrt{\sum_{n=1}^{N}(y_{i[n-Mp]} - x_{j[n]})^2} \tag{1.8}$$

In all cross-correlation formulations, N denotes the total number of elements within an array. The parameter M specifies the magnitude of each individual shift, while p represents the number of times the array has been shifted. Let $\mathcal{D}$ denote the set of distances computed for each corresponding shift value m. The set $\mathcal{D}$ is defined in equation 1.9.

$$\mathcal{D} = \{d_0, d_1, d_2, d_3, \ldots, d_p\} \tag{1.9}$$

where the value of p is determined by the floor function of a specific expression, as shown in equation 1.10

$$p = \left\lfloor \frac{Z}{M} \right\rfloor \tag{1.10}$$

where Z denotes the total number of zeros padded to an array, and $Z \leq 2N$.

The illustrations presented in Figures 1.15 and 1.16 depict a process analogous to the sounds generated by wing flaps, with $M = 1$ and $Z = 6$.

$$A = \begin{bmatrix}5 & 7 & 3\end{bmatrix}$$
$$B = \begin{bmatrix}5 & 7 & 2\end{bmatrix}$$

Figure 1.14. Arrays A and B

The array A serves as the training data, while the array B is the testing data. To prepare array A for analysis, zeros are added to both the beginning and the end of the array. The number of zeros added to the beginning is equal to the total number of elements in the array, and the same number of zeros are added to the end. The resulting array is denoted by $ZA = [0\ 0\ 0\ 5\ 7\ 3\ 0\ 0\ 0]$, which has a length of 9. Similarly, zeros are added to the end of array B to match the length of ZA. In this case, the number of zeros added to the end is twice the total number of elements in the array. The resulting array is denoted by $ZB = [5\ 7\ 2\ 0\ 0\ 0\ 0\ 0\ 0]$, which also has a length of 9.

$$ZA = [0\ \ 0\ \ 0\ \ 5\ \ 7\ \ 3\ \ 0\ \ 0\ \ 0]$$
$$ZB = [5\ \ 7\ \ 2\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0]$$

Figure 1.15. Alignment of the arrays ZA and ZB through zero-padding.

In Figure 1.16, the relative shifting of the elements in array ZB compared to those in array ZA is displayed. The shifting process is visualized as a series of iterations, where after each iteration, the first position in the array is filled with a zero. This shifting process will be repeated a total of six times. The number of shifts is determined by the equation 1.10.

The value of d_p in equation 1.11 represents the Euclidean distance between the two arrays with $p = 0, 1, 2, 3, 4, 5, 6$. The corresponding value for each p in $\mathcal{D} = \{d_0, d_1, d_2, d_3, d_4, d_5, d_6\}$.

$$d_p = \sqrt{\sum_{n=1}^{9} (ZA_n - ZB_{n-Mp})^2} \tag{1.11}$$

The Euclidean distance is a measure of the distance between two points in a multidimensional space and can be used to quantify the degree of dissimilarity between two arrays. In the context of this example, the Euclidean distance between array ZA and ZB is a metric that can help assess the degree of similarity between the two arrays after the iterative shifting process. A smaller distance indicates a higher degree of similarity between the arrays, while a larger distance indicates a lower degree of similarity.

From Figure 1.16, $\mathcal{D} = \{12.5, 11.3, 6.7, 1, 7.7, 11.5, 12.5\}$. The smallest number in $\mathcal{D}$ is 1, which corresponds to the position where the two arrays A and B are lined up.

The example demonstrating the cross-correlation between the two arrays ZA and ZB illustrates the application of cross-correlation in identifying similarities between two arrays.

ZA	0	0	0	5	7	3	0	0	0	$d_0 = 12.5$
ZB	4	7	3	0	0	0	0	0	0	

ZA	0	0	0	5	7	3	0	0	0	$d_1 = 11.3$
ZB	0	4	7	3	0	0	0	0	0	

ZA	0	0	0	5	7	3	0	0	0	$d_2 = 6.7$
ZB	0	0	4	7	3	0	0	0	0	

ZA	0	0	0	5	7	3	0	0	0	$d_3 = 1$
ZB	0	0	0	4	7	3	0	0	0	

ZA	0	0	0	5	7	3	0	0	0	$d_4 = 7.7$
ZB	0	0	0	0	4	7	3	0	0	

ZA	0	0	0	5	7	3	0	0	0	$d_5 = 11.5$
ZB	0	0	0	0	0	4	7	3	0	

ZA	0	0	0	5	7	3	0	0	0	$d_6 = 12.5$
ZB	0	0	0	0	0	0	4	7	3	

Figure 1.16. Example of two arrays with shifting and the similarity d_p using Euclidean distance.

2. DATA COLLECTION AND PREPROCESSING

2.1. Equipment Setup for Audio Recording

The equipment utilized in the experiment is itemized in Table 2.1. A visual depiction of the custom-built cage utilized during the experiment is provided in Figure 2.1. To provide additional context, Figure 2.2 provides an end view schematic of the cage, illustrating the placement of microphones “MP1” and “MP2” positioned above the floor level. Figure 2.3 provides an overhead view schematic of the cage, delineating the microphone placements and the USB webcam location. “MP1” indicates the microphone positioned to capture wing flap audio when the bird traverses between perches, while “MP2” designates the microphone’s location for acquiring sound data as the bird flies towards or away from the perch. “VC” denotes the fixed location of the USB webcam, strategically placed to monitor the birds’ flight paths and perch interactions.

Table 2.1. List of equipment.

Equipment	Manufacture	Manufacture Product Name
USB microphone	Blue	Snowball
USB webcam	Logitech	C922 Pro
Laptop Computer	Gigabyte	Areo 15-Y9

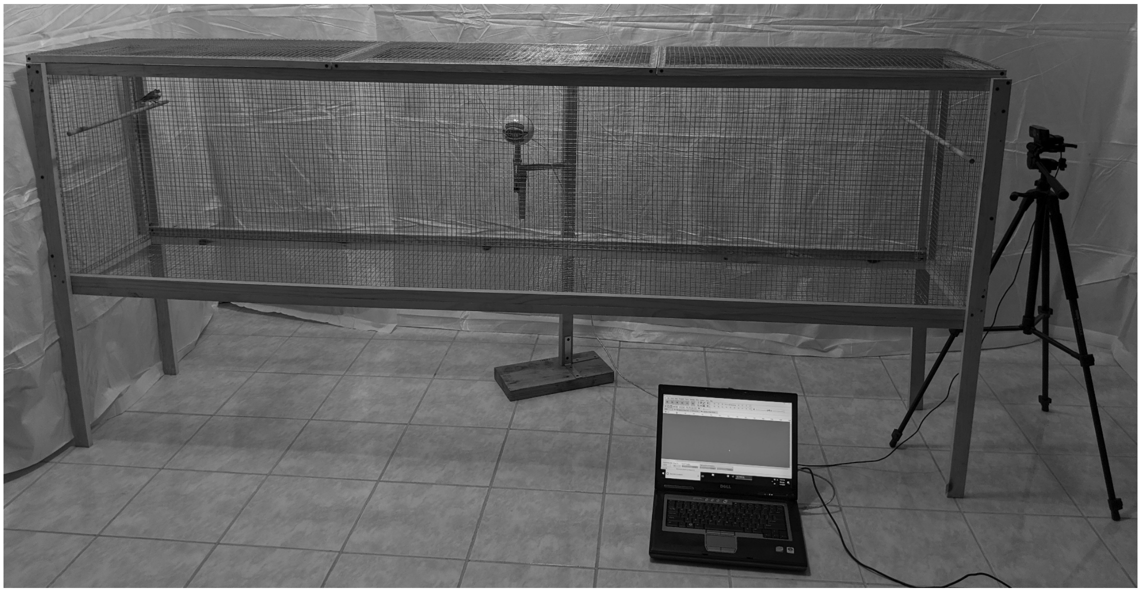


Figure 2.1. Our experiment's custom cage.

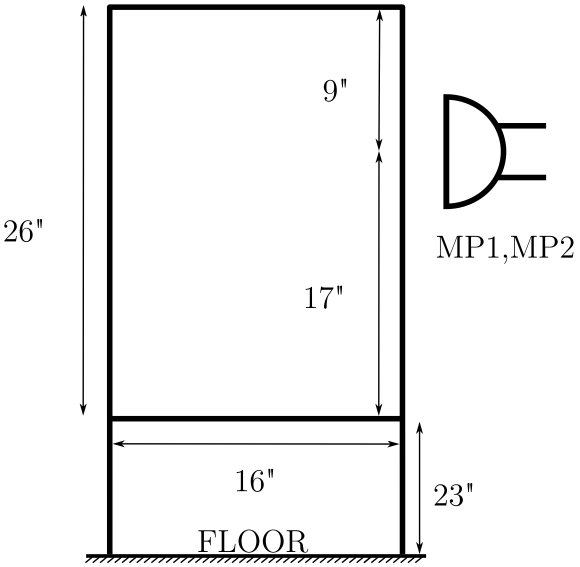


Figure 2.2. The dimension of the end of the cage, and the position of the microphone above the floor.

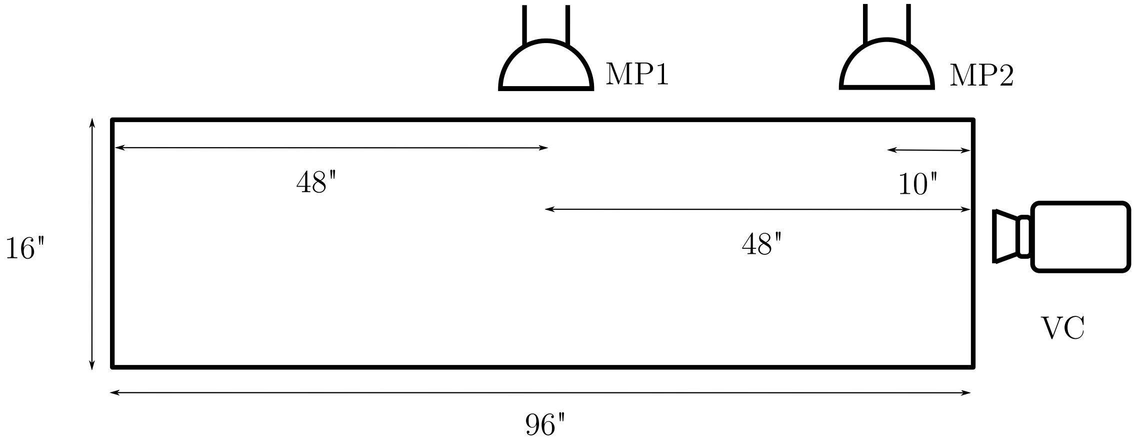


Figure 2.3. The dimension of the top of the cage, the two positions of the microphone, and the position of the camera.

2.2. Audio Data Preprocessing

In the process of audio data preprocessing for this thesis, a pivotal aspect was ensuring the ethical treatment of the subjects involved. The research conducted involved working with birds, specifically utilizing privately owned animals at a private residence. This approach adhered to the guidelines provided by North Dakota State University's Institutional Animal Care and Use Committee (NDSU IACUC), which determined that this specific setup did not require formal IACUC review or approval.

Following these ethical considerations, the study focused on three Zebra Finches, individually named Curly, Moe, and Larry. To help readers visualize these birds, Figure 2.4 displays an image of Curly, Moe, and Larry. To capture their wing flap sounds, individual recordings were made for each bird, with a sampling frequency of 44100Hz. The recordings were made using Audacity software, which is a popular audio recording and editing program. Figure 2.6 shows a waveform of the wing flap sounds recorded using Audacity. In Figure 2.7, the manual selection of wing flap sounds is displayed, with the gray area representing the selected portion of the waveform.

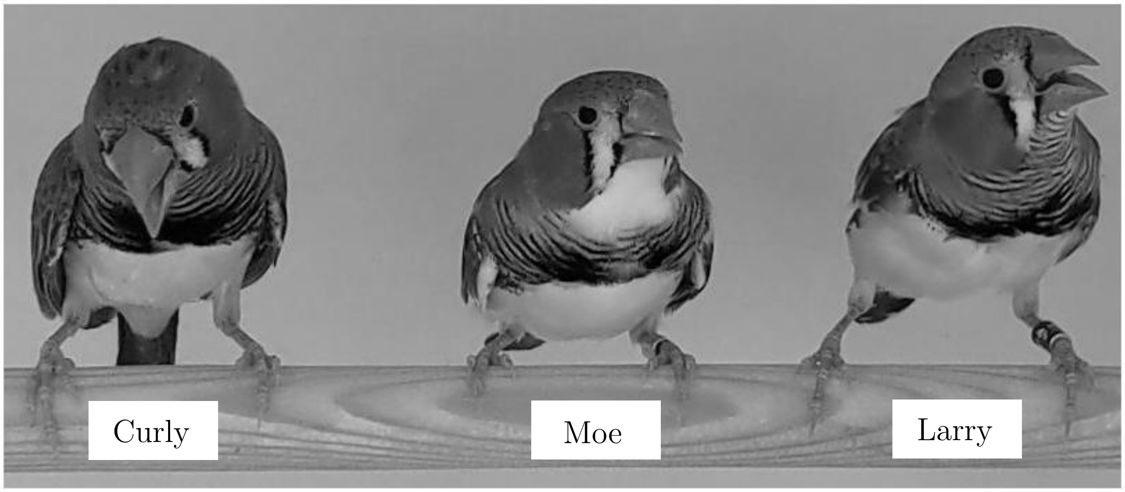


Figure 2.4. The three birds used for this experiment.

Each bird was recorded for 300 flights, with 100 flights recorded for each type of flight, including leaving the perch, flying between the perches, and approaching the perch. This ensured that the data captured a range of flight patterns exhibited by the birds.

In total, the study recorded 900 flights across the three birds. To train a flight classifier that could distinguish between the different flight patterns, the researcher randomly selected 80 flights out of the 100 recorded flights for each type of flight for each bird as training data. The remaining 20 flights of each type of flight for each bird were used as test data.

This resulted in a total of 240 flights being used for training data for each type of flight for each bird, and a total of 720 flights being used in total for training data. The remaining 180 flights, 20 flights for each type of flight for each bird, were used for testing data.

During their flights, the Zebra Finches occasionally produced vocalizations. The frequency of these vocalizations fell within the range of 500Hz to 8kHz. The microphone recordings captured both the sound of the vocalizations and the sound of the wing flaps, which had a frequency range of 25.2Hz to 31.9Hz.

To isolate the wing flap sounds in the audio recordings, vocalizations produced by the birds were attenuated. This was accomplished through the application of a finite impulse response (FIR) Equiripple low-pass filter, designed using MATLAB. The filter specifications included an order of 124, a passband frequency of 100 Hz, a stopband frequency of 1 kHz, a passband ripple of 1 dB, and a stopband attenuation of 80 dB. An FIR filter was selected due to its advantageous properties, such as the ability to produce minimal waveform distortion. The magnitude response of the implemented low-pass filter is depicted in Figure 2.5. Subsequent to the removal of vocal sounds, the audio recordings were rescaled, ensuring that all data points were within the range of -1 to 1.

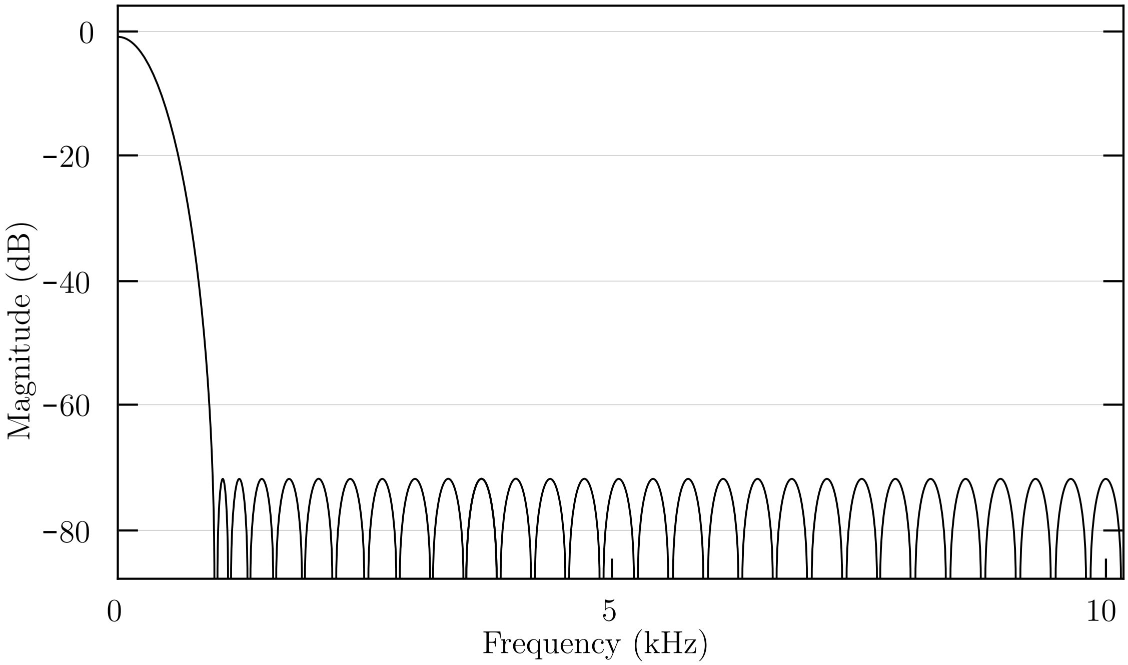


Figure 2.5. Magnitude response of the 100Hz lowpass filter.

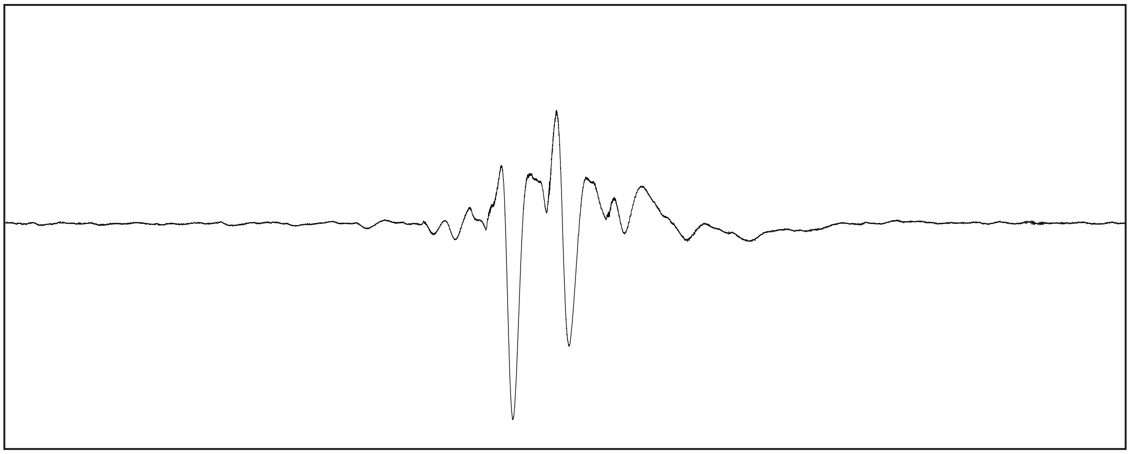
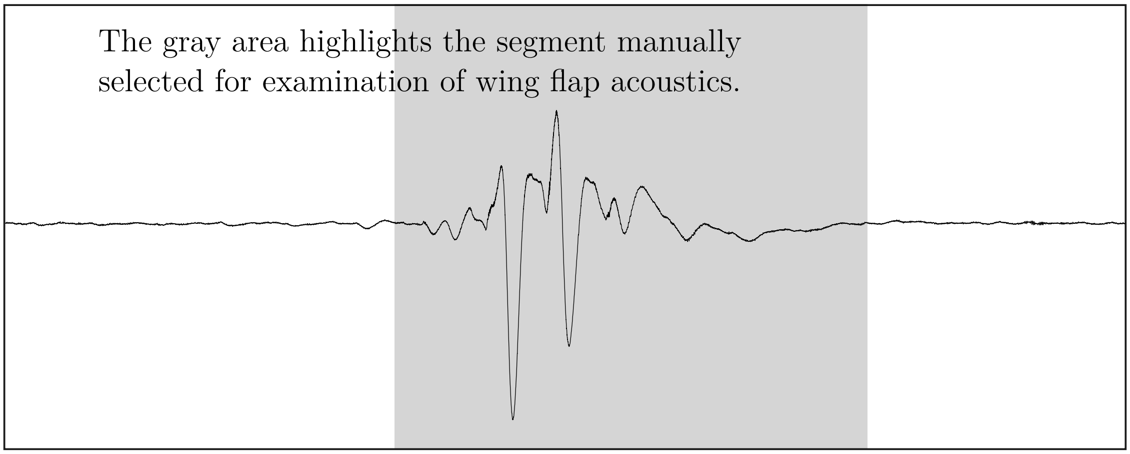


Figure 2.6. Sample wing flap sounds recorded using Audacity.



The gray area highlights the segment manually selected for examination of wing flap acoustics.

Figure 2.7. Audio waveform showcasing manually selected wing flap sounds for analysis, as processed in Audacity.

In Figures 2.8, 2.9, and 2.10, all audio plots were manually lined up and scaled from -1 to 1, with vocal sounds filtered out. Figure 2.8 displays twenty audio plots capturing the sounds of each bird leaving the perch, while Figure 2.9 shows twenty audio plots recording the sounds of each bird in flight between perches. Figure 2.10 presents twenty audio plots of each bird approaching the perch. These figures offer a comprehensive and detailed look at the wing flap sounds of individual birds during different stages of their movements, providing deeper insights into the process of individual recognition.

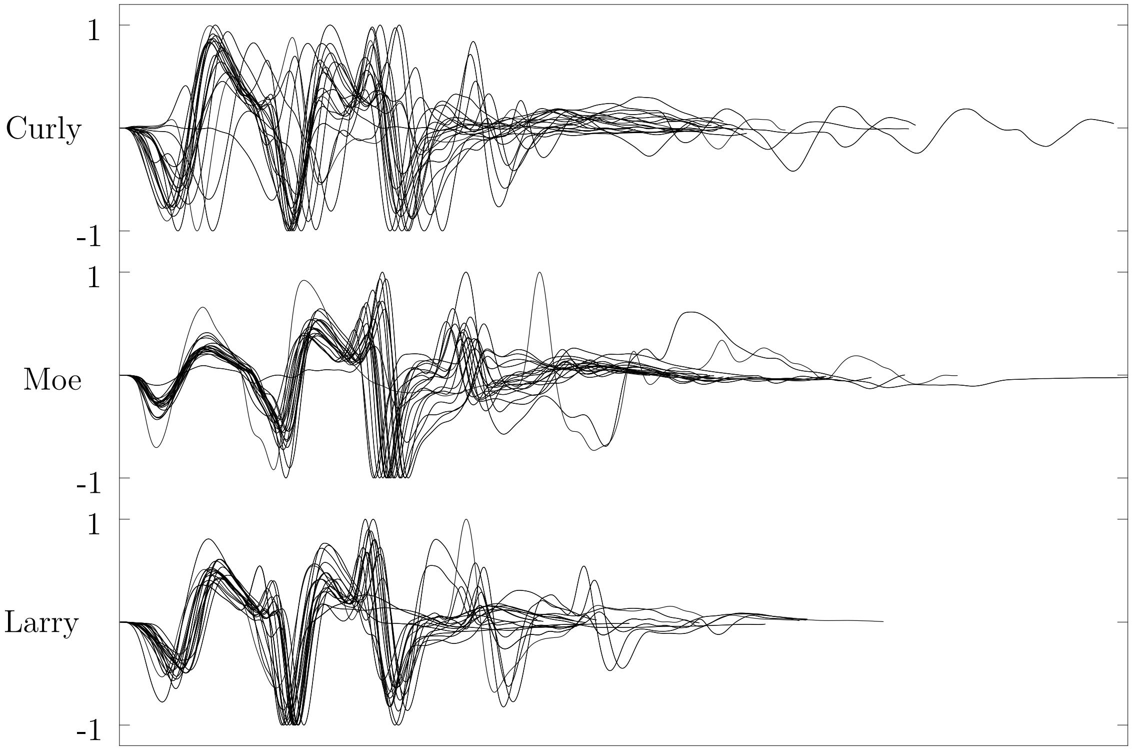


Figure 2.8. Twenty audio plots of the sounds of each bird leaving the perch.

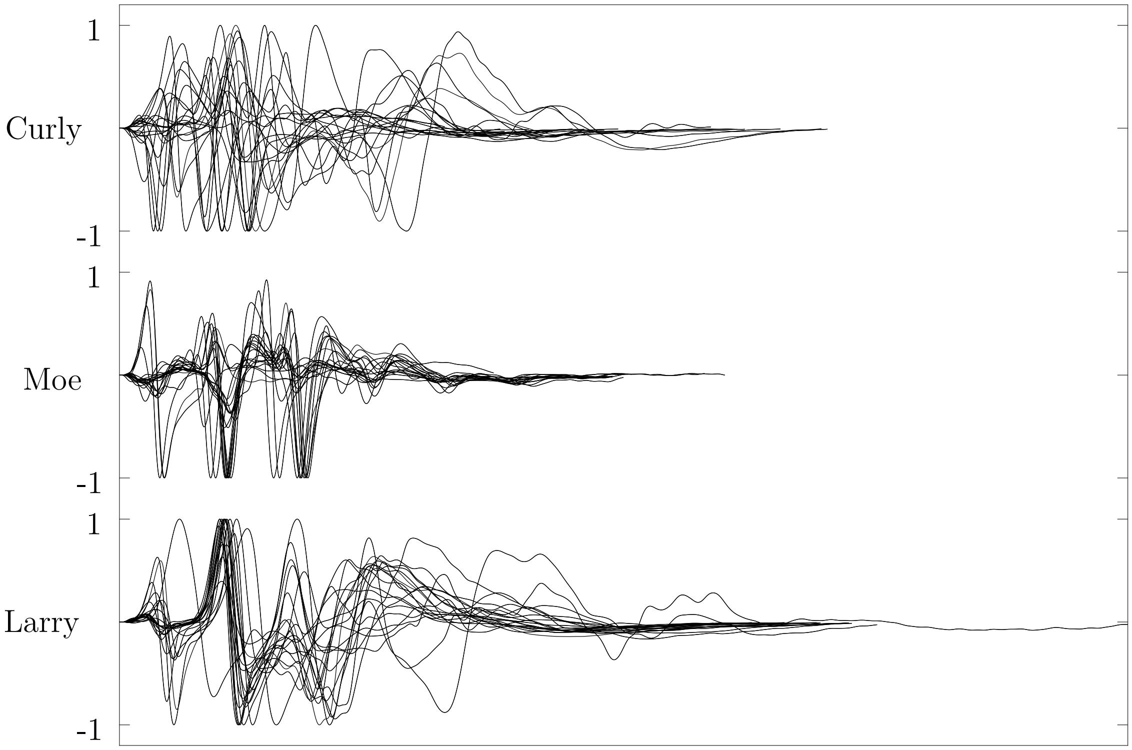


Figure 2.9. Twenty audio plots record the sounds of each bird as it flies between perches.

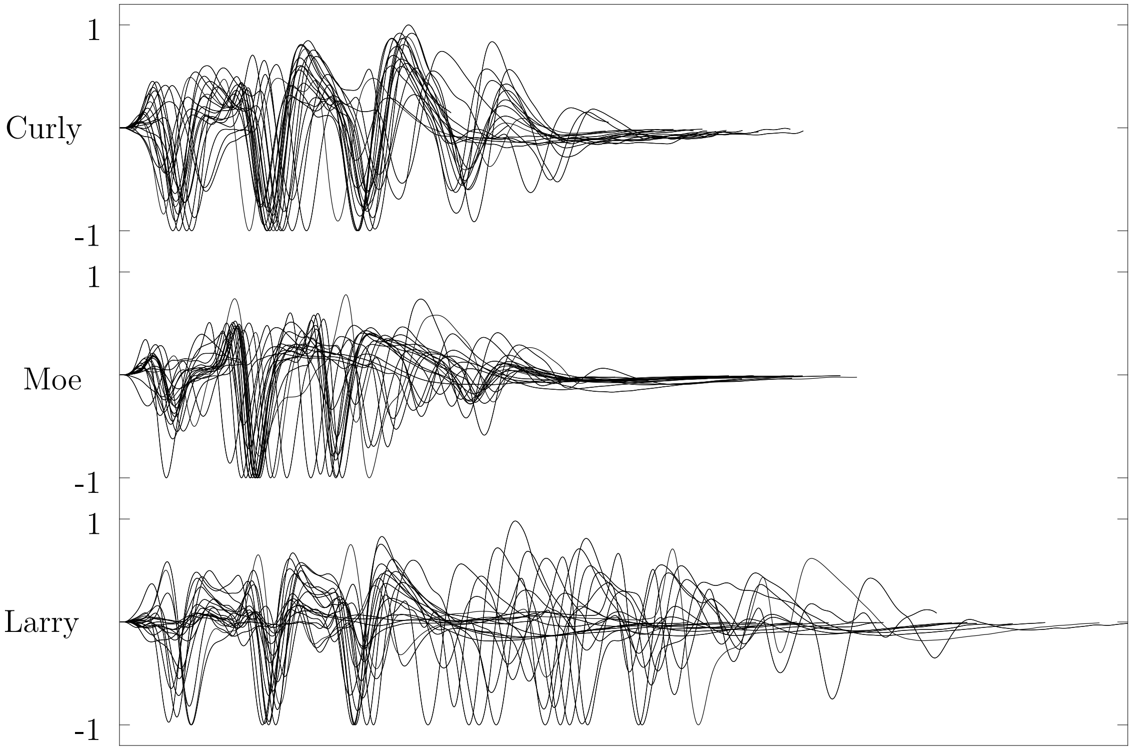


Figure 2.10. Twenty audio plots of the sounds of each bird approaching the perch.

3. INDIVIDUAL CLASSIFICATION OF BIRDS: METHODOLOGY AND RESULTS

Each bird exhibits distinct wing flap acoustic signatures associated with different flight behaviors, such as leaving the perch, flying in a straight line between two perches, and approaching the perch. Both PCA-KNN and Cross-Correlation methods have demonstrated reasonable effectiveness in classifying not only individual birds but also in accurately identifying these specific flight behaviors. This innovative approach leverages the unique acoustic signatures of birds to successfully distinguish between various flight patterns.

Confusion matrices serve as a fundamental tool for analyzing the research results, particularly in quantitatively assessing the accuracy and precision of various classification methods. These matrices are crucial for visualizing and understanding how effectively the sound-based bird flight recognition system performs. They are structured such that the rows represent the predictions made by the system, and the columns correspond to the actual bird flight classes. The matrices clearly delineate correct detections, which appear on the diagonal where predicted and actual classes align, and misclassifications, which are indicated by off-diagonal elements.

To provide a comprehensive overview of the system's capabilities in various scenarios, this thesis employs confusion matrices in two distinct yet interrelated contexts for bird recognition. This includes analyses both with and without consideration of flight directions. Specifically, in scenarios accounting for flight directions, the confusion matrices are 9×9 matrices. The columns and rows of these matrices are labeled with abbreviations such as CuL, CuS, CuA, MoL, MoS, MoA, LaL, LaS, and LaA. 'CuL' denotes Curly leaving the perch, ‘CuS’ represents Curly flying in a straight line between two perches, and ‘CuA’ signifies Curly approaching the perch. Similar labels are used for Moe (‘MoL’, ‘MoS’, ‘MoA’) and Larry (‘LaL’, ‘LaS’, ‘LaA’). In contrast, for scenarios without consideration of flight direction, the confusion matrices are 3×3 matrices, derived by aggregating data from the 9×9 matrices. The resultant matrices are labeled ‘Cu’, ‘Mo’, and ‘La’, abbreviating Curly, Moe, and Larry, respectively. This dual approach in using confusion matrices, varying in size based on the level of detail considered, provides detailed insights into the system’s performance across various bird flight recognition scenarios.

In an effort to further enhance the analytical depth of this study, an additional row labeled ‘PA’, standing for Percentage of Accuracy, has been incorporated below the last row of each confusion matrix, both for matrices considering flying direction and those not considering flying direction. The inclusion of the PA row provides a quick, at-a-glance indicator of the overall accuracy of each scenario being analyzed.

3.1. Bird Classification Results From Using The Combination Of PCA-KNN

In the series of figures, each one is dedicated to a specific distance metric. Figure 3.1 focuses on the Cosine metric, 3.3 on the Cityblock metric, 3.2 on the Correlation metric, and 3.4 on the Euclidean metric. In all these figures, the vertical axis represents ‘Identification Accuracy,’ and the horizontal axis represents the ‘Number of Principal Components.’ Identification Accuracy is computed as the ratio of correct predictions to total predictions, defined as Identification Accuracy = (Number of Correct Predictions) / (Total Number of Test Instances). Each figure illustrates how identification accuracy varies with the number of principal components, ranging from 1 to 200, and examines this relationship across different K-values of 3, 5, 10, and 15.

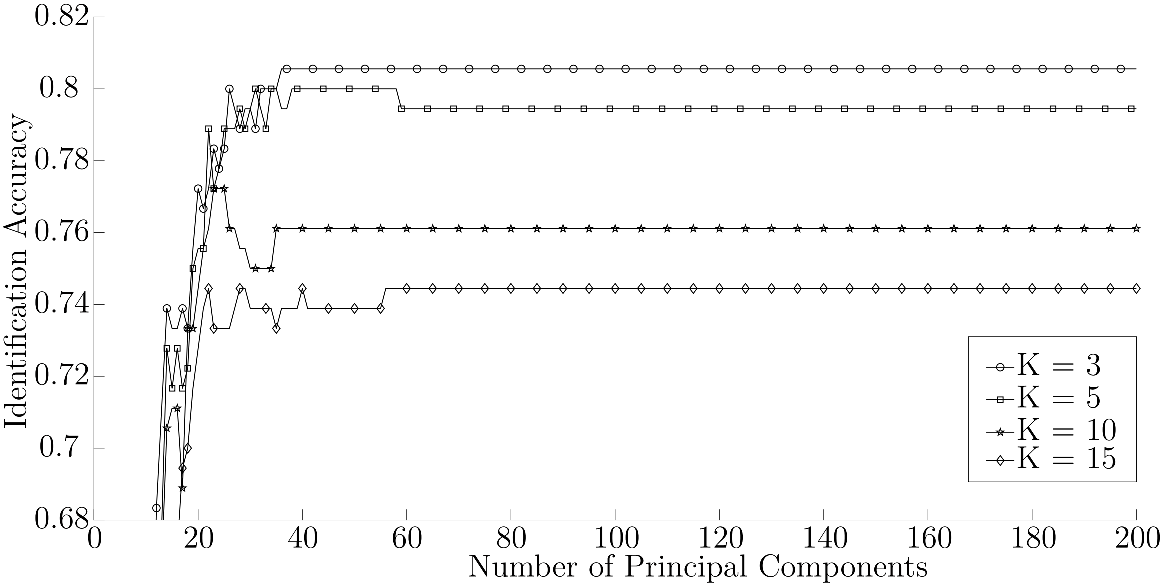


Figure 3.1. Variation of classification accuracy with Number of Principal Components for Cosine distance in PCA-KNN analysis.

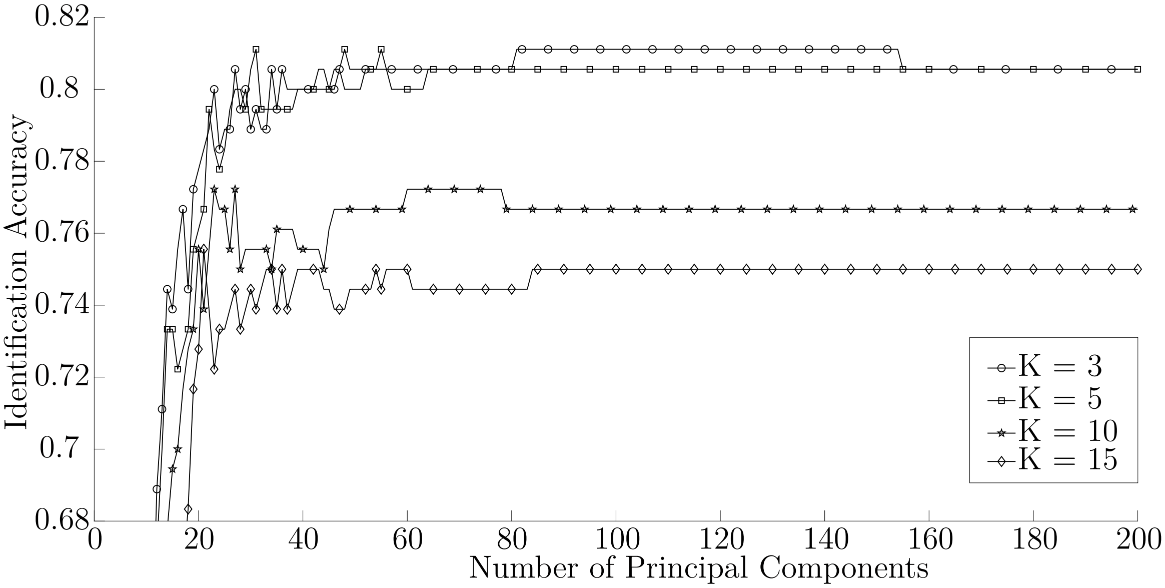


Figure 3.2. Variation of classification accuracy with Number of Principal Components for Correlation distance in PCA-KNN analysis.

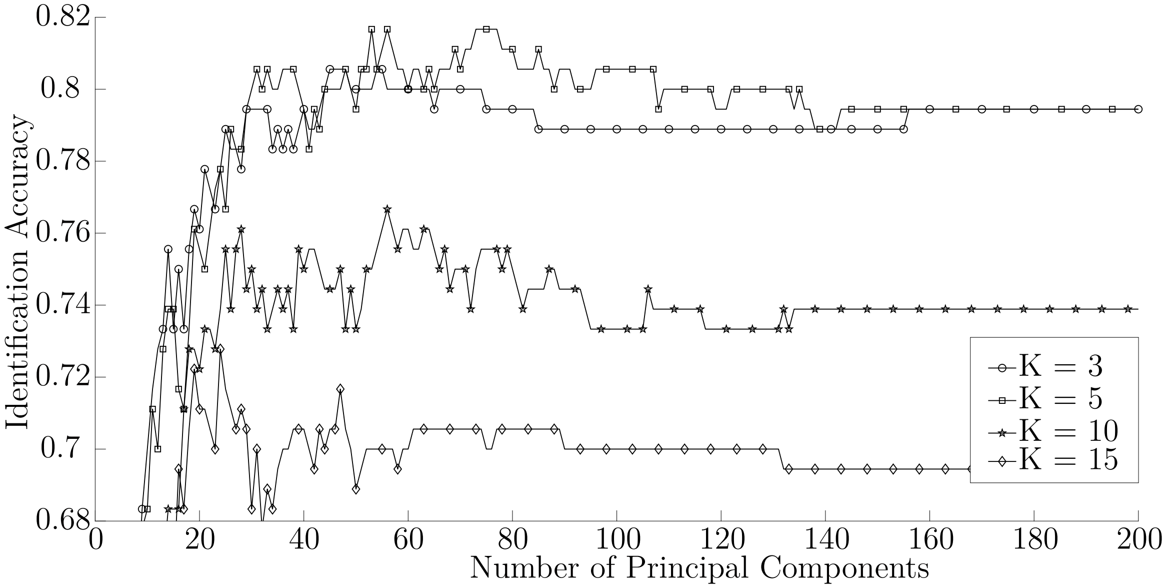


Figure 3.3. Variation of classification accuracy with Number of Principal Components for Cityblock distance in PCA-KNN analysis.

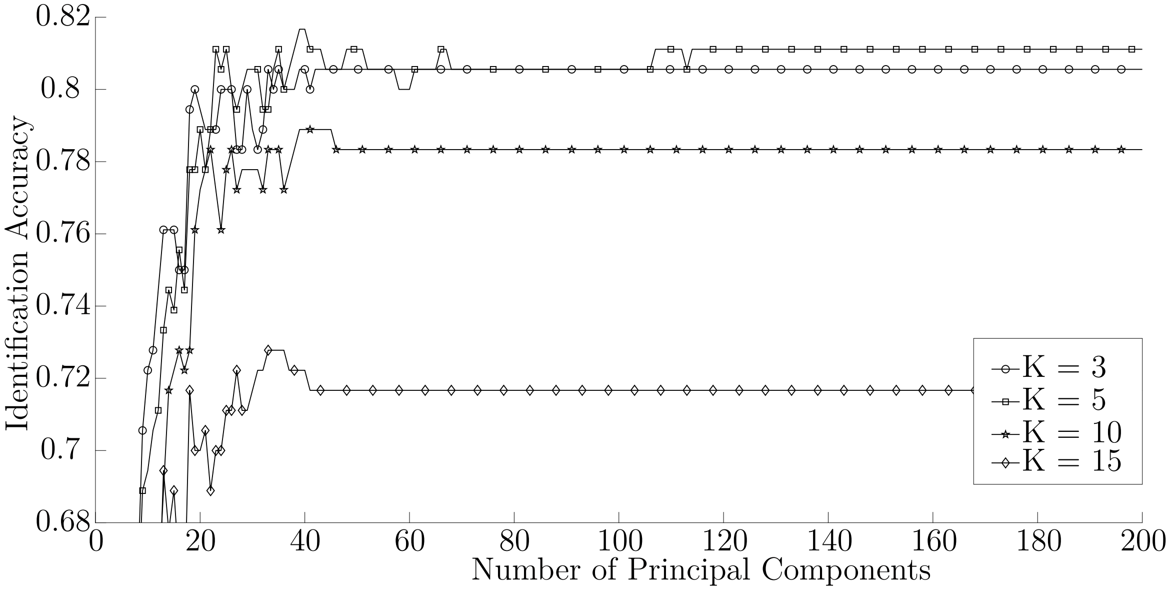


Figure 3.4. Variation of classification accuracy with Number of Principal Components for Euclidean distance in PCA-KNN analysis.

Table 3.1 showcases the highest classification accuracies for varying neighborhood sizes, captured by K values of 3, 5, 10, and 15. This table enumerates the maximum accuracy attained with each considered distance metric: Cosine, Correlation, Cityblock, and Euclidean. In addition, it details the specific number of principal components that are linked to these maximal accuracies. The highlighted rows represent the highest accuracy obtained for each metric. For the Correlation distance metric, while both K = 3 and K = 5 yield identical accuracies, the choice of K = 5 with 31 principal components is optimal. This configuration benefits from a more stable classification due to a higher K value while maintaining model simplicity and computational efficiency with fewer components.

Table 3.1. Optimal accuracy achieved with various distance metrics at different K values

Distance metric	K value	Components	Accuracy
Cosine	3	36	0.80556
Cosine	5	31	0.80000
Cosine	10	23	0.77222
Cosine	15	22	0.74444
Correlation	3	81	0.81111
Correlation	5	31	0.81111
Correlation	10	23	0.77222
Correlation	15	21	0.75556
Cityblock	3	45	0.80556
Cityblock	5	53	0.81667
Cityblock	10	56	0.76667
Cityblock	15	24	0.72778
Euclidean	3	33	0.80556
Euclidean	5	39	0.81667
Euclidean	10	39	0.78889
Euclidean	15	33	0.72778

Figures 3.5, 3.6, 3.7, and 3.8 each display the confusion matrices corresponding to various highlighted results within the Table 3.1. Regarding accuracy, the lowest observed accuracy is 55%. The highest accuracy is 100%. The overall average is 81%.

Predicted Flights	Actual Flights								
	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	16	4		2	1		2		
CuS		11		1	2				
CuA	1		17						
MoL	3			14					1
MoS		3			15				
MoA			2	1		19			
LaL				1		1	18	1	1
LaS		1			1			18	1
LaA		1	1	1	1			1	17
PA	80%	55%	85%	70%	75%	95%	90%	90%	85%

Figure 3.5. Confusion matrix for recognizing birds in flight directions using PCA-KNN with the Cosine distance metric.

Predicted Flights	Actual Flights								
	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	17	3		2			1		
CuS		12		1	1				
CuA			17						1
MoL	2			13			1		1
MoS		3			17				
MoA			2	1		19	1		
LaL	1			1		1	17	1	1
LaS		1		1	1			18	1
LaA		1	1	1	1			1	16
PA	85%	60%	85%	65%	85%	95%	85%	90%	80%

Figure 3.6. Confusion matrix for recognizing birds in flight directions using PCA-KNN with the Correlation distance metric.

	Actual Flights								
Predicted Flights	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	14	3		1	1				
CuS	1	14	1	1	3				
CuA	3		17						
MoL	2			14					
MoS		2		1	14				
MoA			1	1		19			
LaL				1		1	19	1	1
LaS					1			18	1
LaA		1	1	1	1		1	1	18
PA	70%	70%	85%	70%	70%	95%	95%	90%	90%

Figure 3.7. Confusion matrix for recognizing birds in flight directions using PCA-KNN with the Cityblock distance metric.

	Actual Flights								
Predicted Flights	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	14	2			1				
CuS	1	14		1	3				
CuA	1		17						
MoL	3			13					
MoS		2			14				
MoA			2	2		19			
LaL	1			2		1	20	1	1
LaS		1		1	1			18	1
LaA		1	1	1	1			1	18
PA	70%	70%	85%	65%	70%	95%	100%	90%	90%

Figure 3.8. Confusion matrix for recognizing birds in flight directions using PCA-KNN with the Euclidean distance metric.

Figures 3.9, 3.10, 3.11, and 3.12 each display a confusion matrix for the classification of birds' flight without considering the direction of flight. These matrices correspond to the Cosine, Correlation, Cityblock, and Euclidean distance metrics, respectively. Analysis of the matrices that exclude flying direction reveals a range of classification accuracy: the minimum observed accuracy is 80%, the maximum is 100%, and the overall average is 87%

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	49	6	2
	Mo	8	49	1
	La	3	5	57
	PA	82%	82%	95%

Figure 3.9. Confusion matrix for recognizing birds using PCA-KNN with the Cosine distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	49	4	2
	Mo	7	50	3
	La	4	6	55
	PA	82%	83%	92%

Figure 3.10. Confusion matrix for recognizing birds using PCA-KNN with the Correlation distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	53	6	
	Mo	5	49	
	La	2	5	60
	PA	88%	82%	100%

Figure 3.11. Confusion matrix for recognizing birds using PCA-KNN with the Cityblock distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	49	5	
	Mo	7	48	
	La	4	7	60
	PA	82%	80%	100%

Figure 3.12. Confusion matrix for recognizing birds using PCA-KNN with the Euclidean distance metric, without considering flight directions.

Table 3.2. Average recognition percentages from PCA-KNN using different distance metrics, with highest and lowest averages.

	Percent recognition					
	Avr		Lowest avr		Highest avr	
Distance	WD	WOD	WD	WOD	WD	WOD
Cosine	80.5	86.3	80.5			
Correlation	81.1	85.6		85.6		
Cityblock	81.6	90			81.6	90
Euclidean	81.6	87.3			81.6	

Avr/avr - average. WD - With Direction - the recognitions of bird flying with direction. WOD - Without Direction - the recognition is narrowed to birds only, without direction.

3.2. Bird Classification Results From Using Cross-Correlation

Figures 3.13, 3.14, 3.15, and 3.16 each display a confusion matrix, illustrating the classification outcomes for recognizing birds' flight considering their direction. These outcomes are analyzed using Cosine, Correlation, Cityblock, and Euclidean distance metrics, respectively. The matrices reveal a range of accuracies, with the lowest being 60% and the highest reaching 100%. The overall average is 86%. Notably, the Cosine and Correlation metrics yield similar recognition rates. The reasons for these similar rates will be explored in the Discussion section.

		Actual Flights								
		CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
Predicted Flights	CuL	12						1		
	CuS		16			1				
	CuA	3		17			1			3
	MoL	2			19			1		
	MoS		3			18		1		
	MoA				1	1	17			1
	LaL	3						17		
	LaS		1						20	
	LaA			3			2			16
	PA	60%	80%	85%	95%	90%	85%	85%	100%	80%

Figure 3.13. Confusion matrix for recognizing birds in flight directions using Cross-Correlation with the Cosine distance metric.

Predicted Flights	Actual Flights								
	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	12						1		
CuS		16			1				
CuA	3		17			1			3
MoL	2			19			1		
MoS		3			18		1		
MoA				1	1	17			1
LaL	3						17		
LaS		1						20	
LaA			3			2			16
PA	60%	80%	85%	95%	90%	85%	85%	100%	80%

Figure 3.14. Confusion matrix for recognizing birds in flight directions using Cross-Correlation with the Correlation distance metric.

Predicted Flights	Actual Flights								
	CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
CuL	13						1		
CuS		18	1				1	1	
CuA	3		15						1
MoL	1			20					
MoS		1			19				
MoA					1	20			1
LaL	3						18		
LaS		1						19	
LaA			4						18
PA	65%	90%	75%	100%	95%	100%	90%	95%	90%

Figure 3.15. Confusion matrix for recognizing birds in flight directions using Cross-Correlation with the Cityblock distance metric.

		Actual Flights								
		CuL	CuS	CuA	MoL	MoS	MoA	LaL	LaS	LaA
Predicted Flights	CuL	13						1		
	CuS		17	1						
	CuA	3		16						2
	MoL	1			19					
	MoS		2			19		1		
	MoA					1	20			2
	LaL	3			1			18		
	LaS		1						20	
	LaA			3						16
	PA	65%	85%	80%	95%	95%	100%	90%	100%	80%

Figure 3.16. Confusion matrix for recognizing birds in flight directions using Cross-Correlation with the Euclidean distance metric.

Each of the confusion matrices depicted in Figures 3.17, 3.18, 3.19, and 3.20 is focused on classifying birds' flight patterns while omitting the aspect of flight direction. These matrices are individually aligned with the Cosine, Correlation, Cityblock, and Euclidean distance metrics. A review of these matrices, which do not take flight direction into account, indicates a spectrum of accuracy in classification. The lowest recorded accuracy stands at 80%, the highest reaches 100%, and the overall average is 89%.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	48	2	4
	Mo	5	56	3
	La	7	2	53
	PA	80%	93%	88%

Figure 3.17. Confusion matrix for recognizing birds using Cross-Correlation with the Cosine distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	48	2	4
	Mo	5	56	3
	La	7	2	53
	PA	80%	93%	88%

Figure 3.18. Confusion matrix for recognizing birds using Cross-Correlation with the Correlation distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	50		4
	Mo	2	60	1
	La	8		55
	PA	83%	100%	92%

Figure 3.19. Confusion matrix for recognizing birds using Cross-Correlation with the Cityblock distance metric, without considering flight directions.

		Actual Flights		
		Cu	Mo	La
Predicted Flights	Cu	50		3
	Mo	3	59	3
	La	7	1	54
	PA	83%	98%	90%

Figure 3.20. Confusion matrix for recognizing birds using Cross-Correlation with the Euclidean distance metric, without considering flight directions.

Table 3.3. Average recognition percentages from Cross-Correlation using different distance metrics, with highest and lowest averages.

	Percent recognition					
	Avr		Lowest avr		Highest avr	
Distance	WD	WOD	WD	WOD	WD	WOD
Cosine	84.4	87	84.4	87		
Correlation	84.4	87	84.4	87		
Cityblock	88.9	91.7			88.9	91.7
Euclidean	87.7	90.3				

Avr/avr - average. WD - With Direction - the recognitions of bird flying with direction. WOD - Without Direction - the recognition is narrow to birds only, without direction.

4. DISCUSSION

Due to the subtle differences in the wing beat characteristics of individual birds, distinguishing among them poses a challenge. In response to this, the PCA-KNN and Cross-Correlation methods are employed. Both techniques are adept at pattern recognition and effective in scenarios where significant differences in the wing beat characteristics of individual birds are not present. However, the PCA-KNN approach requires precise alignment of the wing flap sounds, which is a manual process that is both tedious and time-consuming. The task requires close attention to detail and the success of the classification significantly depends on how accurately the wing flap sounds are aligned. Any imperfections in the manual alignment process may lower recognition rates.

Unlike the PCA-KNN method, the Cross-Correlation method does not require alignment of the wing flap sounds. This makes it a more efficient option as it does not require a manual alignment process. However, the method requires a longer calculation time due to its iterative process of shifting and recalculating distance metrics between the two wing flap sounds for each shift. This iterative shifting and computation process is repeated until a predetermined number of shifts is reached. In contrast, the PCA-KNN method involves just a single calculation of the distance metric between two arrays. Despite the increased computational demand, the Cross-Correlation method offers a promising alternative to the PCA-KNN approach for bird classification based on wing flap sounds.

Each wing flap sound array comprises approximately 20,000 elements. With the Cross-Correlation method in the current analysis, to conserve computation time, a total of only 4,000 zeros are padded, with the magnitude of each individual shift being 100 elements.

However, it is postulated that with enhanced computational capabilities, a more detailed analysis could be conducted. In such a scenario, padding as many as 40,000 zeros and reducing the magnitude of individual shifts to as small as 1 element could be feasible. This finer resolution in data processing is assumed to potentially yield higher accuracy in identifying and analyzing the wing flap sounds.

In the analysis of wing flap sounds, each data point is treated as a feature. To standardize their lengths for calculations, zeros were appended to the ends of these sounds. This modification results in features with unequal variance, primarily due to the addition of zeros, and also leads to sparse data. Features with unequal variance are situations where the variability or spread of values in one feature is significantly different from that of another, meaning the variance of one feature can be much larger or smaller than another. Sparse data, on the other hand, refers to datasets in which a large proportion of the elements are non-informative, typically zeros or missing entries.

Moreover, in this context, particularly because the data values range between -1 and 1, an underflow error is likely to occur. Underflow or overflow errors are numerical errors that happen in computer programs when values become too small or too large to be represented accurately. The squaring operations involved in calculating Cosine, Correlation, and Euclidean distances can exacerbate this issue, increasing the likelihood of underflow errors.

Tables 3.2 and 3.3 display the average recognition rates, with the highest average recognition rates achieved by both the PCA-KNN and Cross-Correlation methods occurring with the Cityblock distance metric. The superiority of the Cityblock distance is particularly evident when comparing its performance to that of Cosine, Correlation, and Euclidean distances in scenarios involving sparse data and features with unequal variance. This becomes apparent when examining two arrays: arrayA = [1, 0, 0, 0, 0.1, 0.2, 0.3, 0.4] and arrayB = [1, 0, 0, 0, 0, 0, 0, 0]. The calculated distances for Cityblock, Cosine, Correlation, and Euclidean were 1.00, 0.12, 0.10, and 0.55, respectively. With the Cityblock distance showing the highest value of 1.00, it distinctly indicates the dissimilarity between arrayA and arrayB. In contrast, the smaller values obtained from Cosine, Correlation, and Euclidean distances imply less clarity in distinguishing the dissimilarity. This example underscores the effectiveness of Cityblock distance in situations where other distances may not as clearly differentiate between arrays with sparse data and features that exhibit unequal variance.

In the application of the Cross-Correlation method, it is noted that the confusion matrices for Cosine distance (refer to Figure 3.13) and Correlation distance (refer to Figure 3.14) yield the same results. This can be attributed to the data points of the waveform being confined within the -1 to 1 range. Consequently, the average values of these waveforms are relatively low. This results in minimal average terms $\overline{x}_j$ and $\overline{y}_i$ in the Correlation distance formula (as detailed in Equation 1.6), leading Correlation distance to closely mimic the behavior of Cosine distance. However, when applying the PCA-KNN method, the confusion matrices for Cosine distance (shown in Figure 3.5) and Correlation distance (shown in Figure 3.6) do not yield identical results, possibly due to imperfect alignment.

5. CONCLUSION AND FUTURE STUDY

The goal of this study was to differentiate individual Zebra Finch birds by analyzing the distinct characteristics of their wing flap sounds. This exploration was enriched by the discovery that these sounds vary not only among individual birds but also in response to different flight behaviors, such as approaching a perch, flying away from a perch, and flying in a straight line. The study was conducted by recording the flight sounds of each Zebra Finch bird separately; one bird flew by the microphone and was recorded, followed by the next bird, ensuring that only one bird was recorded at a time. This method was utilized for a total of three individual Zebra Finch birds, capturing their unique audio signatures under these varied conditions.

The PCA-KNN combination and the Cross-Correlation method were used to classify a bird based on its acoustic pattern of flight. The highest identification accuracy reached up to 100% for flights with specific behaviors, such as flying away from the perch, approaching the perch, and flying in a straight line, as well as for flights that did not adhere to these specific patterns.

The wing flap sounds method provides a non-invasive technique for studying bird behavior, setting itself apart from conventional methods that typically require capturing, handling, marking, or attaching devices to birds. These traditional practices can lead to stress and potential harm for the birds. In contrast, this new technique is highly beneficial in research contexts, as it significantly reduces stress and harm, thereby enhancing the overall well-being and quality of life of the birds.

In real-world applications, the task of capturing wing flap sounds is fraught with significant challenges. While this study's recordings were made in a quiet, controlled room with birds flying in a consistent pattern past a microphone, the situation is markedly different in natural environments like nesting or feeding sites. Here, the microphone picks up a blend of various sounds – the noise of the wind, calls of other animals, human activities, and the wing flaps of other birds, ect – all mixing with the wing flap sounds of interest. This amalgamation of noises creates interference, complicating the task of accurately isolating and capturing the distinct wing flap sounds of individual birds.

In addressing the complexities of capturing wing flap sounds in natural environments, future work should focus on the development and application of advanced acoustic technologies and methodologies. This includes the utilization of directional microphones or parabolic listening devices, which are more adept at isolating specific sounds from background noise. Additionally, integrating sophisticated noise reduction and signal processing techniques, such as active noise cancellation and advanced filtering algorithms, could substantially enhance the clarity of the wing flap sounds. Furthermore, the implementation of machine learning models for sound analysis, capable of distinguishing and isolating the wing flap sounds from an array of environmental noises, presents a promising avenue. Strategic placement and timing of recordings, considering environmental noise patterns, could also improve data quality. Lastly, continual research into new and emerging acoustic processing technologies will be crucial in refining the methodology, thereby paving the way for more accurate and reliable bird behavior studies in their natural habitats.

In conclusion, the wing flap sounds method is a valuable tool for bird research that minimizes the impact on the birds and provides new opportunities for scientific discovery.

This research presents an innovative approach in ornithology, achieving individual bird recognition through the unique signature of wing flap sounds. This non-invasive technique marks a significant departure from traditional methods that require capturing and marking birds, which can be stressful and harmful to the subjects. Our approach not only preserves the well-being of birds but also facilitates continuous monitoring without interference in their natural behaviors. The successful application of our method demonstrates its potential as a sustainable, ethical alternative for wildlife research, offering an unobtrusive yet highly effective tool for the conservation and study of avian species.

REFERENCES

AbdAlKader, S. A. (2019). Human gait recognition based on feature extraction of support vector machine and pattern network algorithm. In *IOP conference series: Materials science and engineering*, volume 518, page 052010. IOP Publishing.

Ali, N., Neagu, D., and Trundle, P. (2019). Evaluation of k-nearest neighbour classifier performance for heterogeneous data sets. *SN Applied Sciences*, 1:1–15.

Amat, J. A. (1999). Foot losses of metal banded snowy plovers (pérdidas de pies en chorlitejos patinegros (charadrius alexandrinus) con anillas metálicas). *Journal of Field Ornithology*, pages 555–557.

Bedoya, C. L. and Molles, L. E. (2021). Acoustic censusing and individual identification of birds in the wild. *bioRxiv*, pages 2021–10.

Borman, R. I., Napianto, R., Nugroho, N., Pasha, D., Rahmanto, Y., and Yudoutomo, Y. E. P. (2021). Implementation of PCA and KNN algorithms in the classification of indonesian medicinal plants. In *2021 International Conference on Computer Science, Information Technology, and Electrical Engineering (ICOMITEE)*, pages 46–50. IEEE.

Boulton, R. L., Richard, Y., and Armstrong, D. P. (2010). The effect of male incubation feeding, food and temperature on the incubation behaviour of new zealand robins. *Ethology*, 116(6):490–497.

Brin, S. (1995). Near neighbor search in large metric spaces. In *VLDB*, volume 95, pages 574–584. Citeseer.

Bukhari, W., Yun, C., Kassim, A., and Tokhi, M. (2020). Study of k-nearest neighbour classification performance on fatigue and non-fatigue emg signal features. *International Journal of Advanced Computer Science and Applications*, 11(8):41–47.

Cantarero, A., López-Arrabé, J., Plaza, M., Saavedra-Garcés, I., and Moreno, J. (2016). Males feed their mates more and take more risks for nestlings with larger female-built nests: an experimental study in the nuthatch sitta europaea. *Behavioral ecology and sociobiology*, 70(8):1141–1150.

Cohen, W. W., Ravikumar, P., Fienberg, S. E., et al. (2003). A comparison of string distance metrics for name-matching tasks. In *IIWeb*, volume 3, pages 73–78.

Daffertshofer, A., Lamoth, C. J., Meijer, O. G., and Beek, P. J. (2004). PCA in studying coordination and variability: a tutorial. *Clinical biomechanics*, 19(4):415–428.

Ferreira, A. C., Silva, L. R., Renna, F., Brandl, H. B., Renoult, J. P., Farine, D. R., Covas, R., and Doutrelant, C. (2020). Deep learning-based methods for individual recognition in small birds. *Methods in Ecology and Evolution*, 11(9):1072–1085.

Galeotti, P. and Pavan, G. (1991). Individual recognition of male tawny owls (strix aluco) using spectrograms of their territorial calls. *Ethology Ecology & Evolution*, 3(2):113–126.

Griesser, M., Schneider, N. A., Collis, M.-A., Overs, A., Guppy, M., Guppy, S., Takeuchi, N., Collins, P., Peters, A., and Hall, M. L. (2012). Causes of ring-related leg injuries in birds–evidence and recommendations from four field studies. *PLoS One*, 7(12).

Hałupka, K. (1994). Incubation feeding in meadow pipit anthus pratensis affects female time budget. *Journal of Avian Biology*, pages 251–253.

Haque, P., Das, B., and Kaspy, N. N. (2019). Two-handed bangla sign language recognition using principal component analysis (PCA) and KNN algorithm. In *2019 international conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 1–4. IEEE.

Hoi, S. C., Liu, W., Lyu, M. R., and Ma, W.-Y. (2006). Learning distance metrics with contextual constraints for image retrieval. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2072–2078. IEEE.

Jolliffe, I. T. (2002). *Principal component analysis for special types of data.* Springer.

Jutzi, B. and Gross, H. (2009). Nearest neighbour classification on laser point clouds to gain object structures from buildings. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(Part 1):4–7.

Karim, T. F., Lipu, M. S. H., Rahman, M. L., and Sultana, F. (2010). Face recognition using PCA-based method. In *2010 IEEE International Conference on Advanced Management Science (ICAMS 2010)*, volume 3, pages 158–162. IEEE.

Kataria, A. and Singh, M. (2013). A review of data classification using k-nearest neighbour algorithm. *International Journal of Emerging Technology and Advanced Engineering*, 3(6):354–360.

Kramer, O. and Kramer, O. (2013). K-nearest neighbors. *Dimensionality reduction with unsupervised nearest neighbors*, pages 13–23.

Kumar, A., Gupta, S., Singh, S. K., and Shukla, K. K. (2015). Comparison of various metrics used in collaborative filtering for recommendation system. In *2015 Eighth International Conference on Contemporary Computing (IC3)*, pages 150–154. IEEE.

Laaksonen, J. and Oja, E. (1996). Classification with learning k-nearest neighbors. In *Proceedings of international conference on neural networks (ICNN'96)*, volume 3, pages 1480–1483. IEEE.

Lakshmi, S. V. and Prabakaran, T. E. (2014). Application of k-nearest neighbour classification method for intrusion detection in network data. *International Journal of Computer Applications*, 97(7).

Linhart, P., Fuchs, R., Poláková, S., and Slabbekoorn, H. (2012). Once bitten twice shy: long-term behavioural changes caused by trapping experience in willow warblers phylloscopus trochilus. *Journal of avian biology*, 43(2):186–192.

Mucherino, A., Papajorgji, P. J., Pardalos, P. M., Mucherino, A., Papajorgji, P. J., and Pardalos, P. M. (2009). K-nearest neighbor classification. *Data mining in agriculture*, pages 83–106.

Pedroza, A. D., De la Rosa, J. I., Rosas, R., Becerra, A., Villa, J., Moreno, G., González, E., and Alaniz, D. (2020). Acoustic individual identification in birds based on the band-limited phase-only correlation function. *Applied Sciences*, 10(7):2382.

Ramteke, R. and Monali, K. Y. (2012). Automatic medical image classification and abnormality detection using k-nearest neighbour. *International Journal of Advanced Computer Research*, 2(4):190.

Rogers, D. J. and Paton, D. C. (2005). Acoustic identification of individual rufous bristlebirds, a threatened species with complex song repertoires. *Emu-Austral Ornithology*, 105(3):203–210.

Ruan, Y., Xiao, Y., Hao, Z., and Liu, B. (2021). A nearest-neighbor search model for distance metric learning. *Information Sciences*, 552:261–277.

Sandell, M. I., Smith, H. G., and Bruun, M. (1996). Paternal care in the european starling, sturnus vulgaris: nestling provisioning. *Behavioral Ecology and Sociobiology*, 39(5):301–309.

Shlens, J. (2014). A tutorial on principal component analysis. *arXiv preprint arXiv:1404.1100*.

Singh, A., Yadav, A., and Rana, A. (2013). K-means with three different distance metrics. *International Journal of Computer Applications*, 67(10).

Suárez, J. L., García, S., and Herrera, F. (2021). A tutorial on distance metric learning: Mathematical foundations, algorithms, experimental analysis, prospects and challenges. *Neurocomputing*, 425:300–322.

Wierucka, K., Halupka, L., Klimczuk, E., and Sztwiertnia, H. (2016). Survival during the breeding season: nest stage, parental sex, and season advancement affect reed warbler survival. *PLoS One*, 11(3):e0148063.

Wold, S., Esbensen, K., and Geladi, P. (1987). Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52.

Xia, W., Song, T., Yan, Z., Song, K., Chen, D., and Chen, Y. (2021). A method for recognition of mixed gas composition based on PCA and KNN. In *2021 19th International Conference on Optical Communications and Networks (ICOCN)*, pages 1–3. IEEE.

Yang, L. and Jin, R. (2006). Distance metric learning: A comprehensive survey. *Michigan State Universiy*, 2(2):4.

Yu, J., Gao, M., Rong, W., Song, Y., and Xiong, Q. (2017). A social recommender based on factorization and distance metric learning. *IEEE Access*, 5:21557–21566.

Zhang, R., Vogler, C., and Metaxas, D. (2004). Human gait recognition. In *2004 Conference on Computer Vision and Pattern Recognition Workshop*, pages 18–18. IEEE.

APPENDIX

A.1. Appendix MATLAB Code

The MATLAB code is structured into three distinct segments. The first segment is dedicated to preprocessing bird flight waveforms, involving filtering, scaling, padding with zeros to equalize the lengths of all waveforms, and randomly splitting each group into 80% for training and 20% for testing. The second segment applies PCA-KNN for classification, while the third segment employs Cross-Correlation techniques for the same objective.

A.1.1. MATLAB Code Segment I - Waveform Preprocessing

This section of the MATLAB code is meticulously designed for processing bird flight waveforms.

```
% Birds_Fly_Preproc
% This script processes flight waveform data of birds.
%
% Steps:
% 1. Data Loading: Loads flight data from a .mat file and a low-pass
%    filter.
% 2. Data Processing: Applies the filter to different sets of flight
%    waveforms and normalizes them.
% 3. Data Selection and Plotting: Selects random samples and plots data
%    for various flight types.
% 4. Data Pruning: Prunes the beginning and end of each waveform set.
% 5. Determining Longest Waveform: Finds the longest waveform for
%    processing.
% 6. Data Padding: Pads zeros to the end of waveforms to achieve uniform
%    length.
% 7. Data Splitting: Splits data into training and testing sets.
% 8. Saving Processed Data: Saves processed data to a .mat file.

%%
clc;        % Clears the Command Window
close all;  % Closes all open figures
clear;      % Removes all variables from the workspace

load('Data_Birds_Fly.mat'); % Load data from .mat file

Hd = low_pass_filter(); % Load the low-pass filter
```

```
%%
% Call the 'filter_scale' function to apply the low pass filter to each
% flight waveforms and scale each waveform to a range between -1 and 1
CuA_filter_scale = filter_scale(CuA, Hd);
CuL_filter_scale = filter_scale(CuL, Hd);
CuS_filter_scale = filter_scale(CuS, Hd);
MoA_filter_scale = filter_scale(MoA, Hd);
MoL_filter_scale = filter_scale(MoL, Hd);
MoS_filter_scale = filter_scale(MoS, Hd);
LaA_filter_scale = filter_scale(LaA, Hd);
LaL_filter_scale = filter_scale(LaL, Hd);
LaS_filter_scale = filter_scale(LaS, Hd);

%%
% Call the 'selectRandom20' function to randomly select 20 flight
% waveforms for plotting
CuA_selected = selectRandom20(CuA_filter_scale);
CuL_selected = selectRandom20(CuL_filter_scale);
CuS_selected = selectRandom20(CuS_filter_scale);
MoA_selected = selectRandom20(MoA_filter_scale);
MoL_selected = selectRandom20(MoL_filter_scale);
MoS_selected = selectRandom20(MoS_filter_scale);
LaA_selected = selectRandom20(LaA_filter_scale);
LaL_selected = selectRandom20(LaL_filter_scale);
LaS_selected = selectRandom20(LaS_filter_scale);

% Call the 'plotWaveforms' to plot the waveform

% Create subplots for the group type 'Flights Approaching the Perch'
figure; % Create a new figure window
% Plot data for flights approaching the perch'
plotWaveforms(1, CuA_selected, 'Curly''s Flights Approaching the Perch');
plotWaveforms(2, MoA_selected, 'Moe''s Flights Approaching the Perch');
plotWaveforms(3, LaA_selected, 'Larry''s Flights Approaching the Perch');

% Create subplots for the group type 'Flights Leaving the Perch'
figure; % Create a new figure window
% Plot data for flights leaving the perch'
plotWaveforms(1, CuL_selected, 'Curly''s Flights Leaving the Perch');
plotWaveforms(2, MoL_selected, 'Moe''s Flights Leaving the Perch');
plotWaveforms(3, LaL_selected, 'Larry''s Flights Leaving the Perch');

% Create subplots for the group type 'Flights Between Two Perches'
figure; % Create a new figure window
% Plot data for flights between two perches'
plotWaveforms(1, CuS_selected, 'Curly''s Flights Between Two Perches');
plotWaveforms(2, MoS_selected, 'Moe''s Flights Between Two Perches');
plotWaveforms(3, LaS_selected, 'Larry''s Flights Between Two Perches');

%%
% Call the 'prunedStartWaveform' function to prune the beginning of each
% waveform for CuA, CuL, CuS, MoA, MoL, MoS, LaA, LaL, and LaS
```

```
CuA_prunedStart = prunedStartWaveform(CuA_filter_scale);
CuL_prunedStart = prunedStartWaveform(CuL_filter_scale);
CuS_prunedStart = prunedStartWaveform(CuS_filter_scale);
MoA_prunedStart = prunedStartWaveform(MoA_filter_scale);
MoL_prunedStart = prunedStartWaveform(MoL_filter_scale);
MoS_prunedStart = prunedStartWaveform(MoS_filter_scale);
LaA_prunedStart = prunedStartWaveform(LaA_filter_scale);
LaL_prunedStart = prunedStartWaveform(LaL_filter_scale);
LaS_prunedStart = prunedStartWaveform(LaS_filter_scale);

%%
% Call the 'prunedEndWaveform' function to prune the end of each
% waveform for CuA, CuL, CuS, MoA, MoL, MoS, LaA, LaL, and LaS
CuA_prunedEnd = prunedEndWaveform(CuA_prunedStart);
CuL_prunedEnd = prunedEndWaveform(CuL_prunedStart);
CuS_prunedEnd = prunedEndWaveform(CuS_prunedStart);
MoA_prunedEnd = prunedEndWaveform(MoA_prunedStart);
MoL_prunedEnd = prunedEndWaveform(MoL_prunedStart);
MoS_prunedEnd = prunedEndWaveform(MoS_prunedStart);
LaA_prunedEnd = prunedEndWaveform(LaA_prunedStart);
LaL_prunedEnd = prunedEndWaveform(LaL_prunedStart);
LaS_prunedEnd = prunedEndWaveform(LaS_prunedStart);

%%
% This code finds the lengths of the longest and shortest cell arrays

% Create a cell array containing the pruned-end waveforms for CuA, CuL,
% CuS, MoA, MoL, MoS, LaA, LaL, and LaS
cellArrays = {CuA_prunedEnd, CuL_prunedEnd, CuS_prunedEnd, ...
             MoA_prunedEnd, MoL_prunedEnd, MoS_prunedEnd, ...
             LaA_prunedEnd, LaL_prunedEnd, LaS_prunedEnd};

% Initialize min and max lengths
minLength = inf; % Start with infinity for minimum
maxLength = -inf; % Start with negative infinity for maximum

% Loop through each cell array
for i = 1:length(cellArrays)
    % Loop through each array in the cell array
    for j = 1:length(cellArrays{i})
        currentLength = length(cellArrays{i}{j});
        if currentLength < minLength
            minLength = currentLength;
        end
        if currentLength > maxLength
            maxLength = currentLength;
        end
    end
end

% Display results
disp(['Shortest Array Length: ', num2str(minLength)]);
disp(['Longest Array Length: ', num2str(maxLength)]);

%%
% Plot to visualize the flight waveform before and after pruning both
% ends

% Randomly select one array from CuA_filter_scale
selectedArrayIndex = randi(length(CuA_filter_scale));

% Instead of selecting randomly, set selectedArrayIndex = 46 for clarity
selectedArrayIndex = 46;

selectedArrayProcessed = CuA_filter_scale{selectedArrayIndex};
selectedArrayNew = CuA_prunedEnd{selectedArrayIndex};

% Determine the maximum length for the x-axis
maxLength = max(length(selectedArrayProcessed), length(selectedArrayNew));

% Find local maxima and minima in the processed flight waveform
[positivePeaksProcessed, locsPositiveProcessed] = ...
    findpeaks(selectedArrayProcessed);
[negativePeaksProcessed, locsNegativeProcessed] = ...
    findpeaks(-selectedArrayProcessed);
% Correct sign for minima
negativePeaksProcessed = -negativePeaksProcessed;

% Find local maxima and minima in the new (pruned) waveform
[positivePeaksNew, locsPositiveNew] = findpeaks(selectedArrayNew);
[negativePeaksNew, locsNegativeNew] = findpeaks(-selectedArrayNew);
negativePeaksNew = -negativePeaksNew; % Correct sign for minima

% Plot the selected waveform from CuA_filter_scale
figure;
subplot(2,1,1);
plot(selectedArrayProcessed, 'k'); % Black color for the waveform
hold on;

% Mark and label the local maxima in processed waveform
for i = 1:length(locsPositiveProcessed)
    plot(locsPositiveProcessed(i), positivePeaksProcessed(i), ...
         'v', 'MarkerSize', 10, 'MarkerEdgeColor', 'k', ...
         'MarkerFaceColor', 'none'); % Black color for markers
    text(locsPositiveProcessed(i), positivePeaksProcessed(i), ...
         sprintf('%.2f', positivePeaksProcessed(i)), ...
         'VerticalAlignment', 'bottom');
end

% Mark and label the local minima in processed waveform
for i = 1:length(locsNegativeProcessed)
    plot(locsNegativeProcessed(i), negativePeaksProcessed(i), ...
         'o', 'MarkerSize', 10, 'MarkerEdgeColor', 'k', ...
         'MarkerFaceColor', 'none'); % Black color for markers
    text(locsNegativeProcessed(i), negativePeaksProcessed(i), ...
         sprintf('%.2f', negativePeaksProcessed(i)), ...
         'VerticalAlignment', 'top');
end

% Draw dashed lines at y = 0.1, y = -0.1, and y = 0
yline(0.1, '--', 'Color', 'k');
yline(-0.1, '--', 'Color', 'k');
yline(0, ':', 'Color', 'k'); % Dotted line at y = 0

% Set x-axis and y-axis limits
xlim([0 maxLength]);
ylim([-1 1]);

hold off;
title('Selected flight waveform before both ends are pruned');
xlabel('Sample Index');
ylabel('Amplitude');

% Plot the corresponding flight waveform from CuA_prunedEnd
subplot(2,1,2);
plot(selectedArrayNew, 'k'); % Black color for the waveform
hold on;

% Mark and label the local maxima in corresponding flight waveform
for i = 1:length(locsPositiveNew)
    plot(locsPositiveNew(i), positivePeaksNew(i), ...
         'v', 'MarkerSize', 10, 'MarkerEdgeColor', 'k', ...
         'MarkerFaceColor', 'none'); % Black color for markers
    text(locsPositiveNew(i), positivePeaksNew(i), ...
         sprintf('%.2f', positivePeaksNew(i)), ...
         'VerticalAlignment', 'bottom');
end

% Mark and label the local minima in corresponding flight waveform
for i = 1:length(locsNegativeNew)
    plot(locsNegativeNew(i), negativePeaksNew(i), ...
         'o', 'MarkerSize', 10, 'MarkerEdgeColor', 'k', ...
         'MarkerFaceColor', 'none'); % Black color for markers
    text(locsNegativeNew(i), negativePeaksNew(i), ...
         sprintf('%.2f', negativePeaksNew(i)), ...
         'VerticalAlignment', 'top');
end

% Draw dashed lines at y = 0.1, y = -0.1, and y = 0
yline(0.1, '--', 'Color', 'k');
yline(-0.1, '--', 'Color', 'k');
yline(0, ':', 'Color', 'k'); % Dotted line at y = 0

% Set x-axis and y-axis limits
xlim([0 maxLength]);
ylim([-1 1]);
```

```
hold off;
title('Corresponding flight waveform with both ends pruned');
xlabel('Sample Index');
ylabel('Amplitude');

%%
% Determine the length of the longest flight waveform to ascertain the
% number of zeros to be added to each waveform

% Store the list of variable names in a cell array
variables = {'CuA_prunedEnd', 'CuL_prunedEnd', 'CuS_prunedEnd', ...
             'MoA_prunedEnd', 'MoL_prunedEnd', 'MoS_prunedEnd', ...
             'LaA_prunedEnd', 'LaL_prunedEnd', 'LaS_prunedEnd'};

% Initialize an array to store the maxLengthResults
maxLengthResults = [];

% Loop through each variable name
for i = 1:length(variables)
    % Get the result for the current variable
    currentResult = find_longest_array_length_in_variable(variables{i});

    % Append the result to the maxLengthResults array
    maxLengthResults = [maxLengthResults, currentResult];
end

% Find the maximum value in the MaxLength column
[maxLength, idx] = max([maxLengthResults.MaxLength]);

% Store the maximum length in a variable
longestArrayLength = maxLength;

%%
% Pad zeros to waveforms to make all waveforms uniformly length

% Array of original cell array names
originalNames = {'CuA_prunedEnd', 'CuL_prunedEnd', 'CuS_prunedEnd', ...
                 'MoA_prunedEnd', 'MoL_prunedEnd', 'MoS_prunedEnd', ...
                 'LaA_prunedEnd', 'LaL_prunedEnd', 'LaS_prunedEnd'};

% Prefix for new variable names
prefix = 'preProcess_';

% Loop through each cell array name
for k = 1:length(originalNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', originalNames{k});

    % Loop through each array in the current cell array
    for i = 1:length(currentCellArray)
        currentLength = length(currentCellArray{i});
        if currentLength < longestArrayLength
            % Calculate the number of zeros needed for padding
            paddingLength = longestArrayLength - currentLength;

            % Determine if the array is a row or column vector and pad
            % with zeros
            if isrow(currentCellArray{i})
                currentCellArray{i} = [currentCellArray{i}, ...
                    zeros(1, paddingLength)];
            else % For a column vector
                currentCellArray{i} = [currentCellArray{i}; ...
                    zeros(paddingLength, 1)];
            end
        end
    end

    % Create new variable name
    newVarName = [prefix, strrep(originalNames{k}, '_prunedEnd', '')];

    % Assign the processed cell array to the new variable in the
    % base workspace
    assignin('base', newVarName, currentCellArray);
end

%%
% Count the number of waveforms  in each group. If a group contains more than
% 100 waveforms, randomly select 100 waveforms . Do nothing if the group has
% exactly 100 waveforms.

% Set the seed for the random number generator
rng(1);

% Store the list of variable names in a cell array
originalNames = {'preProcess_CuA', 'preProcess_CuL', 'preProcess_CuS', ...
                 'preProcess_MoA', 'preProcess_MoL', 'preProcess_MoS', ...
                 'preProcess_LaA', 'preProcess_LaL', 'preProcess_LaS'};

% Suffix for new variable names
suffix = '_100_selected';

% Number of arrays to select
numToSelect = 100;

% Loop through each cell array name
for k = 1:length(originalNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', originalNames{k});

    % Check if the cell array has more than numToSelect waveforms
    if length(currentCellArray) > numToSelect
        % Randomly select numToSelect indices
        selectedIndices = randperm(length(currentCellArray), numToSelect);
        % Select waveforms at these indices
        selectedArrays = currentCellArray(selectedIndices);
    else
        % If the cell array has numToSelect or fewer waveforms, use it as is
        selectedArrays = currentCellArray;
    end

    % Create new variable name
    newVarName = [originalNames{k}, suffix];

    % Assign the selected arrays to the new variable in the base workspace
    assignin('base', newVarName, selectedArrays);
end

%%
% Randomly split each group, which consists of 100 waveforms,
% into an 80/20 ratio

% Set the seed for the random number generator for consistency
rng(4);

% Store the list of variable names in a cell array
originalNames = {'preProcess_CuA_100_selected', ...
    'preProcess_CuL_100_selected', 'preProcess_CuS_100_selected', ...
    'preProcess_MoA_100_selected', 'preProcess_MoL_100_selected', ...
    'preProcess_MoS_100_selected', 'preProcess_LaA_100_selected', ...
    'preProcess_LaL_100_selected', 'preProcess_LaS_100_selected'};

% Prefix for the new variable names
prefixes = {'CuA', 'CuL', 'CuS', 'MoA', 'MoL', 'MoS', 'LaA', 'LaL', ...
    'LaS'};

% Loop through each cell array name
for k = 1:length(originalNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', originalNames{k});

    % Determine the split indices (80/20 split)
    totalLength = length(currentCellArray);
    splitIndex = round(totalLength * 0.8);

    % Randomly shuffle the indices
    shuffledIndices = randperm(totalLength);

    % Split the cell array into training (80%) and testing (20%) subsets
    trainingData = currentCellArray(shuffledIndices(1:splitIndex));
    testingData = currentCellArray(shuffledIndices(splitIndex+1:end));

    % Create new variable names
    trainingVarName = ['trainingData_', prefixes{k}];
    testingVarName = ['testingData_', prefixes{k}];
    % Assign the subsets to new variables in the base workspace
    assignin('base', trainingVarName, trainingData);
    assignin('base', testingVarName, testingData);
end

%%
% Transposes the contents of training and testing data cell arrays

% Store the list of variable names in a cell array
% Array of cell array names for training and testing data
trainingNames = {'trainingData_CuA', 'trainingData_CuL', ...
    'trainingData_CuS', 'trainingData_MoA', 'trainingData_MoL', ...
    'trainingData_MoS', 'trainingData_LaA', 'trainingData_LaL', ...
    'trainingData_LaS'};
testingNames = {'testingData_CuA', 'testingData_CuL', ...
    'testingData_CuS', 'testingData_MoA', 'testingData_MoL', ...
    'testingData_MoS', 'testingData_LaA', 'testingData_LaL', ...
    'testingData_LaS'};

% Transpose each array in the training cell arrays
for k = 1:length(trainingNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', trainingNames{k});

    % Transpose each array within the cell array
    transposedCellArray = cellfun(@(x) x', currentCellArray, ...
        'UniformOutput', false);

    % Assign the transposed cell array back to the base workspace
    assignin('base', trainingNames{k}, transposedCellArray);
end

% Transpose each array in the testing cell arrays
for k = 1:length(testingNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', testingNames{k});

    % Transpose each array within the cell array
    transposedCellArray = cellfun(@(x) x', currentCellArray, ...
        'UniformOutput', false);

    % Assign the transposed cell array back to the base workspace
    assignin('base', testingNames{k}, transposedCellArray);
end

%%
%Save the dataset, including both training and testing data, ready for
% classification

% Store the list of variable names in a cell array
% Specify the variable names to save
variableNames = {'trainingData_CuA', 'trainingData_CuL', ...
```

```
    'trainingData_CuS', 'trainingData_MoA', 'trainingData_MoL', ...
    'trainingData_MoS', 'trainingData_LaA', 'trainingData_LaL', ...
    'trainingData_LaS', 'testingData_CuA', 'testingData_CuL', ...
    'testingData_CuS', 'testingData_MoA', 'testingData_MoL', ...
    'testingData_MoS', 'testingData_LaA', 'testingData_LaL', ...
    'testingData_LaS'};

% Save the variables to 'preProcData.mat'
save('preProcBirdsFliesData.mat', variableNames{:});

%%
%END OF 'Birds_Fly_Preproc' CODE

function Hd = low_pass_filter
%FILTER1 Returns a discrete-time filter object.

% MATLAB Code
% Generated by MATLAB(R) 9.6 and Signal Processing Toolbox 8.2.
% Generated on: 01-Jul-2019 16:43:39

% Butterworth Lowpass filter designed using FDESIGN.LOWPASS.

% All frequency values are in Hz.
Fs = 44100;  % Sampling Frequency

Fpass = 100;         % Passband Frequency
Fstop = 1000;        % Stopband Frequency
Apass = 1;           % Passband Ripple (dB)
Astop = 80;          % Stopband Attenuation (dB)
match = 'stopband';  % Band to match exactly

% Construct an FDESIGN object and call its BUTTER method.
h  = fdesign.lowpass(Fpass, Fstop, Apass, Astop, Fs);
Hd = design(h, 'butter', 'MatchExactly', match);

% END OF 'low_pass_filter' FUNCTION

function processedCellArray = filter_scale(cellArray, filterObj)
    % This function applies a filter and normalization to each waveforms in
    % a cell array.
    %
    % This function takes a cell array of waveforms and a filter object. It
    % applies the filter to each waveforms in the cell array and then
    % normalizes the filtered waveform so that its values are within the
    % range [-1, 1].
    %
    % Inputs:
    % cellArray - A cell array where each element is a waveforms to be
    % processed.
    % filterObj - A filter function used for low-pass filtering the waveforms.
    %
    % Output:
    % processedCellArray - A cell array containing the filtered and normalized
    % waveforms.

    % Initialize an empty cell array of the same size as the input cellArray
    processedCellArray = cell(size(cellArray));

    % Loop over each element in the cell array
    for i = 1:numel(cellArray)
        % Apply the low-pass filter specified by filterObj to the input array
        filteredArray = filter(filterObj, cellArray{i});

        % Find the maximum absolute value in the filtered array
        maxAbsVal = max(abs(filteredArray));

        % Check if the maximum absolute value is nonzero
        if maxAbsVal ~= 0
            % Normalize the filtered array to be within the range [-1, 1]
            processedCellArray{i} = filteredArray / maxAbsVal;
        else
            % If the maximum absolute value is zero, return the filtered
            % array as is
            processedCellArray{i} = filteredArray;
        end
    end
end
% END OF 'filter_scale' FUNCTION

function selected = selectRandom20(cellArray)
    % This function selects 20 random waveforms from a cell array.
    %
    % This function takes a cell array as input and randomly selects up to 20
    % waveforms from it. If the cell array contains 20 or fewer waveforms, it
    % selects all waveforms in a random order. If the cell array contains more
    % than 20 waveforms, it selects 20 waveforms at random without replacement
    .
    %
    % Input:
    % cellArray - A cell array from which waveforms will be randomly selected.
    %
    % Output:
    % selected - A cell array containing the randomly selected waveforms.
    indices = randperm(length(cellArray), min(20, length(cellArray)));
    selected = cellArray(indices);
end
% END OF 'selectRandom20' FUNCTION

function plotWaveforms(subplotIndex, data, titleStr)
    % This function plots the waveforms in a specified subplot.
    %
    % This function takes a series of waveforms (in a cell array), a subplot
    % index, and a title string, then plots each waveform in the specified
    % subplot of a 3-row subplot structure. It is designed to work within a
    % figure that has been divided into three subplots vertically.
    %
    % Inputs:
    % subplotIndex - An integer specifying which subplot to use (1, 2, or 3).
    % data - A cell array where each cell contains waveform to be plotted.
    % titleStr - A string for the title of the subplot.
    subplot(3,1,subplotIndex);
    hold on;
    for i = 1:length(data)
        plot(data{i});
    end
    hold off;
    title(titleStr);
end
% END OF 'plotWaveforms' FUNCTION

function processedCellArray = prunedStartWaveform(cellArray)
    % This function trims each waveform contained in cell arrays.
    % It identifies peaks and troughs in each waveform
    % and potentially trims the waveform starting from the first element of
    % the waveform to a point relative to the first peak or trough.
    %
    % The trimming process is based on the first significant peak or trough,
    % where 'significant' is defined by a threshold: greater than or equal to
    % 0.1 for a peak and less than or equal to -0.1 for a trough. The
    % waveform is trimmed to a point where it crosses zero before this peak
    % or trough. If no such peak or troughis found, or if the waveform does
    % not cross zero before the first peak or trough, the waveform is not
    % trimmed.
    %
    % Inputs:
    % cellArray - A cell array where each element is a waveform to be
    %             processed.
    %
    % Outputs:
    % processedCellArray - A cell array containing the trimmed waveforms.

    % Initialize an empty cell array of the same size
    processedCellArray = cell(size(cellArray));

    % Loop over each waveform in the cell array
    for i = 1:numel(cellArray)
        waveform = cellArray{i};

        % Find local maxima and minima in the waveform
        [pksMax, locsMax] = findpeaks(waveform);
        [pksMin, locsMin] = findpeaks(-waveform);
        pksMin = -pksMin; % Correct the sign for minima

        % Find the first significant local maximum or minimum
        firstMaxIndex = find(pksMax >= 0.1, 1);
        firstMinIndex = find(pksMin <= -0.1, 1);

        % Determine the start point for trimming
        if isempty(firstMaxIndex) && isempty(firstMinIndex)
```

```
            zeroIndex = []; % No significant peaks found
        else
            if isempty(firstMinIndex) || (~isempty(firstMaxIndex) && ...
                    locsMax(firstMaxIndex) < locsMin(firstMinIndex))
                % First significant peak is a maximum
                zeroIndex = find(waveform(1:locsMax(firstMaxIndex))...
                    <= 0, 1, 'last');
            else
                % First significant peak is a minimum
                zeroIndex = find(waveform(1:locsMin(firstMinIndex))...
                    >= 0, 1, 'last');
            end
        end

        % Trim the waveform if a valid zeroIndex is found
        if ~isempty(zeroIndex)
            processedCellArray{i} = waveform(zeroIndex+1:end);
        else
            processedCellArray{i} = waveform;
        end
    end
end
% END OF 'prunedStartWaveform' FUNCTION

function trimmed_waveforms = prunedEndWaveform(original_waveforms)
    % This function trims each waveform contained in cell arrays.
    % It identifies peaks and troughs in each waveform
    % and potentially trims the waveform starting from the last element back
    % to a point relative to the last peak or trough.
    %
    % The trimming process is based on the last significant peak or trough,
    % where 'significant' is defined by a threshold: greater than or equal to
    % 0.1 for a peak and less than or equal to -0.1 for a trough. The waveform
    % is trimmed to a point where it crosses zero after this peak or trough.
    % If no such peak or trough is found, or if the waveform does not cross
    % zero after the last peak or trough, the waveform is not trimmed.
    %
    % Inputs:
    % original_waveforms - A cell array where each element is a waveform
    %                      to be processed.
    %
    % Outputs:
    % trimmed_waveforms - A cell array containing the trimmed waveforms.

    % Process each waveform in the cell array
    trimmed_waveforms = cell(size(original_waveforms));
    for i = 1:length(original_waveforms)
        waveform = original_waveforms{i};
        [pks_max, locs_max] = findpeaks(waveform);
        [pks_min, locs_min] = findpeaks(-waveform);

        % Convert locations of minima to positive peaks
        pks_min = -pks_min;
        % Filter peaks based on the threshold
        valid_max = locs_max(pks_max >= 0.1);
        valid_min = locs_min(pks_min <= -0.1);

        % Find the last valid peak, max or min
        if ~isempty(valid_max) && ~isempty(valid_min)
            last_peak_loc = max(max(valid_max), max(valid_min));
        elseif ~isempty(valid_max)
            last_peak_loc = max(valid_max);
        elseif ~isempty(valid_min)
            last_peak_loc = max(valid_min);
        else
            last_peak_loc = 0;
        end

        % Trim the waveform based on the last peak
        if last_peak_loc > 0
            if any(waveform(last_peak_loc:end) <= 0) && ...
                    ismember(last_peak_loc, valid_max)
                zero_cross = find(waveform(last_peak_loc:end) <= ...
                    0, 1, 'first');
                waveform = waveform(1:last_peak_loc+zero_cross-2);
            elseif any(waveform(last_peak_loc:end) >= 0) ...
                    && ismember(last_peak_loc, valid_min)
                zero_cross = find(waveform(last_peak_loc:end) >= ...
                    0, 1, 'first');
                waveform = waveform(1:last_peak_loc+zero_cross-2);
            end
        end

        % Save the trimmed waveform
        trimmed_waveforms{i} = waveform;
    end
end
% END OF 'prunedEndWaveform' FUNCTION
```

A.1.2. MATLAB Code Segment II - PCA-KNN Classification

This segment implements PCA (Principal Component Analysis) combined with KNN (K-Nearest Neighbors) for the classification of bird flight patterns.

```
% PCA_KNN.m
% This script performs Principal Component Analysis (PCA) and K-Nearest
% Neighbors (KNN) classification on bird flight data.
%
% Steps:
% 1. Data Loading: Loads preprocessed bird flight data from a .mat file.
% 2. Length Check: Verifies if all arrays in the dataset have equal lengths.
%                  Reports any discrepancies.
% 3. Data Concatenation: Vertically stacks arrays of flight waveforms for
%                        both training and testing data.
% 4. Label Creation: Creates label vectors for training and testing datasets.
% 5. Data Standardization: Standardizes the training and testing data based on
%                          the training data mean and standard deviation.
% 6. PCA-KNN Classification: Applies PCA to reduce dimensionality and then
%                            performs KNN classification.
%    - Loops over various distance metrics and K values for KNN.
%    - Calculates and stores classification accuracies.
%    - Determines optimal PCA components and K values for each distance
%      metric.
% 7. Accuracy Visualization: Plots accuracy versus the number of principal
%    components for different K values and distance metrics.
% 8. Summary Table: Creates and displays a summary table of the best accuracy
%    details for each combination of distance metric and K value.
% 9. Optimal Parameters Identification: Stores and displays the best K value
%    and the number of principal components for each distance metric.
% 10. Confusion Matrix Calculation: Reruns the PCA-KNN analysis using the
%     optimal parameters to generate confusion matrices.
% 11. Confusion Matrix Visualization: Visualizes the confusion matrices for
%     different distance metrics.
% 12. Confusion Matrix Grouping: Groups the 9x9 confusion matrix into a
%     smaller 3x3 matrix, calculates recognition percentages, and displays
%     the matrices with enhanced graphical details.
%
% Auxiliary Functions:
% - concatenateArrays: Function to concatenate arrays within a cell.
% - extractConfusionMatrices: Function to extract confusion matrices for
%   specific distance metrics from a structured KNN classification collection.

%%
clc;        % Clears the Command Window
close all;  % Closes all open figures
clear;      % Removes all variables from the workspace

% Load data from .mat file
load('preProcBirdsFliesData_rng(4).mat');

%%
% Check if the lengths of all waveforms are equal

% Creates a list of names for training and testing data cell arrays
cellArrayNames = {'trainingData_CuA', 'trainingData_CuL', ...
    'trainingData_CuS', 'trainingData_MoA', 'trainingData_MoL', ...
    'trainingData_MoS', 'trainingData_LaA', 'trainingData_LaL', ...
    'trainingData_LaS', 'testingData_CuA', 'testingData_CuL', ...
    'testingData_CuS', 'testingData_MoA', 'testingData_MoL', ...
    'testingData_MoS', 'testingData_LaA', 'testingData_LaL', ...
    'testingData_LaS'};

% Determine the maximum number of arrays in any cell array
```

```
maxArrays = 0;
for k = 1:length(cellArrayNames)
    numArrays = length(evalin('base', cellArrayNames{k}));
    maxArrays = max(maxArrays, numArrays);
end

% Initialize a matrix to store lengths from all cell arrays
numCellArrays = length(cellArrayNames);
allLengths = zeros(maxArrays, numCellArrays);

% Loop through each cell array name
for k = 1:numCellArrays
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', cellArrayNames{k});

    % Store lengths of current cell array
    currentLengths = cellfun(@length, currentCellArray);
    allLengths(1:length(currentLengths), k) = currentLengths;
end

% Compare lengths across cell arrays for each array position and report
% non-equal lengths
discrepancies = false;
for i = 1:maxArrays
    uniqueLengths = unique(allLengths(i, allLengths(i, :) > 0));
    if length(uniqueLengths) > 1
        discrepancies = true;
        fprintf('Non-equal lengths found at array position %d:\n', i);
        for k = 1:numCellArrays
            if allLengths(i, k) > 0
                fprintf('  %s has array length %d.\n', ...
                    cellArrayNames{k}, allLengths(i, k));
            end
        end
    end
end

% Display overall result
if discrepancies
    disp(['There are discrepancies in array lengths ', ...
      'across the cell arrays.']);
else
    disp(['All corresponding arrays across the cell arrays ', ...
        'have equal lengths.']);
end

%%
% Vertically stack arrays of flight waveforms.

% Concatenate Training Data
allTrainingData = [concatenateArrays(trainingData_CuA); ...
    concatenateArrays(trainingData_CuL); ...
    concatenateArrays(trainingData_CuS);
                   concatenateArrays(trainingData_MoA); ...
                   concatenateArrays(trainingData_MoL); ...
                   concatenateArrays(trainingData_MoS);
                   concatenateArrays(trainingData_LaA); ...
                   concatenateArrays(trainingData_LaL); ...
                   concatenateArrays(trainingData_LaS)];

% Concatenate Testing Data
allTestingData = [concatenateArrays(testingData_CuA); ...
    concatenateArrays(testingData_CuL); ...
    concatenateArrays(testingData_CuS);
                  concatenateArrays(testingData_MoA); ...
                  concatenateArrays(testingData_MoL); ...
                  concatenateArrays(testingData_MoS);
                  concatenateArrays(testingData_LaA); ...
                  concatenateArrays(testingData_LaL); ...
                  concatenateArrays(testingData_LaS)];

%%
% Generates label vectors for training and testing datasets

% Defines a list of labels
labels = {'CuA', 'CuL', 'CuS', 'MoA', 'MoL', 'MoS', 'LaA', 'LaL', 'LaS'};
Y_train = [];
Y_test = [];
for i = 1:length(labels)
    % Assuming each training cell array contains 80 arrays
    Y_train = [Y_train; repmat(labels(i), 80, 1)];
    % Assuming each testing cell array contains 20 arrays
    Y_test = [Y_test; repmat(labels(i), 20, 1)];
end

%%
%Standardize Training and Testing Data
meanTraining = mean(allTrainingData, 1);
stdTraining = std(allTrainingData, 0, 1);

% Standardize training data
allTrainingDataStandardized = (allTrainingData - meanTraining) ./ ...
    stdTraining;

% Standardize testing data
allTestingDataStandardized = (allTestingData - meanTraining) ./ ...
    stdTraining;

%%
% KNN PCA classification

% Define parameters
maxComponents = 200; % Maximum number of PCA components to consider
kValues = [3, 5, 10, 15]; % Different k values for KNN
```

```
distanceMetrics = {'cosine', 'cityblock', 'correlation', 'euclidean'};
markers = {'o', 's', 'p', 'd', '*'};

% Initialize structure for accuracies and max accuracy details
accuracies = struct();
maxAccuracyDetails = struct();

% Loop over distance metrics
for dIndex = 1:length(distanceMetrics)
    distanceMetric = distanceMetrics{dIndex};

    % Initialize accuracy matrix and max accuracy details for each
    % k value
    accuracies.(distanceMetric) = zeros(maxComponents, length(kValues));
    for kIndex = 1:length(kValues)
        maxAccuracyDetails.(distanceMetric).(...
            ['k' num2str(kValues(kIndex))]) = ...
            struct('Component', 0, 'Accuracy', 0);
    end

    % Loop over the number of components
    for numComponents = 1:maxComponents
        % Apply PCA
        [coeff, score] = pca(allTrainingDataStandardized, ...
            'NumComponents', numComponents);
        X_train_pca = score;
        X_test_pca = allTestingDataStandardized * ...
            coeff(:, 1:numComponents);

        % Loop over different values of k
        for kIndex = 1:length(kValues)
            k = kValues(kIndex);
            disp(['Distance: ' distanceMetric ', K: ' num2str(k) ...
                ', PCA component: ' num2str(numComponents)]);

            % Train KNN Classifier
            knnModel = fitcknn(X_train_pca, Y_train, 'NumNeighbors', ...
                k, 'Distance', distanceMetric);

            % Test Classifier and calculate accuracy
            Y_pred = predict(knnModel, X_test_pca);
            accuracy = sum(strcmp(Y_pred, Y_test)) / length(Y_test);

            % Store and update accuracy
            accuracies.(distanceMetric)(numComponents, kIndex) = ...
                accuracy;
            if accuracy > maxAccuracyDetails.(distanceMetric).(...
                    ['k' num2str(k)]).Accuracy
                maxAccuracyDetails.(distanceMetric).(...
                    ['k' num2str(k)]).Accuracy = accuracy;
                maxAccuracyDetails.(distanceMetric).(...
                    ['k' num2str(k)]).Component = numComponents;
            end
        end
    end

    % Plotting
    figure;
    hold on;

    for kIndex = 1:length(kValues)
        yData = accuracies.(distanceMetric)(:, kIndex);
        xData = 1:maxComponents;

        % Calculate the second derivative (curvature)
        d2y = abs(diff(yData, 2));
        % Append zeros at the ends since diff reduces the length by 2
        d2y = [0; d2y; 0];

        % Define a threshold for the curvature
        curvatureThreshold = 0.01;  % Adjust based on your data
        minSpacing = 5; % Minimum spacing between markers
        markerIndices = [];
        lastAddedIndex = 1;

        for i = 1:length(yData)
            if d2y(i) > curvatureThreshold || (i - lastAddedIndex) ...
                    >= minSpacing
                markerIndices = [markerIndices, i];
                lastAddedIndex = i;
            end
        end

        % Combine line and markers in one plot command
        p = plot(xData, yData, 'Color', 'k', 'DisplayName', ...
            ['k = ' num2str(kValues(kIndex))]);
        set(p, 'Marker', markers{kIndex}, 'MarkerIndices', ...
            markerIndices);
    end

    hold off;
    xlabel('Number of Principal Components');
    ylabel('Accuracy');
    title(['Accuracy vs Number of Principal Components '...
       '(Distance Metric: ' distanceMetric ')']);
    legend('Location', 'southeast');
    ylim([0.68, 0.82]);
end


%%
% Create summary table and display the summary table

% Create summary table
% Initialize the arrays and the cell array
distanceMetricsArray = {};
kValuesArray = [];
componentsArray = [];
accuraciesArray = [];

% Populate the arrays
for dIndex = 1:length(distanceMetrics)
    distanceMetric = distanceMetrics{dIndex};
    for kIndex = 1:length(kValues)
        k = kValues(kIndex);
        maxAccDetail = maxAccuracyDetails.(distanceMetric).(...
            ['k' num2str(k)]);

        distanceMetricsArray{end+1} = distanceMetric;
        kValuesArray(end+1) = k;
        componentsArray(end+1) = maxAccDetail.Component;
        accuraciesArray(end+1) = maxAccDetail.Accuracy;
    end
end

% Display the summary table
summaryTable = table(distanceMetricsArray', kValuesArray', ...
    componentsArray', accuraciesArray', ...
                     'VariableNames', {'DistanceMetric', 'KValue', ...
                     'Components', 'Accuracy'});
disp(summaryTable);

%%
% Store and display the optimal K and the optimal principal components

% Initialize structure to store the best parameters
bestParameters = struct();

% Loop over distance metrics to find the best parameters
for dIndex = 1:length(distanceMetrics)
    distanceMetric = distanceMetrics{dIndex};
    maxAccuracy = 0; % Initialize the maximum accuracy

    % Loop over the number of components
    for numComponents = 1:maxComponents
        for kIndex = 1:length(kValues)
            k = kValues(kIndex);
            accuracy = accuracies.(distanceMetric)(numComponents, kIndex);

            % Check if current accuracy is greater than the max
            % found so far
            if accuracy > maxAccuracy
                maxAccuracy = accuracy;
                bestParameters.(distanceMetric).KValue = k;
                bestParameters.(distanceMetric).Component = numComponents;
            end
        end
    end
end

% Display the best parameters for each distance metric
disp('Best Parameters for each Distance Metric:');
for dIndex = 1:length(distanceMetrics)
    distanceMetric = distanceMetrics{dIndex};
    fprintf('%s: k = %d, Components = %d, Accuracy = %.4f\n', ...
            distanceMetric, bestParameters.(distanceMetric).KValue, ...
            bestParameters.(distanceMetric).Component, ...
            maxAccuracyDetails.(distanceMetric).(...
            ['k' num2str(bestParameters.(distanceMetric).KValue)]...
            ).Accuracy);
end

% The bestParameters contains the best k value and component for
% each distance metric

%%
% Rerun the PCA-KNN analysis using the optimal K and principal components
% for each distance metric to generate data for the confusion matrices

classOrder = unique(Y_train);
confusionMatrices = struct();
labelToIndexMap = containers.Map(classOrder, 1:length(classOrder));

% Loop through distance metrics
for dIndex = 1:length(distanceMetrics)
    distanceMetric = distanceMetrics{dIndex};
    bestK = bestParameters.(distanceMetric).KValue;
    bestComponent = bestParameters.(distanceMetric).Component;

    % Apply PCA
    [coeff, ~] = pca(allTrainingDataStandardized, 'NumComponents', ...
        bestComponent);
    X_train_pca = allTrainingDataStandardized * coeff(:, 1:bestComponent);
    X_test_pca = allTestingDataStandardized * coeff(:, 1:bestComponent);

    % Train KNN Classifier
    knnModel = fitcknn(X_train_pca, Y_train, 'NumNeighbors', bestK, ...
        'Distance', distanceMetric);

    % Predict responses
    Y_pred = predict(knnModel, X_test_pca);

    % Initialize confusion matrix
    C_manual = zeros(length(classOrder), length(classOrder));

    % Populate confusion matrix
    for i = 1:length(Y_test)
        trueLabel = Y_test{i};
        predictedLabel = Y_pred{i};
        if isKey(labelToIndexMap, trueLabel) && isKey(labelToIndexMap, ...
                predictedLabel)
            trueIndex = labelToIndexMap(trueLabel);
            predIndex = labelToIndexMap(predictedLabel);
            C_manual(predIndex, trueIndex) = ...
                C_manual(predIndex, trueIndex) + 1;
        end
    end

    % Store confusion matrix in the 'confusionMatrices'
    confusionMatrices.(distanceMetric).(['k' num2str(bestK)]).(...
        ['Comp' num2str(bestComponent)]) = C_manual;
end

%%
% Calculates the recognition percentages,
% and displays the confusion matrices with enhanced graphical details

% These confusion matrices and the corresponding recognition percentages
% indicate the accuracy in identifying birds' flight patterns with
% directions

% Define arrays with the names of training and testing matrices for
% different categories
trainingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};
testingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};

% Calls the function to extract specific confusion matrices for various
% distance metrics from the 'confusionMatrices' structure
confusionMatrices_extract = extractConfusionMatrices(confusionMatrices);
for metric = distanceMetrics
    % Transpose the matrix to have predicted classes on the y-axis
    % and actual classes on the x-axis
    cm = confusionMatrices_extract.(metric{1});

    % Calculate the percent of correct classifications for each
    % actual class
    sumPerColumn = sum(cm, 1);  % Total count for each actual class
    % Percentage of correct predictions
    percentCorrect = diag(cm) ./ sumPerColumn' * 100;
    % Append the percentage row to the confusion matrix
    cmWithPercent = [cm; percentCorrect'];

    % Create a new figure for the confusion matrix visualization
    figure;
    imagesc(cmWithPercent);  % Display the confusion matrix as an image
    % Set the colormap to 'parula' (you can choose a different colormap)
    colormap(parula);
    % Lighten the colormap for aesthetic purposes
    currentColormap = colormap;
    % Adjust the colormap to be lighter
    colormap(currentColormap + 0.7 * (1 - currentColormap));

    % Set titles and labels
    title(['Confusion Matrix for ' metric{1} ' Distance']);
    xlabel('Actual');
    ylabel('Predicted');
    set(gca, 'XTick', 1:length(testingMatrixNames) + 1, 'XTickLabel', ...
        [testingMatrixNames, {'% Correct'}]);
    set(gca, 'YTick', 1:length(trainingMatrixNames) + 1, 'YTickLabel', ...
        [trainingMatrixNames, {'% Correct'}]);

    % Add numerical labels to each cell of the matrix
    [rows, cols] = size(cmWithPercent);
    for i = 1:rows
        for j = 1:cols
            num = cmWithPercent(i, j);  % Get the cell value
            text(j, i, num2str(num, '%0.0f'), 'HorizontalAlignment', ...
                'center', 'VerticalAlignment', 'middle');
        end
    end

    % Rotate the x-axis labels for better readability if there are
    % many classes
    if length(trainingMatrixNames) > 10
        set(gca, 'XTickLabelRotation', 45);
    end
end

%%
% Groups the 9x9 confusion matrix into a smaller 3x3 matrix by summing
% values in groups of three, and calculates the recognition percentages
% Display confusion matrices in the MATLAB command window as textual output

% These confusion matrices and the corresponding recognition percentages
% indicate the accuracy in identifying birds' flight patterns without
% directions

% Define distance metrics used in the analysis
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};

% Define labels for the groups
groupLabels = {'Cu', 'Mo', 'La'};

% Process each confusion matrix for each metric
for metricIdx = 1:length(distanceMetrics)
    metric = distanceMetrics{metricIdx};
    fprintf('Processing for %s Distance:\n', metric);

    % Get the field names for the current metric
    % Get the 'k' field name
    kField = fieldnames(confusionMatrices.(metric));
    % Get the 'Comp' field name
    compField = fieldnames(confusionMatrices.(metric).(kField{1}));

    % Access the confusion matrix
    cm = confusionMatrices.(metric).(kField{1}).(compField{1});

    % Transpose the confusion matrix if necessary
    cmT = cm;

    % Sum by groups of three
    groupedSum = zeros(3, 3);
    for i = 1:3:size(cmT, 1)
        for j = 1:3:size(cmT, 2)
            groupedSum((i-1)/3 + 1, (j-1)/3 + 1) = ...
                sum(sum(cmT(i:i+2, j:j+2)));
        end
    end

    % Calculate the recognition percentage for each column
    recognitionPercentages = zeros(1, size(groupedSum, 2));
    for col = 1:size(groupedSum, 2)
        diagonalElement = groupedSum(col, col);  % Diagonal element
        columnSum = sum(groupedSum(:, col));  % Sum of the column
        recognitionPercentages(col) = (diagonalElement / columnSum) * 100;
    end

    % Display the grouped sums and recognition percentages
    disp(['Grouped Sums for ' metric ':']);
    disp(groupedSum);
    disp(['Recognition Percentages (%) for ' metric ':']);
    for i = 1:length(recognitionPercentages)
        fprintf('%s: %.2f%%\n', groupLabels{i}, ...
            recognitionPercentages(i));
    end

    fprintf('\n');
end

%%
% Groups the 9x9 confusion matrix into a smaller 3x3 matrix by summing
% values in groups of three, calculates the recognition percentages,
% and displays the confusion matrices with enhanced graphical details

% These confusion matrices and the corresponding recognition percentages
% indicate the accuracy in identifying birds' flight patterns without
% directions

% Define distance metrics used in the analysis
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};
```

```
% Define labels for the groups
groupLabels = {'Cu', 'Mo', 'La'};

% Process and visualize each confusion matrix for each metric
for metricIdx = 1:length(distanceMetrics)
    metric = distanceMetrics{metricIdx};

    % Get the field names for the current metric
    % Get the 'k' field name
    kField = fieldnames(confusionMatrices.(metric));
    % Get the 'Comp' field name
    compField = fieldnames(confusionMatrices.(metric).(kField{1}));

    % Access the confusion matrix
    cm = confusionMatrices.(metric).(kField{1}).(compField{1});

    % Sum by groups of three
    groupedSum = zeros(3, 3);
    for i = 1:3:size(cm, 1)
        for j = 1:3:size(cm, 2)
            groupedSum((i-1)/3 + 1, (j-1)/3 + 1) = ...
                sum(sum(cm(i:i+2, j:j+2)));
        end
    end

    % Calculate the recognition percentage for each column
    recognitionPercentages = zeros(1, size(groupedSum, 2));
    for col = 1:size(groupedSum, 2)
        diagonalElement = groupedSum(col, col);  % Diagonal element
        columnSum = sum(groupedSum(:, col));  % Sum of the column
        recognitionPercentages(col) = (diagonalElement / columnSum) * 100;
    end

    % Create figure for visualization of the small matrices
    figure;
    % Append recognition percentages as a new row
    cmWithPercent = [groupedSum; recognitionPercentages];
    % Display matrix with recognition percentages
    imagesc(cmWithPercent);
    colormap(parula);  % Choose colormap

    % Lighten the colormap
    currentColormap = colormap;
    colormap(currentColormap + 0.7 * (1 - currentColormap));

    % Set titles and labels
    title(['Confusion Matrix for ' metric ' Distance']);
    xlabel('Actual');
    ylabel('Predicted');
    set(gca, 'XTick', 1:4, 'XTickLabel', [groupLabels, {'% Correct'}]);
    set(gca, 'YTick', 1:4, 'YTickLabel', [groupLabels, {'% Correct'}]);
    % Add numerical labels to each cell
    [rows, cols] = size(cmWithPercent);
    for i = 1:rows
        for j = 1:cols
            num = cmWithPercent(i, j);
            if i == rows  % For the last row (recognition percentages)
                text(j, i, [num2str(num, '%0.2f') '%'], ...
                    'HorizontalAlignment', 'center', ...
                    'VerticalAlignment', 'middle');
            else  % For the grouped sum matrix
                text(j, i, num2str(num, '%0.0f'), ...
                    'HorizontalAlignment', 'center', ...
                    'VerticalAlignment', 'middle');
            end
        end
    end
end
%%
%END OF 'PCA_KNN' CODE

function confusionMatrices_extract = ...
    extractConfusionMatrices(confusionMatrices)
 % This function extracts specific confusion matrices from a structured
    % collection.
    %
    % This function takes a structured array containing confusion matrices
    % for various distance metrics and K values in a K-Nearest Neighbors
    % (KNN) classification setting. It extracts the confusion matrix for
    % each distance metric using the first K value and the first set of
    % components specified in the structure.
    %
    % The function is designed to work with a structured array where each
    % field corresponds to a different distance metric, and each subfield
    % corresponds to different K values and PCA component counts.
    %
    % Input:
    % confusionMatrices - A structured array containing confusion matrices
    %                     for different distance metrics, K values, and PCA
    %                     components.
    %
    % Output:
    % confusionMatrices_extract - A structured array with extracted confusion
    %                             matrices for each distance metric. The
    %                             extracted matrix corresponds to the first
    %                             K value andthe first set of components for
    %                             each metric.

    % Extract the confusion matrix for cosine distance
    cosineFields = fieldnames(confusionMatrices.cosine);
    kCosine = cosineFields{1};
    compCosineFields = fieldnames(confusionMatrices.cosine.(kCosine));
    compCosine = compCosineFields{1};
    confusionMatrices_extract.cosine = ...
```

```
        confusionMatrices.cosine.(kCosine).(compCosine);

    % Extract the confusion matrix for cityblock distance
    cityblockFields = fieldnames(confusionMatrices.cityblock);
    kCityblock = cityblockFields{1};
    compCityblockFields = ...
        fieldnames(confusionMatrices.cityblock.(kCityblock));
    compCityblock = compCityblockFields{1};
    confusionMatrices_extract.cityblock = ...
        confusionMatrices.cityblock.(kCityblock).(compCityblock);

    % Extract the confusion matrix for correlation distance
    correlationFields = fieldnames(confusionMatrices.correlation);
    kCorrelation = correlationFields{1};
    compCorrelationFields = ...
        fieldnames(confusionMatrices.correlation.(kCorrelation));
    compCorrelation = compCorrelationFields{1};
    confusionMatrices_extract.correlation = ...
        confusionMatrices.correlation.(kCorrelation).(compCorrelation);

    % Extract the confusion matrix for euclidean distance
    euclideanFields = fieldnames(confusionMatrices.euclidean);
    kEuclidean = euclideanFields{1};
    compEuclideanFields = ...
        fieldnames(confusionMatrices.euclidean.(kEuclidean));
    compEuclidean = compEuclideanFields{1};
    confusionMatrices_extract.euclidean = ...
        confusionMatrices.euclidean.(kEuclidean).(compEuclidean);
end
% END OF 'extractConfusionMatrices' FUNCTION

function concatenatedMatrix = concatenateArrays(cellArray)
    % This function concatenates waveforms within a cell array into a single
    % matrix.
    %
    % This function takes a cell array containing waveforms and concatenates
   them
    % vertically into a single matrix. The waveforms within the cell array are
    % expected to have the same number of columns but can have different
    % numbers of rows.
    %
    % Input:
    % cellArray - A cell array where each waveform is an array to be
    % concatenated.
    %
    % Output:
    % concatenatedMatrix - A matrix formed by vertically concatenating the
    %                       waveforms from the cell array.
    concatenatedMatrix = [];
    for i = 1:length(cellArray)
        concatenatedMatrix = [concatenatedMatrix; cellArray{i}];
    end
end
```

```
% END OF 'concatenateArrays' FUNCTION
```

A.1.3. MATLAB Code Segment III - Cross-Correlation Classification

In this final segment, cross-correlation techniques are utilized to classify bird flight patterns.

```
% Cross_Corr.m
% This script performs cross-correlation analysis on bird flight data.
%
% Steps:
% 1. Data Loading: Loads preprocessed bird flight data from a .mat file.
% 2. Length Check: Verifies if all arrays in the dataset have equal
%    lengths.
% 3. Zero-Padding: Adds padding to facilitate cross-correlation analysis.
% 4. Data Plotting: Plots training and testing data after padding.
% 5. Cross-Correlation Analysis: Calculates the smallest distance between
%    datasets using various distance metrics (cityblock, euclidean,
%    correlation, cosine).
% 6. Confusion Matrix: Constructs and presents confusion matrices for
%    each distance metric, and calculates the recognition percentages.
%    These are the confusion matrices for birds flying with direction.
% 7. Group the confusion matrices for birds flying with direction to
%    form the confusion matrices for birds flying without direction, and
%    calculate the recognition percentages.

%%
clc;        % Clears the Command Window
close all;  % Closes all open figures
clear;      % Removes all variables from the workspace

% Load data from .mat file
load('preProcBirdsFliesData_rng(4).mat');

%%
% Check if the lengths of all arrays are equal

% Array of cell array names
cellArrayNames = {'trainingData_CuA', 'trainingData_CuL', ...
    'trainingData_CuS', 'trainingData_MoA', 'trainingData_MoL', ...
    'trainingData_MoS', 'trainingData_LaA', 'trainingData_LaL', ...
    'trainingData_LaS', 'testingData_CuA', 'testingData_CuL', ...
    'testingData_CuS', 'testingData_MoA', 'testingData_MoL', ...
    'testingData_MoS', 'testingData_LaA', 'testingData_LaL', ...
    'testingData_LaS'};

% Determine the maximum number of arrays in any cell array
maxArrays = 0;
for k = 1:length(cellArrayNames)
```

```
    numArrays = length(evalin('base', cellArrayNames{k}));
    maxArrays = max(maxArrays, numArrays);
end

% Initialize a matrix to store lengths from all cell arrays
numCellArrays = length(cellArrayNames);
allLengths = zeros(maxArrays, numCellArrays);

% Loop through each cell array name
for k = 1:numCellArrays
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', cellArrayNames{k});

    % Store lengths of current cell array
    currentLengths = cellfun(@length, currentCellArray);
    allLengths(1:length(currentLengths), k) = currentLengths;
end

% Compare lengths across cell arrays for each array position and report
% non-equal lengths
discrepancies = false;
for i = 1:maxArrays
    uniqueLengths = unique(allLengths(i, allLengths(i, :) > 0));
    if length(uniqueLengths) > 1
        discrepancies = true;
        fprintf('Non-equal lengths found at array position %d:\n', i);
        for k = 1:numCellArrays
            if allLengths(i, k) > 0
                fprintf('  %s has array length %d.\n', ...
                    cellArrayNames{k}, allLengths(i, k));
            end
        end
    end
end

% Display overall result
if discrepancies
    disp(['There are discrepancies in array lengths ', ...
      'across the cell arrays.']);
else
    disp(['All corresponding arrays across the cell arrays ', ...
        'have equal lengths.']);
end

%%
% Padding zeros to facilitate the cross-correlation method

paddingSize = 4000;

% Array of original training cell array names
trainingNames = {'trainingData_CuA', 'trainingData_CuL', ...
    'trainingData_CuS', 'trainingData_MoA', 'trainingData_MoL', ...
    'trainingData_MoS', 'trainingData_LaA', 'trainingData_LaL', ...
    'trainingData_LaS'};
testingNames = {'testingData_CuA', 'testingData_CuL', ...
    'testingData_CuS', 'testingData_MoA', 'testingData_MoL', ...
    'testingData_MoS', 'testingData_LaA', 'testingData_LaL', ...
    'testingData_LaS'};

% Function for padding single-row arrays in training data
paddingFunctionTraining = @(array) [zeros(1, paddingSize), array, ...
    zeros(1, paddingSize)];

% Function for padding single-row arrays in testing data
paddingFunctionTesting = @(array) [array, zeros(1, paddingSize), ...
    zeros(1, paddingSize)];

% Loop through each training cell array name and pad
for k = 1:length(trainingNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', trainingNames{k});

    % Add padding to each single-row array within the cell array
    paddedCellArray = cellfun(paddingFunctionTraining, ...
        currentCellArray, 'UniformOutput', false);

    % Store the padded cell array in the base workspace
    assignin('base', ['padded_' trainingNames{k}], paddedCellArray);
end

% Loop through each testing cell array name and pad
for k = 1:length(testingNames)
    % Load the current cell array from the base workspace
    currentCellArray = evalin('base', testingNames{k});

    % Add padding to each single-row array within the cell array
    paddedCellArray = cellfun(paddingFunctionTesting, ...
        currentCellArray, 'UniformOutput', false);

    % Store the padded cell array in the base workspace
    assignin('base', ['padded_' testingNames{k}], paddedCellArray);
end

%%
% Plot the training and testing data following zero-padding

% Array of padded training and testing cell array names
paddedTrainingNames = {'padded_trainingData_CuA', ...
    'padded_trainingData_CuL', 'padded_trainingData_CuS', ...
    'padded_trainingData_MoA', 'padded_trainingData_MoL', ...
    'padded_trainingData_MoS', 'padded_trainingData_LaA', ...
    'padded_trainingData_LaL', 'padded_trainingData_LaS'};
paddedTestingNames = {'padded_testingData_CuA', ...
    'padded_testingData_CuL', 'padded_testingData_CuS', ...
    'padded_testingData_MoA', 'padded_testingData_MoL', ...
    'padded_testingData_MoS', 'padded_testingData_LaA', ...
    'padded_testingData_LaL', 'padded_testingData_LaS'};

% Loop through each pair of padded training and testing cell array names
for k = 1:length(paddedTrainingNames)
    % Load the current padded training cell array from the base workspace
    currentPaddedTrainingArray = evalin('base', paddedTrainingNames{k});
    % Load the current padded testing cell array from the base workspace
    currentPaddedTestingArray = evalin('base', paddedTestingNames{k});

    % Number of arrays in the current training and testing cell arrays
    numTrainingArrays = length(currentPaddedTrainingArray);
    numTestingArrays = length(currentPaddedTestingArray);

    % Create a new figure for each pair of cell arrays
    figure;

    % Subplot for the training data
    subplot(2, 1, 1);
    hold on;
    for i = 1:numTrainingArrays
        plot(currentPaddedTrainingArray{i});
    end
    hold off;
    title(['Training Data - ' paddedTrainingNames{k}]);

    % Subplot for the testing data
    subplot(2, 1, 2);
    hold on;
    for i = 1:numTestingArrays
        plot(currentPaddedTestingArray{i});
    end
    hold off;
    title(['Testing Data - ' paddedTestingNames{k}]);
end

%%
% Find the smallest distance using cross-correlation

% Define the matrix names for training and testing data
trainingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};
testingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};

% Define distance metrics to be used for comparison
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};

% Initialize variables for shift calculations
shift_size = 100;
shift_total = floor(paddingSize / shift_size);
```

```
% Initialize a structure to store the results of distance calculations
distanceResults = struct();

% Loop through each testing dataset
for testMatrixIndex = 1:length(testingMatrixNames)
    % Load the current testing data set
    testingDataName = ['padded_testingData_' ...
        testingMatrixNames{testMatrixIndex}];
    testingData = evalin('base', testingDataName);

    % Loop through each row of the testing data
    for rowIndex = 1:length(testingData)
        disp(['Testing Data Name: ' testingDataName ', Row: ' ...
            num2str(rowIndex)]);  % Display current testing data and row
        testDataRow = testingData{rowIndex};

        % Loop through each distance metric
        for metricIndex = 1:length(distanceMetrics)
            metric = distanceMetrics{metricIndex};

            % Initialize arrays to store the smallest distance for each
            % training dataset
            minDistancesPerDataset = inf(length(trainingMatrixNames), 1);

            % Perform distance calculations for each shift of the
            % test data
            for shift = 0:shift_total
                % Shift the test data row
                shiftedTestDataRow = [zeros(1, shift * shift_size), ...
                    testDataRow];
                if length(shiftedTestDataRow) > length(testDataRow)
                    shiftedTestDataRow = shiftedTestDataRow(1:length(...
                        testDataRow));
                end

                % Loop through each training dataset for
                % distance calculations
                for trainingMatrixIndex = 1:length(trainingMatrixNames)
                    % Load the current training data set
                    trainingDataName = ['padded_trainingData_' ...
                        trainingMatrixNames{trainingMatrixIndex}];
                    trainingData = evalin('base', trainingDataName);

                    % Loop through each array in the current training data
                    for trainingRowIndex = 1:length(trainingData)
                        trainingRow = trainingData{trainingRowIndex};

                        % Calculate the distance
                        distance = pdist2(shiftedTestDataRow, ...
                            trainingRow, metric);
                        % Update the minimum distance if necessary
                        if distance < ...
                               minDistancesPerDataset(trainingMatrixIndex)
                           minDistancesPerDataset(trainingMatrixIndex) ...
                                = distance;
                        end
                    end
                end
            end

            % Find the smallest distance across all datasets for
            % the current metric
            [smallestDistance, datasetIndex] = ...
                min(minDistancesPerDataset);

            % Store the result in the distanceResults structure
            resultField = sprintf('%s_%s_Row%d', metric, ...
                testingMatrixNames{testMatrixIndex}, rowIndex);
            distanceResults.(resultField) = struct('SmallestDistance', ...
                smallestDistance, 'TrainingDataSet', ...
                trainingMatrixNames{datasetIndex});
        end
    end
end

%%
% Display confusion matrices in the MATLAB command window as textual output
% These confusion matrices indicate the recognition of birds' flying
% directions

% Define the matrix names for training and testing data
trainingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};
testingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA', ...
    'LaL', 'LaS', 'LaA'};

% Define distance metrics to be used in the analysis
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};

% Initialize confusion matrices for each metric
confusionMatrices = struct();
for metric = distanceMetrics
    confusionMatrices.(metric{1}) = zeros(length(testingMatrixNames), ...
        length(trainingMatrixNames));
end

% Mapping from dataset names to indices for confusion matrix construction
nameToIndex = containers.Map(trainingMatrixNames, ...
    1:length(trainingMatrixNames));

% Assuming distanceResults is available and contains the
% necessary information
% Extract and process results for each metric
for metric = distanceMetrics
    % Retrieve all relevant field names for the current metric
    % from distanceResults
    metricFieldNames = fieldnames(distanceResults);
    metricFieldNames = metricFieldNames(contains(metricFieldNames, ...
        metric{1}));

    % Process each field name to update the confusion matrix
    for i = 1:length(metricFieldNames)
        fieldName = metricFieldNames{i};
        % Extract the actual dataset (true label) and the predicted
        % dataset label
        trueLabel = extractBetween(fieldName, '_', '_Row');
        predictedLabel = distanceResults.(fieldName).TrainingDataSet;

        % Format the predicted label to match the indexing format
        predictedLabel = strrep(predictedLabel, 'Padded', '');
        predictedLabel = strrep(predictedLabel, 'trainingData_', '');

        % Update the corresponding cell in the confusion matrix
        trueIndex = nameToIndex(trueLabel{1});
        predictedIndex = nameToIndex(predictedLabel);
        confusionMatrices.(metric{1})(trueIndex, predictedIndex) = ...
            confusionMatrices.(metric{1})(trueIndex, predictedIndex) + 1;
    end
end

% Display each confusion matrix for the respective distance metric
for metric = distanceMetrics
    fprintf('Confusion Matrix for %s Distance:\n', metric{1});
    disp(confusionMatrices.(metric{1}));
end

%%
% Calculates the recognition percentages,
% and displays the confusion matrices with enhanced graphical details

% These confusion matrices indicate the recognition of birds' flying
% directions

% Define the matrix names for training and testing data
trainingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA',...
    'LaL', 'LaS', 'LaA'};
testingMatrixNames = {'CuL', 'CuS', 'CuA', 'MoL', 'MoS', 'MoA',...
    'LaL', 'LaS', 'LaA'};

% Assuming confusionMatrices structure is already populated
% and distanceMetrics array is defined
for metric = distanceMetrics
    % Transpose the matrix
    cm = confusionMatrices.(metric{1})';
    % Calculate percent correct
    sumPerColumn = sum(cm, 1);  % Total count for each actual class
    % Percentage of correct predictions
    percentCorrect = diag(cm) ./ sumPerColumn' * 100;
    cmWithPercent = [cm; percentCorrect'];  % Append percentage row

    % Create figure for visualization
    figure;
    imagesc(cmWithPercent);  % Display matrix
    colormap(parula);  % Choose colormap

    % Lighten the colormap
    currentColormap = colormap;
    colormap(currentColormap + 0.7 * (1 - currentColormap));

    % Set titles and labels
title(['Confusion Matrix for ' metric{1} ' Distance']);
xlabel('Actual');
ylabel('Predicted');
set(gca, 'XTick', 1:length(testingMatrixNames) + 1, 'XTickLabel', ...
    [testingMatrixNames, {'% Correct'}]);
set(gca, 'YTick', 1:length(trainingMatrixNames) + 1, 'YTickLabel', ...
    [trainingMatrixNames, {'% Correct'}]);


    % Add numerical labels to each cell
    [rows, cols] = size(cmWithPercent);
    for i = 1:rows
        for j = 1:cols
            num = cmWithPercent(i, j);
            text(j, i, num2str(num, '%0.0f'), 'HorizontalAlignment', ...
                'center', 'VerticalAlignment', 'middle');
        end
    end

    % Rotate x-axis labels for readability
    if length(trainingMatrixNames) > 10
        set(gca, 'XTickLabelRotation', 45);
    end
end

%%
% Group the 9x9 confusion matrices into 3x3 matrices and calculate the
% recognition percentages
% Display confusion matrices in the MATLAB command window as textual output
% These confusion matrices indicate the recognition of birds' flying
% without directions

% Define the matrix names
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};
```

```
% Define labels for the groups
groupLabels = {'Cu', 'Mo', 'La'};

% Process each confusion matrix for each metric
for metric = distanceMetrics
    fprintf('Processing for %s Distance:\n', metric{1});

    % Access the confusion matrix for the current metric
    cm = confusionMatrices.(metric{1});

    % Transpose the confusion matrix
    cmT = cm';

    % Sum by groups of three
    groupedSum = zeros(3, 3);
    for i = 1:3:size(cmT, 1)
        for j = 1:3:size(cmT, 2)
            groupedSum((i-1)/3 + 1, (j-1)/3 + 1) = ...
                sum(sum(cmT(i:i+2, j:j+2)));
        end
    end

    % Calculate the recognition percentage for each column
    recognitionPercentages = zeros(1, size(groupedSum, 2));
    for col = 1:size(groupedSum, 2)
        diagonalElement = groupedSum(col, col);  % Diagonal element
        columnSum = sum(groupedSum(:, col));  % Sum of the column
        recognitionPercentages(col) = ...
            (diagonalElement / columnSum) * 100;
    end

    % Display the grouped sums and recognition percentages
    disp(['Grouped Sums for ' metric{1} ':']);
    disp(groupedSum);
    disp(['Recognition Percentages (%) for ' metric{1} ':']);
    for i = 1:length(recognitionPercentages)
        fprintf('%s: %.2f%%\n', groupLabels{i}, ...
            recognitionPercentages(i));
    end
    fprintf('\n');
end

%%
% Groups the 9x9 confusion matrix into a smaller 3x3 matrix by summing
% values in groups of three, calculates the recognition percentages,
% and displays the confusion matrices with enhanced graphical details

% These confusion matrices indicate the recognition of birds' flying
% without directions

% Define distance metrics used in the analysis
distanceMetrics = {'cityblock', 'euclidean', 'correlation', 'cosine'};
```

```
% Define labels for the groups
groupLabels = {'Cu', 'Mo', 'La'};

% Process and visualize each confusion matrix for each metric
for metric = distanceMetrics
    %fprintf('Processing and Visualizing for %s Distance:\n', metric{1});

    % Access the confusion matrix for the current metric
    cm = confusionMatrices.(metric{1});

    % Transpose the confusion matrix
    cmT = cm';

    % Sum by groups of three
    groupedSum = zeros(3, 3);
    for i = 1:3:size(cmT, 1)
        for j = 1:3:size(cmT, 2)
            groupedSum((i-1)/3 + 1, (j-1)/3 + 1) = ...
                sum(sum(cmT(i:i+2, j:j+2)));
        end
    end

    % Calculate the recognition percentage for each column
    recognitionPercentages = zeros(1, size(groupedSum, 2));
    for col = 1:size(groupedSum, 2)
        diagonalElement = groupedSum(col, col);  % Diagonal element
        columnSum = sum(groupedSum(:, col));  % Sum of the column
        recognitionPercentages(col) = (diagonalElement / columnSum) * 100;
    end

    % Create figure for visualization of the small matrices
    figure;
    % Append recognition percentages as a new row
    cmWithPercent = [groupedSum; recognitionPercentages];
    % Display matrix with recognition percentages
    imagesc(cmWithPercent);
    colormap(parula);  % Choose colormap

    % Lighten the colormap
    currentColormap = colormap;
    colormap(currentColormap + 0.7 * (1 - currentColormap));

    % Set titles and labels
    title(['Confusion Matrix for ' metric{1} ' Distance']);
    xlabel('Actual');
    ylabel('Predicted');
    set(gca, 'XTick', 1:4, 'XTickLabel', [groupLabels, {'% Correct'}]);
    set(gca, 'YTick', 1:4, 'YTickLabel', [groupLabels, {'% Correct'}]);

    % Add numerical labels to each cell
    [rows, cols] = size(cmWithPercent);
    for i = 1:rows
        for j = 1:cols
            num = cmWithPercent(i, j);
            if i == rows  % For the last row (recognition percentages)
                text(j, i, [num2str(num, '%0.2f') '%'], ...
                    'HorizontalAlignment', 'center', ...
                    'VerticalAlignment', 'middle');
            else  % For the grouped sum matrix
                text(j, i, num2str(num, '%0.0f'), ...
                    'HorizontalAlignment', 'center', ...
                    'VerticalAlignment', 'middle');
            end
        end
    end
end

%%
% END OF 'Croos_Corr' CODE
```