Automated Classification of Pectinodon Bakkeri Teeth Images Using Machine Learning

Jacob A. Bahn
Southern Adventist University, jacobabahn@southern.edu

AUTOMATED CLASSIFICATION OF FOSSILIZED PECTINODON BAKKERI TEETH IMAGES USING MACHINE LEARNING

by

Jacob Bahn

A THESIS

Presented to the Faculty of

The School of Computing at the Southern Adventist University

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Dr. Germán Harvey Alférez Salinas

Collegedale, Tennessee

April, 2023

Automated Classification of Fossilized Pectinodon Bakkeri Teeth Images Using Machine Learning

Approved by:

Germán Harvey Alférez

Professor Germán Harvey Alférez Salinas, Adviser

School of Computing, Southern Adventist University

Keith Snyder

Professor Keith Snyder

Biology and Allied Health Department, Southern Adventist University

[illegible]

Professor Richard Halterman

School of Computing, Southern Adventist University

Date Approved 4/28/2023

AUTOMATED CLASSIFICATION OF FOSSILIZED PECTINODON BAKKERI TEETH IMAGES USING MACHINE LEARNING

Jacob Bahn, M.S.

Southern Adventist University, 2023

Adviser: Dr. Germán Harvey Alférez Salinas

Microfossil dinosaur teeth are studied by paleontologists in order to better understand dinosaurs. Currently, tooth classification is a long, manual, error-ridden process. Deep learning offers a solution that allows for an automated way of classifying images of these microfossil teeth. In this thesis, we aimed to use deep learning in order to develop an automated approach for classifying images of *Pectinodon bakkeri* teeth. The proposed model was trained using a custom topology and it classified the images based on clusters created via K-Means. The model had an accuracy of 71%, a precision of 71%, a recall of 70.5%, and an F1-score of 70.5%.

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Statement

Microfossils are paleontological remains that are more clearly visible through a microscope [4]. The manual classification of microfossils is a cumbersome task. As a result, there are recent approaches that apply deep learning for the automatic classification of images of microfossils, for example in the case of fish teeth [5]. Unfortunately, deep learning has not been applied to the automatic classification of images of dinosaur teeth.

The goal of this project is to apply deep learning for the automatic classification of images of microfossilized teeth of the *Pectinodon bakkeri* species. To this end, a convolutional neural network topology was created with TensorFlow. A sample data set with 367 images of individual teeth provided by the Biology Department at Southern Adventist University was used for training and evaluating the deep learning model. Alongside there is a numerical dataset with tooth measurements with the following variables: crown height, fore aft basal width, basal width, anterior denticles, and posterior denticles per millimeter. The image dataset does not contain information about the fore aft basal width and basal width.

In order to know the groups in which the images were going to be classified, this project proposed the application of PCA for understanding the underlying variables and clustering (unsupervised learning) on the numerical dataset. The images were organized according to these groups. Then, these images were used to train a deep learning model. A topology for such a model was built. The model was validated and tested with the following metrics: accuracy, precision, recall, and F1-score.

1.2 Goals and Requirements

These are the major goals of this project:

- Apply Principal Component Analysis and K-Means clustering (i.e., unsupervised machine learning) to a comma-separated values (CSV) file containing numerical measurements for the samples of *Pectinodon bakkeri* teeth. The data is comprised of the following numerical variables: crown height, fore aft basal width, basal width, anterior denticles, and posterior denticles per millimeter. The clusters generated with K-Means were used for determining the different groups that a deep learning model used for classifying the dataset containing 487 images of mircrofossilized teeth.

- Train and validate the deep learning model that classifies images of *Pectinodon bakkeri* teeth. The validation was carried out by means of accuracy, precision, recall, and F1-score.

1.3 Motivation

Although there are approaches for the manual classification of microfossils [6, 7], determining these characteristics from a set of teeth can be extremely time consuming and error-prone. Some of these characteristics could include crown height, basal width, anterior denticles, posterior denticles per millimeter, and more. Advances in machine learning and computer vision offer an opportunity to train deep learning models that are able to classify images microfossils. This automated opportunity would open the door to faster classification times for paleontologists.

Chapter 2

Background

The theoretical framework and state of the art introduce terms and include relevant works related to the topics of deep learning and microfossils, respectively.

2.1 Theoretical Framework

This section presents the definitions of the concepts covered in this research work. Figure 2.1 presents the underlying concept map.

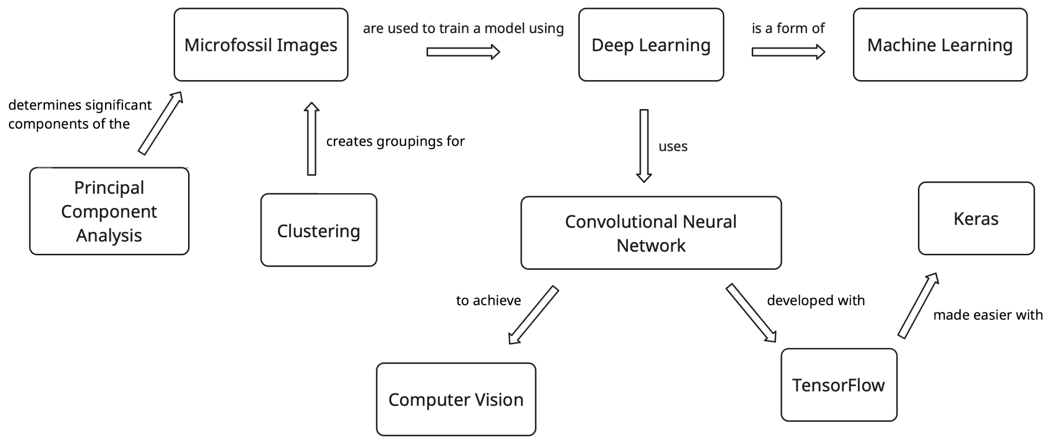


Figure 2.1: Concept map showcasing the relationship between the concepts presented in this section

2.1.1 Microfossils

Microfossils are remains of fossils, usually smaller than 1mm, that are only viewable through a microscope. They come in many forms including bacteria, plants, and animals. Microfossils are studied independently of each other due to the way that rock samples are processed and the need for microscopes in their study. Microfossils are arguably the most important type of fossil. Conveniently, they are also the most abundant type of fossil, with large numbers found in sedimentary rocks. They are useful in the study of age-dating, correlation, and reconstructing ancient environments [4]. Some of the measurements used in their classification can be seen in Figure 2.2.

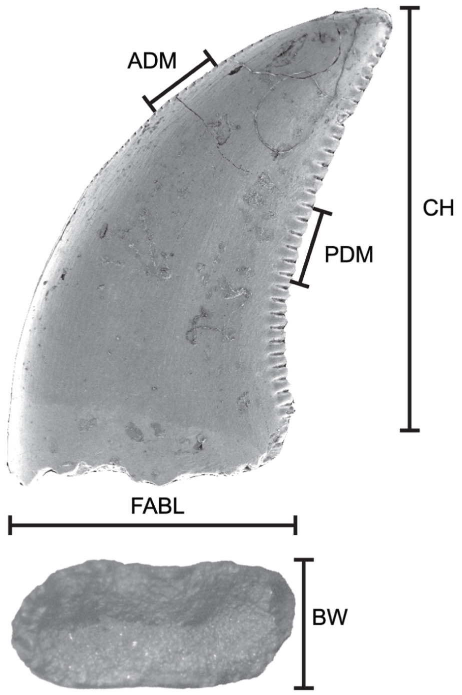


Figure 2.2: Small microfossilized theropod dinosaur tooth with measurements for the anterior denticles per millimeter, basal width, crown height, fore-aft basal length, and posterior denticles per millimeter [1].

2.1.2 Principal Component Analysis

Principal Component Analysis is an unsupervised machine learning technique used for reducing the dimensions of a data set. The algorithm accomplishes this by transforming a large set of variables into a smaller set while retaining most of the larger set's information. This results in a list of significant components that allow for easier analysis of the data set [8].

2.1.3 K-Means Clustering

Clustering refers to separating data points into groups that contain similar elements. The K-means clustering algorithm is a popular unsupervised approach for clustering data. It helps with identifying natural groupings in a data set. K-means works by iteratively assigning data points to the nearest cluster centroid, based on a specified number of clusters, until the assignments stay the same or maximum number of iterations is met [8].

2.1.4 Machine Learning

Machine learning can be defined as the process where computers learn from data without explicitly being programmed. It is especially useful for detecting patterns in images [9]. Machine learning is often split into 2 categories, supervised and unsupervised learning. Supervised learning is where the training data is labeled. Unsupervised learning is where the data is unlabeled. In unsupervised learning, the machine learning algorithms group the data in order to discover patterns [10].

2.1.5 Deep Learning

Deep learning is a method of machine learning in which computers learn from experience and understand the world through a hierarchical structure of concepts. This approach allows the computer to acquire knowledge through experience rather than requiring manual input from a human operator. The hierarchical concept structure enables the computer to understand complex ideas by breaking them down into simpler components [11].

2.1.6 Convolutional Neural Network

A convolutional neural network (CNN) is a specialized neural network that processes an input image by assigning importance to various objects and features within it, allowing it to differentiate between them. A CNN has the following elements [12]:

- Convolution Layer: This is the core of a CNN and it carries most of the network's computational load. In this layer, a dot product's output is calculated between an area of an input image and a weight matrix, also known as a filter. The filter slides through the image repeating the dot product operation. An example can be seen in Figure 2.3.

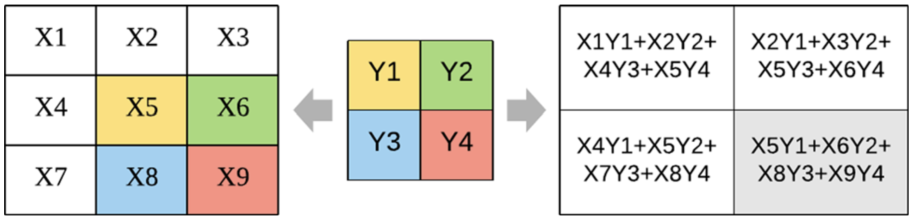


Figure 2.3: Convolution Operation [2].

- Grouping Layer: This layer is used for reduction of the spatial dimension without affecting the depth. Max pooling can be utilized in order to get highest number of the input's area, the most sensitives area. Figure 2.4 is an example of max pooling. Average pooling can be used to find the mean of the input area also.

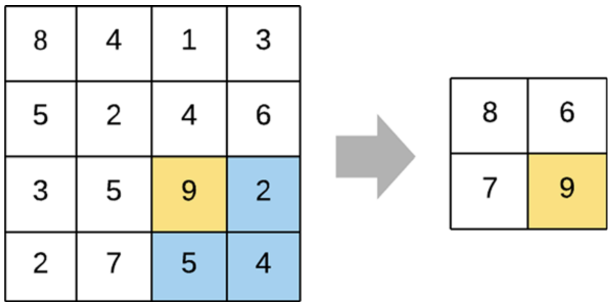


Figure 2.4: Max Pooling [2].

- Nonlinearity layer: This layer utilizes the ReLU activation function that returns zero for every negative value and original number for every positive value. See Figure 2.5.

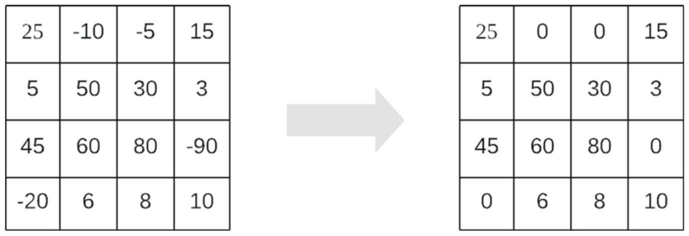


Figure 2.5: ReLU Operation [2].

- Fully Connected Layer: This layer flattens out the result of the previous convolutional layer and connects each node of the current layer to another node in the next layer.

- Dropout Layer: This layer removes the contribution of some neurons towards the subqequent layers. It is a regularization technique that prevents overfitting.
- Flatten Layer: This layer transforms the 2 dimensional arrays that result from the pooling layer into a single linear vector that can be passed into a fully connected layer.

2.1.7 Computer Vision

Computer Vision, also called artificial vision, is a field of deep learning that seeks to understand and interpret characteristics of the physical world through the analysis of images. Computer vision aims at emulating the way humans use vision by rebuilding properties from images like shape and color [13]. Computer vision can be implemented using CNN's, which enable image processing at a pixel level [14].

2.1.8 Tensor Flow

TensorFlow is an open-source library created by Google that is used for distributed numerical computation using data flow graphs. TensorFlow defines and executes calculations involving tensors, which are n-dimensional arrays of base data types [9]. Each element in a tensor has an identical data type and the type is always known. The shape, however, is not always fully known before runtime. TensorFlow's usage is generally for training deep neural networks.

2.1.9 Keras

Keras is a high-level API built on top of TensorFlow that enables faster neural network development. It accomplishes this by focusing on simplicity and flexibility. Keras' built-in layers and models provide abstractions which simplify deep learning model creation [15].

2.2 State of the Art

This section describes the current usage of deep learning for image classification, manual classification of microfossils, and automated classification of microfossils.

2.2.1 Application of Deep Learning and Computer Vision for Image Classification

This section presents examples of projects developed by the advisor of this research project for the automatic classification of images using deep learning.

In [16], the authors developed a deep-leaning model to identify plutonic rocks in order to save time and money. They utilized TensorFlow to create a convolutional neural network and trained it to identify 6 different types of rocks. They found their model to have an accuracy of 95%, a precision of 96%, a recall of 95%, and an F1 score of 95%.

Relating to the medical field, in [2], the authors created a mobile app that utilized a convolutional neural network to detect if a patient has a difficult airway. They used the MobileNetV2 computer vision model and TensorFlow to train a deep learning model based off of 240 images. The authors model had an average accuracy of 88.5%. In [17], the authors trained a deep learning model for detecting glaucoma when given a picture of a human eye. They specifically targeted the Latino population with 260 total pictures. The MobileNet and Inception V3 models were used to create the deep learning model. The best results came from the Inception V3 model, which had a value of 90% for precision, accuracy, recall, and F1 score.

2.2.2 Manual Classification of Microfossils

In [6], the authors introduce characteristics of microfossills that allow for them to be identified in clusters of fossilized bones. These clusters, also known as microsites, are often comprised of many different species. The specific microsites studied were from the Hell Creek Formation in Montana. The research project contains images that show the features of different species and the authors showcase the differences between them. Also, the authors do a deep dive on the comparative anatomy of vertebrate animals, creating a guide for assigning fossils to their respective animal groups. This guide is useful for directing the manual classification of microfossils from microsites.

In [7], the authors studied, measured, and compared features of more than 500 dinosaur teeth from 18 locations. They observed how the shape of the tooth implied predatory behavior. This led them to closely analyze relationships between characteristics of teeth from the same species and see if there were any patterns. From this, they noticed a correlation between serration size and tooth size in dinosaurs and predatory vertebrates.

In [1], the authors carefully studied and assessed a dataset of 1,183 dinosaur teeth using multivariate statistical methods. The teeth, obtained in North America, were found to have qualities dependent on the formation they came from. So, teeth found in time equivalent formations were indistinguishable and distinct compared to teeth from other formations. In the study, 5 different measurements were utilized in order to aid in manually classifying the teeth.

2.2.3 Automated Classification of Microfossils

In [5], the authors detail their approach for training two open source deep learning models for tooth classifications. The first model is called "Mask R-CNN" and it was used for object detection. The second model is the "EfficientNet-V2" and it was used for classifying the teeth. Overall, their model achieved 89.0% precision and 78.6% recall. Our approach is different from this because we are working with images of dinosaur teeth instead of fish teeth and we will train and evaluate our own deep neural network topology.

In [18], the authors used deep learning to classify microfossil species using sediments obtained from the Southern Ocean and Japan Sea. The model is limited to classifying 2 sediments, but the authors were able to achieve an accuracy greater than 90% and a confidence level of 0.90. This paper's results show that machine learning has positive results for usage with images of microfossils. It is different from ours because it is specifically classifying underwater sediments.

In [19], the authors use machine learning to classify Cenozoic radiolarians. The authors utilized the MobileNet convolutional neural network for the microfossil classification. The model resulted in average accuracy of 73%. In [20], the authors explain their process of training a deep learning model to classify microfossil species. Both of these studies vary from our approach due to the type of microfossil studied and the choice of convolutional neural network.

Chapter 3

Methodology

The aim of this research work is to use deep learning to develop an automated approach for classifying images of microfossil dinosaur teeth of the *Pectinodon bakkeri* species. We accomplished it by creating a topology for a deep learning neural network that is trained on the provided microfossil images from the Biology Department of Southern Adventist University. The technology utilized for building it was Keras.

The IBM Foundational Methodology for Data Science [3] was used in this research project because it is a proven methodology for working on data science projects (see Figure 3.1). The first step of this methodology requires a comprehensive understanding of the problem. This was followed by looking at the techniques and technologies utilized for the solution. The next step was to focus on data, including identifying the type of data required, the methods for collecting it, techniques for gaining deeper insights, and preparing it for analysis. Lastly, it covered the deployment and evaluation steps. These steps are explained in the proceeding subsections.

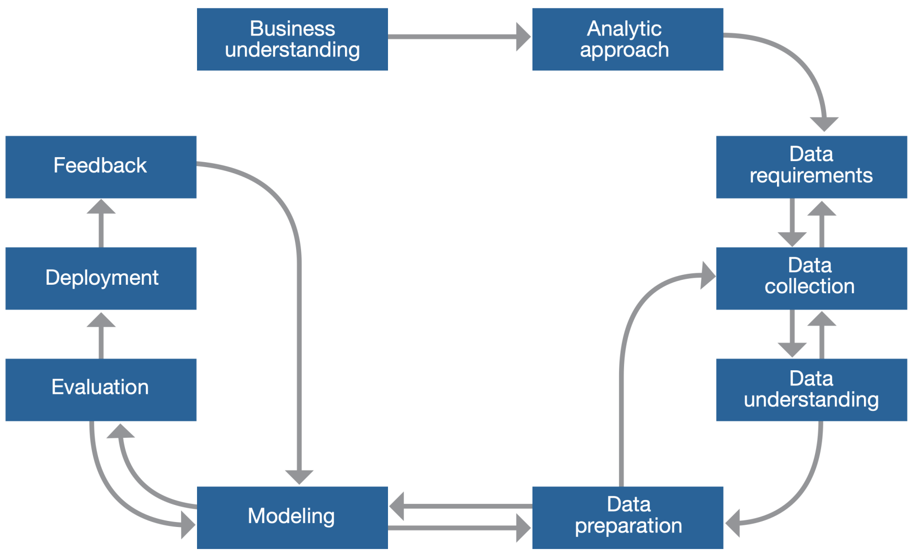


Figure 3.1: IBM foundational Data Science Methodology [3].

3.1 Problem Understanding

Paleontologists have done work studying microfossil teeth [6] and learning from measured data, but current approaches to classify dinosaur teeth are time-consuming. As deep learning has grown, an opportunity has arisen to incorporate computer vision into classifying these teeth. Microfossil *Pectinodon bakkeri* teeth numerical data and images have been provided by the Biology department of Southern Adventist University to aid this research.

3.1.1 Analytic Approach

R was used to create a PCA and K-Means models from the numerical dataset containing measurement values of the microfossil teeth. The PCA model let us understand the underlying variables of the collected microfossils. The K-Means model let us organize the samples into clusters. Deep Learning was used to develop a classification model of the *Pectinodon bakkeri* tooth images according to the clusters found with K-Means. Specifically, Python was used with TensorFlow and Keras to create a convolutional neural network.

3.1.2 Data Requirements

For the data set given, it was necessary to have samples with measurements calculated to properly apply the PCA and clustering analysis. To this end, we were provided with a numerical dataset of 484 *Pectinodon bakkeri* tooth samples with the the following variables. There were data samples without complete numerical measurements, which were removed from the analysis. The dataset is stored in a CSV file, which is available online[1]:

- Crown Height: The total height of the tooth.
- Fore Aft Basal Length: The length of the base of the tooth, from the front to the back.
- Basal Width: The width of the tooth's base.
- Posterior Denticles Per Millimeter: The number of small, pointed structures called denticles located on the posterior or back part of a tooth, within a millimeter of tooth length.
- Anterior Denticles: Presence or absence of anterior denticles.

Figure 3.2 shows a sample of one of the images of the samples that are used in this study. The images are in JPG format. The numerical dataset and image dataset can be found online also at[1].

[1] https://github.com/jacobabahn/MicrofossilResearch

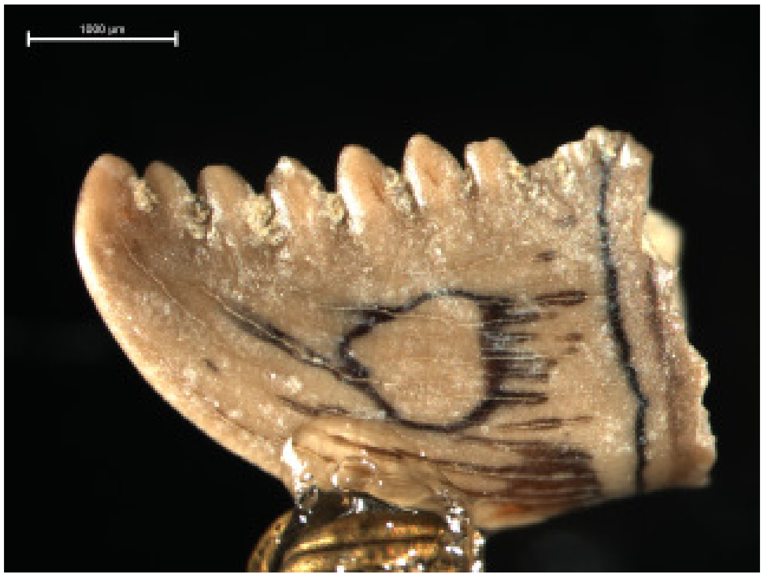


Figure 3.2: Microfossil Dinosaur Tooth Image

3.1.3 Data Collection

The data was provided by the Biology Department at Southern Adventist University using the Dino-Lite Edge 3.0 digital microscope, see Figure 3.3. The original images were cropped in order to remove extra information, such as the holding clip. Figure 3.4 shows an image in its original form and after it was cropped.

3.1.4 Data Understanding

To enhance our comprehension of the data, we employed PCA analysis, which aided in identifying the critical variables within the dataset. To ensure data consistency, only samples with values for every measurement were included in the analysis. The Elbow Method was used in order to find the number of clusters to separate the data set into. K-means was used to determine which samples belonged to each cluster.

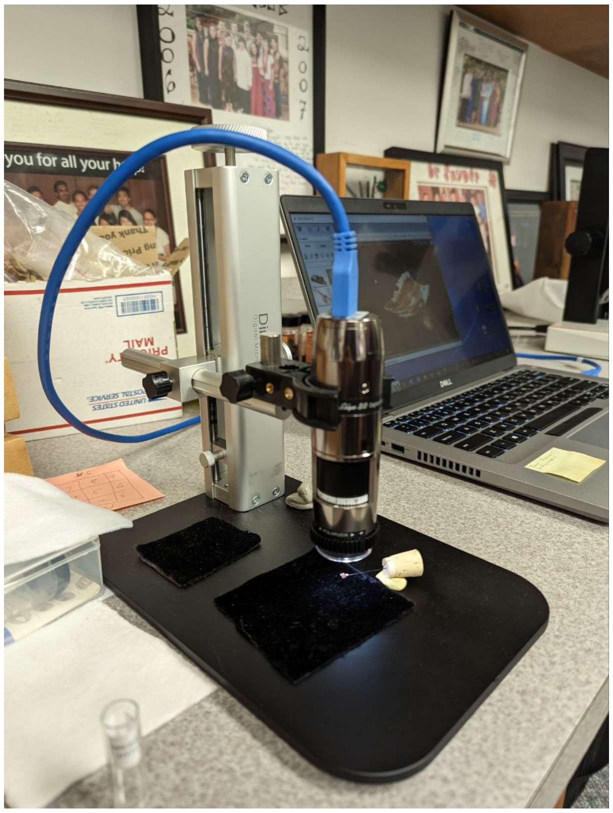


Figure 3.3: Dino-Lite Edge 3.0 Digital Microscope

3.1.5 Data Preparation and Modeling

For the most part, the data provided was usable. However, there were 15 samples that had missing values for some of the measurements. Therefore, these samples were not included in the experiments. In total, 459 samples were used in this research.

From the image dataset, there were 367 usable samples, but the clusters were unbalanced. In order to balance them for the deep learning experiments, image augmentation was incorporated using Python. Figure 3.5 and Figure 3.6 provide examples of images before and after augmentation, which in this case were rotation and horizontal flips.

Listing 5.19 shows the code used to add augmented images. Lines 1-4 show

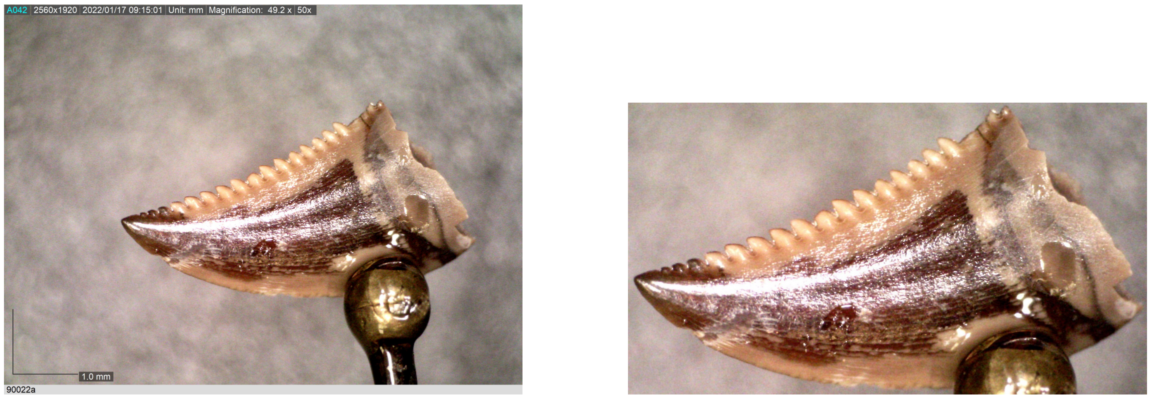

A042 2560x1920 2022/01/17 09:15:01 Unit: mm Magnification: 49.2 x 50x
1.0 mm
90022a

Figure 3.4: Microfossil Tooth Image Before Crop (Left) and After Crop (Right)

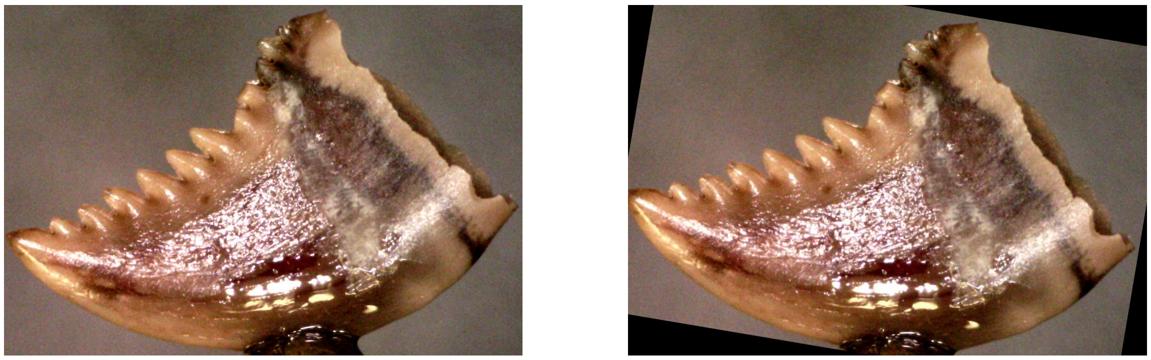


Figure 3.5: The Original Tooth (Left) and Augmented Tooth (Right)

the necessary imports for the program. On line 6, the directory containing images is stored in a variable. On line 7, the directory where the augmented images will be placed is stored in a variable. On lines 9-12, the augmentations are defined using the *imgaug* library. The augmentation on line 10 sets a random rotation between 10% and -10%. On line 11, a horizontal flip is defined for 50% of the images. On line 14, a list of image files is received and stored in the *image_files* variable. On line 15, the *count* variable is created to ensure that there will not be too many images generated, making the clusters imbalanced. On line 16, the conditional breaks when the correct amount of images to even out the clusters is met. In this case, that is 71 images. On line 19, a try-catch block is created to

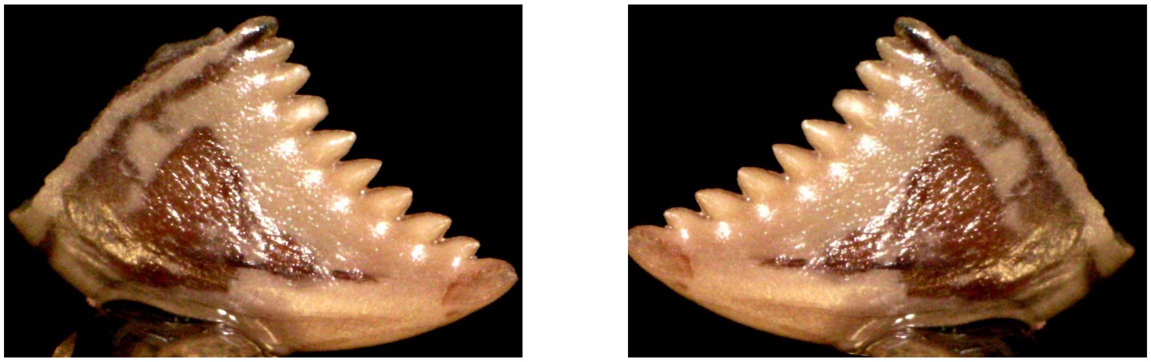


Figure 3.6: The Original Tooth (Left) and Augmented Tooth (Right)

handle line 20 where the program attempts to open an image. On line 24, the image is converted to a *numpy* array. On line 25, the image is augmented using the predefined augmentation. On line 27, the image array is converted back into an image. On line 29, the filename is set with the *'_augmented'* postfix. On line 30, the augmented image is saved to the predefined output directory. On line 32, the count is incremented so that the conditional at the beginning of the loop can check the number of images augmented.

```
1  import os
2  from PIL import Image
3  import numpy as np
4  import imgaug.augmenters as iaa
5  
6  input_dir = './path-to-images'
7  output_dir = './path-to-output-dir'
8  
9  augmentations = iaa.Sequential([
10     iaa.Affine(rotate=(-10, 10)), # rotate by -10 to 10 degrees
11     iaa.Fliplr(0.5), # horizontally flip 50% of the images
12 ])
13 
```

```python
image_files = os.listdir(input_dir)
count = 0
for file_name in image_files:
    if count == 71:
        break
    try:
        image = Image.open(os.path.join(input_dir, file_name)).convert('RGB')
    except:
        continue

    image_array = np.array(image)
    augmented_image_array = augmentations(image=image_array)

    augmented_image = Image.fromarray(np.uint8(augmented_image_array))

    output_file_name = os.path.splitext(file_name)[0] + '_augmented.jpg'
    augmented_image.save(os.path.join(output_dir, output_file_name))

    count += 1
```

Listing 3.1: Image Augmentation Script

3.1.6 Evaluation

The deep learning model was evaluated using the metrics of accuracy, precision, recall, and F1-score.

3.1.7 Deployment

A Python script was created to execute the classification on the new server of the School of Computing using the previously trained classification model. The server's specifications are as follows: two AMD Rome EPYC 7F32 8C/16T 3.7GB 128M CPUs, two NVIDIA PNYQuadroTRXA4000 16GB GPUs, 512 GB RDIMM, and two SSDs of 1.9T.

Chapter 4

Evaluation Plan

The deep learning model was evaluated using accuracy, precision, recall, and F1-score. The equations use the following metrics: Correct Positives (CP), Correct Negatives (CN), False Positives (FP), and False Negatives (FN).

- Accuracy: a measure of the total number of correct predictions. It is calculated by dividing the number of correct predictions from the total number of predictions.

$$\text{Accuracy} = \frac{CP+CN}{CP+FP+CN+FN}$$

- Precision: a measure of how many positive predictions are right. It is calculated by dividing the number of correct positive predictions by the total number of positive predictions made.

$$\text{Precision} = \frac{CP}{FP+CP}$$

- Recall: a measure of how many positive predictions were found out of the total number of positive results. It is calculated by dividing the number of positives found by the total amount of positives.

$$\text{Recall} = \frac{CP}{CP+FN}$$

- F1-Score: a measure that combines precision and recall.

$$\text{F1-Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

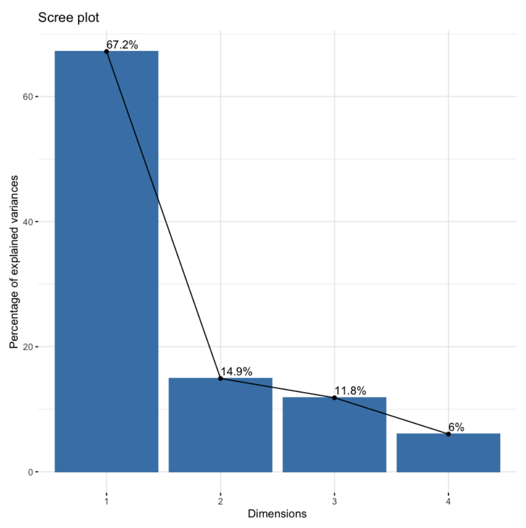
Chapter 5

Results

Initially, the PCA analysis and K-Means clustering were conducted on the numerical dataset containing the following features: crown height, fore aft basal width, basal width, anterior denticles, and posterior denticles per millimeter. The discoveries obtained in this step were used to automatically organize the images according to the clusters generated. Finally, the tooth images were used to train a deep learning model.

5.1 PCA and K-Means Clustering

According to the results from PCA, a majority of the variance (82.1%) in the data is found in dimensions 1 and 2 (left-hand side of 5.1). Looking at the correlation plot (right-hand side of 5.1), it is possible to see the presence of every feature per dimension. In dimension 1, the most relevant feature is crown height. In dimension 2, the main features represented are Posterior Denticles Per Millimeter and Fore Aft Basal Length. The plots of Figure 5.2 and Figure 5.3 show different representations of the contributions of features per dimension.



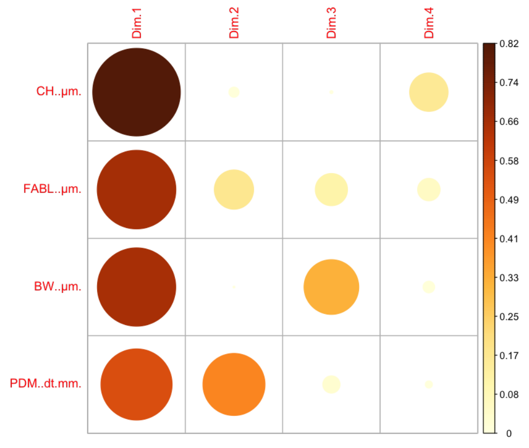
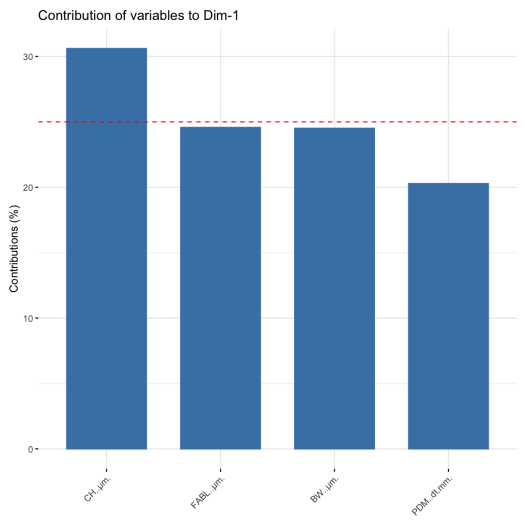


Figure 5.1: Scree Plot of Dimensional variance (Left) and Correlation Plot of Feature Contribution to each dimension (Right)



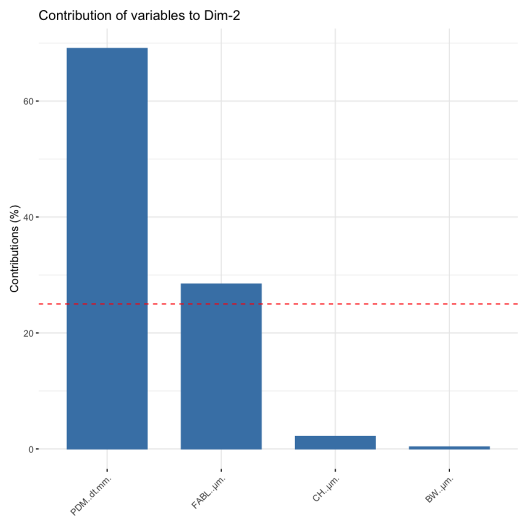


Figure 5.2: Feature Contributions for Dimension 1 (left) and Dimension 2 (Right)

Next, the Elbow Method and a clustered silhouette plot were used for determining the proper number of clusters for splitting up the data. Per the results from those algorithms (see Figure 5.4), the data was organized into 3 clusters.

Performing the K-Means clustering with 3 clusters on the modified data set rendered clusters with 177 samples for Cluster 1, 76 samples for Cluster 2, and

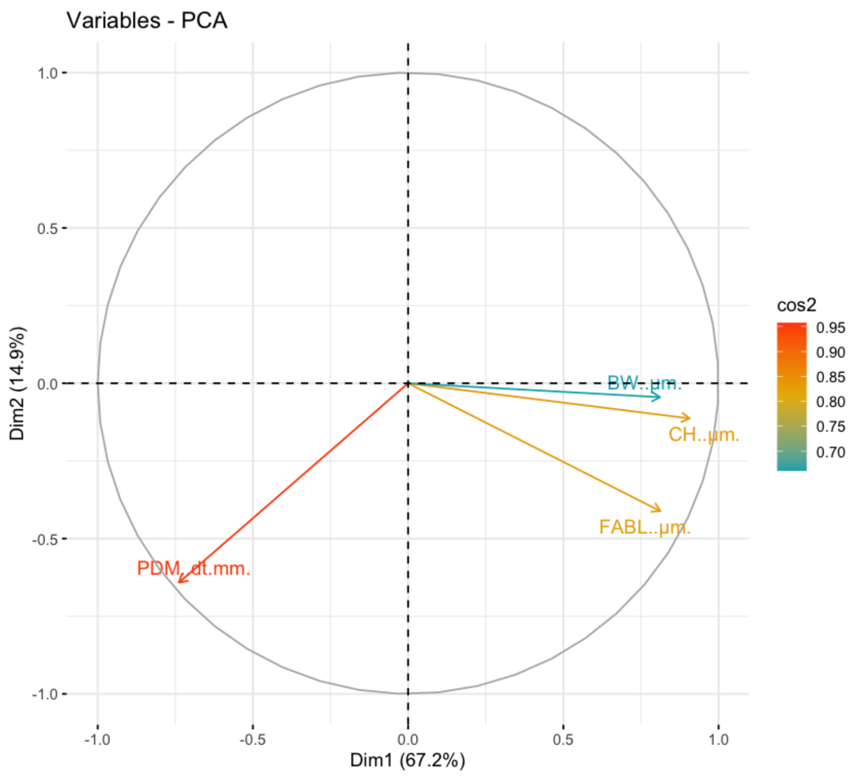


Figure 5.3: Variable Contribution plot for Dimension 1 vs Dimension 2

206 samples for Cluster 3 (see Figure 5.5). In Table 5.1, the unnormalized feature averages per cluster are shown. Cluster 1 contains the largest teeth. Cluster 2 has the smallest teeth, which have the most posterior denticles per millimeter. Cluster 3 has teeth that are bigger than the ones in cluster 2 but smaller than the ones in cluster 1.

5.1.1 Code for PCA and K-Means

The code for this section is available on GitHub[1]. In Listing 5.1 the code modifies the CSV data set with numerical tooth data to remove non-numerical rows and

[1] https://github.com/jacobabahn/MicrofossilResearch

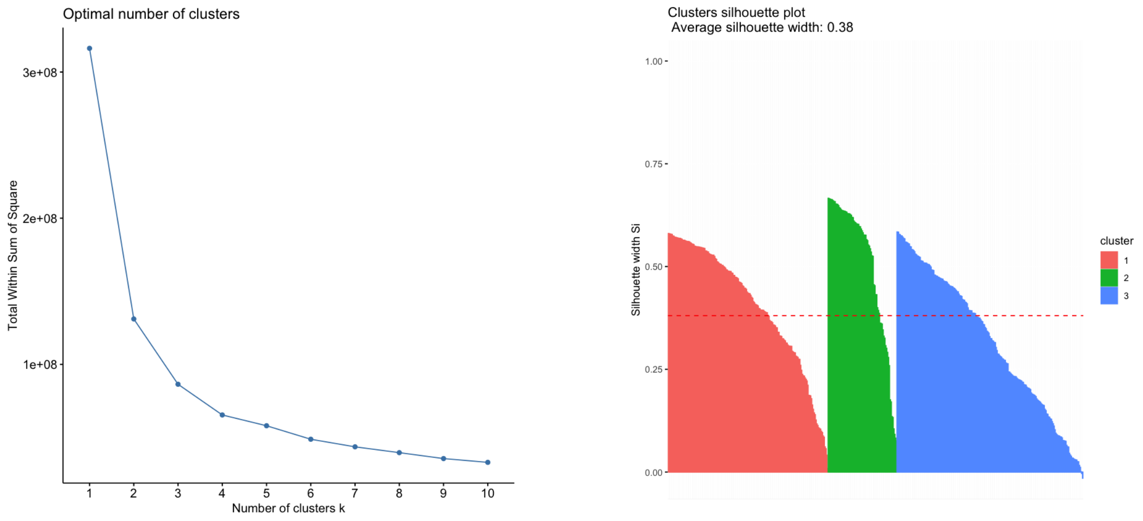


Figure 5.4: Elbow Plot and Silhouette Plot

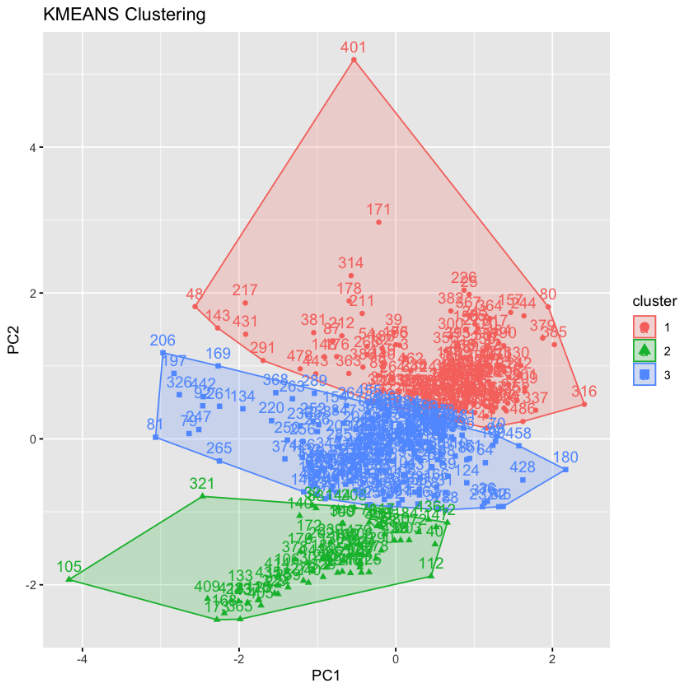


Figure 5.5: K-Means Clustering result with 3 Clusters

incomplete fields. On line 1, the first 2 rows that contain descriptive information and an empty row were removed from the data. On line 2, the names for all of the samples are grabbed and stored. On line 3, the columns with string values were

Cluster	Crown Height	Fore Aft Basal Length	Basal Width	Posterior Denticles Per Millimeter
1	4,495.408	2,773.842	1,192.1974	1.892105
2	2,854.301	2,160.572	7,85.4046	3.689133
3	3,671.417	2,541.441	941.2941	2.967946

Table 5.1: Feature Averages Per Cluster (Unnormalized)

removed from the data. On line 4, the rows are converted to the numeric type, so that they can be processed later. On line 5, every row with missing fields are removed from the data.

```
1  data <- data[-c(1, 2, 356),]
2  names <- data[, 1]
3  modded_data <- data %>% dplyr::select(-1, -6, -7)
4  modded_data[, c(1:4)] <- sapply(modded_data[, c(1:4)], as.
  numeric)
5  modded_data <- modded_data[complete.cases(modded_data), ]
```

Listing 5.1: Data Modification

Listing 5.2 shows the code that performs PCA. The *prcomp* function on line 1 performs PCA on the modified data set and scales the result. On the next line, the *get_pca_var()* function is called which extracts information from each variable in the PCA object.

```
1  pca <- prcomp(modded_data, scale = TRUE)
2  var <- get_pca_var(pca)
```

Listing 5.2: PCA

Listing 5.3 shows how the plots in 5.1 are created. The *fviz_eig* function creates a scree plot from the *pca* object. The scree plot showcases a visualization of the variance in each of the principal components. On line 2, the *corrplot* function is called to create a correlation plot of the squared cosine values in each of the PCA variables. The cos2 values are the amount that each variable is represented in the PCA for each of the dimensions. The amount is visualized with different colors and sizes in the correlation plot. The graphs generated from this code can be seen in Figure 5.1. On lines 4-6, the *fviz_pca_var* function is called. This function generates a scatter plot of the PCA variables where each are colored by the representation of *cos2*.

```
1    fviz_eig(pca, addlabels = TRUE)
2    corrplot(var$cos2, is.corr = FALSE)
3
4    fviz_pca_var(pca, col.var = "cos2",
5                 gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
6                 repel = TRUE,
7                 )
```

Listing 5.3: Scree and Correlation Plots

In Listing 5.4, the K-Means algorithm is executed and the clusters are determined. On line 1, the seed is set in order to ensure that results will be reproducible each time the code is executed. On line 2, PCA is run again this time making sure that data centering is set to false and that a rank of 2 is set. The reason for running PCA again was to set the rank to 2, which tells the *prcomp* function to compute only the first 2 principal components, which contain a majority of the data's variation. On line 5, a results variable is created which contains the principal component values. The *fviz_nbclust* function is used to run the Elbow Method algorithm which determines the optimal number of clusters for a dataset. It plots within-cluster sum of squares (*wss*) vs number of clusters. On line 8, the optimal number of clusters selected by the *fviz_nbclust* function is returned. On line 10, the *eclust* function is called for performing K-Means clustering on the PCA results. Euclidean distance is used as the distance metric, and the number of clusters is set to 3. On line 12, the *fviz_silhoutte* function is used to assess the quality of a number of clusters for a K-Means output. In the graph, if one of the clusters is below the red dotted line (see Figure 5.1), it is not an effective cluster.

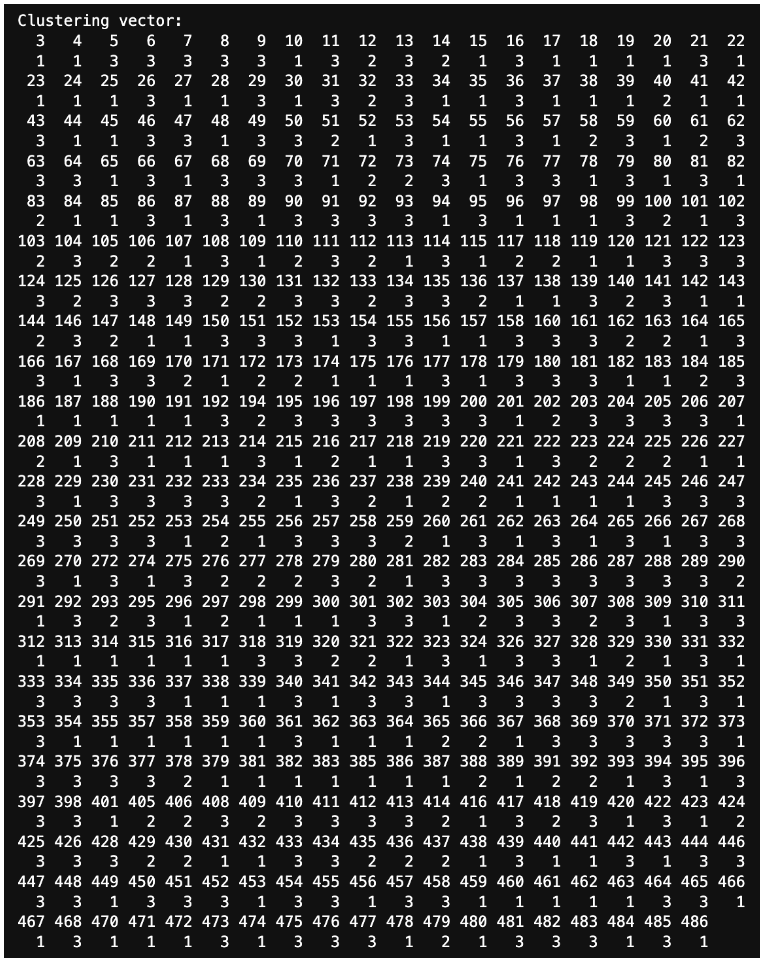
```r
1  set.seed(123)
2  
3  pca1 <- prcomp(modded_data, center=FALSE, scale=TRUE, rank. = 2)
4  
5  results <- pca1$x
6  
7  fviz_nbclust(modded_data, kmeans, method = "wss")
8  km1$nbclust
9  
10 km1<-eclust(results, "kmeans", hc_metric="eucliden", k=3)
11 
12 fviz_silhouette(km1)
```

Listing 5.4: K-Means Clustering

5.2 Python Script for the Automatic Organization of Images

After performing clustering on the numerical data set, we had a list of rows from the CSV file and the cluster that the row corresponded too. The odd rows were a sequence of samples, and the following row contained the cluster that each sample belonged to (see Figure 5.6). For example, the first value in the first row, 3, belongs to cluster 1, which is the value in the proceeding row, located in the same column.

The next step after this was grouping the images based on that output, but this was going to be a long manual task.



```
Clustering vector:
  3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21  22 
  1   1   3   3   3   3   3   1   3   2   3   2   1   3   1   1   1   1   3   1 
 23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 
  1   1   1   3   1   1   3   1   3   2   3   1   1   3   1   1   1   2   1   1 
 43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 
  3   1   1   3   3   1   3   3   2   1   3   1   1   3   1   2   3   1   2   3 
 63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 
  3   3   1   3   1   3   3   3   1   2   2   3   1   3   3   1   3   1   3   1 
 83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 
  2   1   1   3   1   3   1   3   3   3   3   1   3   1   1   1   3   2   1   3 
103 104 105 106 107 108 109 110 111 112 113 114 115 117 118 119 120 121 122 123 
  2   3   2   2   1   3   1   2   3   2   1   3   1   2   2   1   1   3   3   3 
124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 
  3   2   3   3   3   2   2   3   3   2   3   3   2   1   1   3   2   3   1   1 
144 146 147 148 149 150 151 152 153 154 155 156 157 158 160 161 162 163 164 165 
  2   3   2   1   1   3   3   3   1   3   3   1   1   3   3   3   2   2   1   3 
166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 
  3   1   3   3   2   1   2   2   1   1   1   3   1   3   3   3   1   1   2   3 
186 187 188 190 191 192 194 195 196 197 198 199 200 201 202 203 204 205 206 207 
  1   1   1   1   1   3   2   3   3   3   3   3   3   1   2   3   3   3   3   1 
208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 
  2   1   3   1   1   1   3   1   2   1   1   3   3   1   3   2   2   2   1   1 
228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 
  3   1   3   3   3   3   2   1   3   2   1   2   2   1   1   1   1   3   3   3 
249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 
  3   3   3   3   1   2   1   3   3   3   2   1   3   1   3   1   3   1   3   3 
269 270 272 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 
  3   1   3   1   3   2   2   2   3   2   1   3   3   3   3   3   3   3   3   2 
291 292 293 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 
  1   3   2   3   1   2   1   1   1   3   3   1   2   3   3   2   3   1   3   3 
312 313 314 315 316 317 318 319 320 321 322 323 324 326 327 328 329 330 331 332 
  1   1   1   1   1   1   3   3   2   2   1   3   1   3   3   1   2   1   3   1 
333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 
  3   3   3   3   1   1   1   3   1   3   3   1   3   3   3   3   2   1   3   1 
353 354 355 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 
  3   1   1   1   1   1   1   3   1   1   1   2   2   1   3   3   3   3   3   1 
374 375 376 377 378 379 381 382 383 385 386 387 388 389 391 392 393 394 395 396 
  3   3   3   3   2   1   1   1   1   1   1   1   2   1   2   2   1   3   1   3 
397 398 401 405 406 408 409 410 411 412 413 414 416 417 418 419 420 422 423 424 
  3   3   1   2   2   3   2   3   3   3   3   2   1   3   2   3   1   3   1   2 
425 426 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 446 
  3   3   3   2   2   1   1   3   3   2   2   2   1   3   1   1   3   1   3   3 
447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 
  3   3   1   3   3   3   1   3   3   1   3   3   1   1   1   1   1   3   3   1 
467 468 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 
  1   3   1   1   1   3   1   3   3   3   1   2   1   3   3   3   1   3   1 
```

Figure 5.6: K-Means Clustering Output

In order to organize the images according to the clusters, a Python program was created. The source code presented in this section is available online[2]. The first part of the program matched matched samples with the cluster that they belonged to. Following that, the program would take a directory containing image samples and group all of them in the folder according to the clusters created in the previous

[2]https://github.com/jacobabahn/MicrofossilResearch

step. The final result was 3 folders corresponding to the 3 clusters with the image samples that belong to them (see Figure 5.7). These images are available online[2].

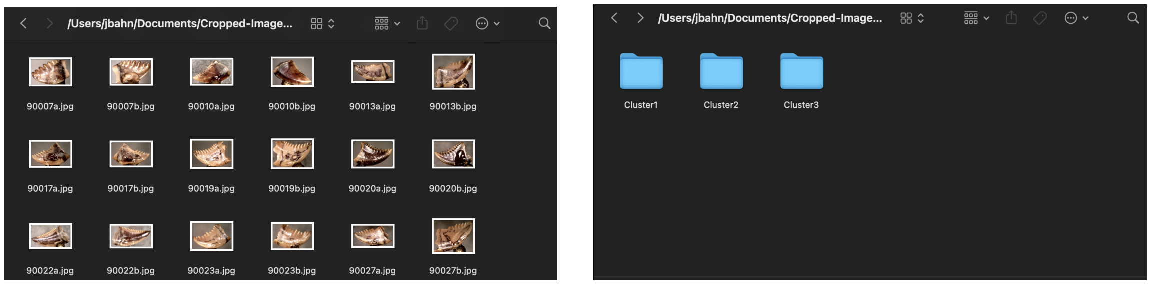


Figure 5.7: Directory Before Running the Clustering Program (Left) and After Running the Program (Right)

Listing 5.5 shows the function that puts samples in the correct clusters based on the K-Means output. Lines 4-11 show the transformation from the K-Means text file to a 2D Array. On line 14, the clusters array is created with each index being one of the clusters. Lines 15-18 are where each sample is put into the correct cluster array. Due to how the K-Means output was organized, the even rows corresponded to the samples and the odd rows corresponded to which cluster a sample belonged to. The samples were matched with the cluster that was in the same column, where samples in row n were matched with the cluster given in $n + 1$.

```
1 def put_samples_in_clusters(modded, sampleList):
2     new = []
3 
4     for line in file:
5         if line == "":
6             continue
7         line = line.split(" ")
```

```
        for value in line:
            if value == "":
                line.remove(value)
        new.append(line)

    clusters = [[], [], []]

    for i in range(len(new)):
        for j in range(len(new[i])):
            if i % 2 == 0:
                clusters[int(new[i + 1][j]) - 1].append(sampleList[int(new[i][j]) - 1])

    return clusters
```

Listing 5.5: put_samples_in_clusters() Function

Listing 5.6 moves on to the next function in the program which moves image files from a source directory to specific cluster directories. The function accomplishes this by using the clusters array generated in the *put_samples_in_clusters()* function. In lines 2-7, the directory containing tooth images is set as well as the output directories for the clusters. Line 9 is where the contents of the directory are retrieved. In lines 12-26, each image is moved into the cluster directories. Specifically, the if condition is used to strip the file name into a string that can be correctly used as an input for the *get_cluster()* function. For example, a file name could be 95000a.jpeg or 95000b.jpeg for a specific sample, 95000, where the a and b stand for the side of the tooth that the image was taken on. The *if* condition would strip it to be 95000 so that it can be properly used as an input for the *get_cluster()* function. After this, on line 20, the *get_cluster()* function is called giving the sample name and the clusters array. Lines 22-26 check that a cluster was found for the given sample and then the current image file is moved to the correct cluster folder.

```
def put_image_in_cluster_folder(clusters):
    directory = './path-to-directory'
    clusterPaths = [
            './path-to-directory/cluster1',
            './path-to-directory/cluster2',
            './path-to-directory/cluster3'
        ]

    files = os.listdir(directory)

    sample = ''
    for file in files:
        if "a" in file:
            sample = file.split("a")[0]
        elif "b" in file:
            sample = file.split("b")[0]
        else:
            print("Not a correct file name: " + file)
            continue

        cluster = getCluster(sample, clusters)
        if cluster == None:
            print("Cluster not found for " + sample)
            continue
        else:
            shutil.move(directory + "/" + file, clusterPaths[
cluster])
```

Listing 5.6: put_image_in_cluster_folder Function

Listing 5.7 shows the *get_cluster()* function. It is a helper function that returns the cluster a sample belongs to. The first input is a sample name and the second input is a 2D array where each of the inner arrays correspond to a cluster. In line 2, a for loop is created for checking each of the clusters. In line 3, a list comprehension is used to check for the presence of the sample in the cluster. If the sample is in the cluster it is added to the *res* variable. In lines 4 and 5, the program checks if a value was added to *res*, and if there was, the cluster number is returned from the function.

```
1 def get_cluster(sample, clusters):
2     for i in range(len(clusters)):
3         res = [ele for ele in clusters[i] if(sample in ele)]
4         if len(res) != 0:
5             return i
```

Listing 5.7: get_cluster() Function

There were a few more helper functions in the program that will only be mentioned briefly. There is a *print_clusters()* function that will print all of the sample names for each cluster. There is a *list_cluster()* function that allows a user to enter a sample name and the program will output the cluster that the sample belongs to. There is also an error checking method that looks at the contents of a cluster directory and determines if all of the images in that directory belong to it. Overall, the program was utilized to easily group the sample images into the proper clusters determined from the K-Means algorithm in order to be used in the training of the deep learning model.

5.3 Deep Learning Topology

This section presents the deep learning topology that was used to train a model to classify images of *Pectinodon bakkeri* teeth. Table 5.2 summarizes this topology. This topology has eight layers. The first layer applies a convolution to the images. It has an output shape of (None, 180, 180, 16), where the *None* represents the batch size, *180, 180* represent the number of pixels in the input, and 16 is the filter count. 448 parameters are trained in this layer. The next layer applies max pooling to the output of the previous layer. It has an output shape of (None, 90, 90, 16) and parameter count of 0. The following layer performs a dropout on the previous layer and also has an output shape of (None, 90, 90, 16), with a parameter count of 0. The next layer is another convolution layer and it has an output shape of (None, 90, 90, 32) and a parameter count of 2,080. The following layer is a max pooling layer which has an output shape of (None, 45, 45, 32) and a parameter count of 0. The layer after this is a flatten layer, which has an output shape of (None, 64,800) and a parameter count of 0. This layer takes all of the pixels from the previous layer's output and combines them into a one dimensional vector. The next layer performs a dropout, and the output shape is (None, 64) with a parameter count of 4,147,264. The final layer is a dense layer which has an output shape of (None, 2) and a parameter count of 130.

The source code of the topology is available on GitHub[3]. This source code is described as follows. First, Listing 5.8 shows the program's imports.

```python
import sys
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
```

[3]https://github.com/jacobabahn/MicrofossilResearch

Layer (type)	Output Shape	Param #
conv2D (Conv2D)	(None, 180, 180, 16)	448
max_pooling2D (MaxPooling2D)	(None, 90, 90, 16)	0
dropout (Dropout)	(None, 90, 90, 16)	0
conv2D_1 (Conv2D)	(None, 90, 90, 32)	2080
max_pooling2D_1 (MaxPooling2D)	(None, 45, 45, 32)	0
flatten (Flatten)	(None, 64800)	0
dropout (Dropout)	(None, 64)	4147264
dense_1 (Dense)	(None, 2)	130

Table 5.2: The Custom Deep Learning Topology

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import Dropout, Flatten, Dense
from tensorflow.keras import layers
from tensorflow.keras import backend as K
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt
from matplotlib import pyplot
from livelossplot.tf_keras import PlotLossesCallback
import numpy as np
from sklearn.metrics import classification_report
```

Listing 5.8: The Deep Learning Model's Imports

Listing 5.9 shows the global variable setup. On line 1, the path to the directory of training images is stored. On line 2, the path to the directory of validation images is stored. On line 4, the number of epochs for training the model is set. The amount of epochs determines the number of times the entire dataset is used to train a model. So, in this case, the entire dataset will be used to train the model 100 times. On line 5, the batch size is set to 32. Batch size specifies the number of samples in each batch of data that is processed by the model during training.

```
trainingData = "./path-to-directory"
validationData = "./path-to-directory"

epochs = 100
batch_size = 32
```

Listing 5.9: Global Variable Setup

Listing 5.10 enables distributed computing using the two GPUs available in the server. On line 1, a *HierarchicalCopyAllReduce* object is created. These objects are used for optimizing communication between devices when training with a distributed approach. On line 3, a *MirroredStrategy* object is created to perform synchronous data parallelism by replicating the model across both GPUs. This is another optimization strategy. On line 7, the number of devices is outputted to the console to verify that the program is correctly identifying both GPUs.

```
1 cross_device_ops = tf.distribute.HierarchicalCopyAllReduce()
2 #,"/gpu:1","/gpu:0"]
3 strategy = tf.distribute.MirroredStrategy(
4     ["device:GPU:%d" % i for i in range(2)],
5     cross_device_ops=cross_device_ops)
6 # outputs 2
7 print('Number of devices: {}'.format(strategy.num_replicas_in_sync))
```

Listing 5.10: Distributed Computing Setup

Listing 5.11 shows the code where the custom topology is implemented. On line 1, the function for creating the model is defined. On line 2, a Sequential model is created. The Sequential model is used for creating deep learning models that consist of a linear stack of layers. On line 3, a convolutional layer that performs 2D convolutions was added to the model. The first argument, 16, determines the number of filters in the layer. The second argument, (3, 3), specifies the filter's size. In this case, it is a 3 by 3 filter. Setting the padding argument to same makes sure that the input and output have the same spatial dimensions. The *input_shape* argument determines the shape of the input data. The input is a 3D tensor set to a size of (180, 180, 3). The activation argument is set to Rectified Linear Unit (RELU). This argument determines the activation function that is applied to the output from the layer. On line 4, a Max Pooling layer is added to perform 2D max pooling on the input. Max pooling is recommended to reduce the complexity of the data set without removing essential information. This helps the model's training to remain accurate while taking less time. The *pool_size* argument, (2, 2), specifies the size of the pooling window. In this case, it is a 2 by 2 window. On line 5, a Dropout layer is added to the model. A dropout layer randomly drops out a certain percentage of units in the layer. This helps to regularize the layer and prevents overfitting. The argument passed in to the layer, 0.2, specifies the percentage of units to drop out. In this case, it is 20%. On line 6, another convolutional layer is added to the model, with 32 filters this time instead of 16. On line 7, another Max Pooling layer is added to the model. On line 8, a Flatten layer is added which transforms the data into a 1D tensor from a 3D tensor. On line 9, a Dense layer is added to the model. This is a fully connected layer. The first argument, 64, determines the number of units in the layer. The second argument determines the activation function to apply to the output. On line 10, another Dropout layer is added. This time, 50% of the units will be dropped out. On line 11, another Dense layer is added. This time, the layer has 2 units and the softmax function is applied to the output. Softmax maps the output of the model to a probability distribution over the possible classes, ensuring that the predicted probabilities sum to 1. This makes it so that the output of the model can be interpreted as a probability of the input belonging to each class.

```python
def create_model():
    model = Sequential()
    model.add(Conv2D(16, (3, 3), padding ="same", input_shape = (180, 180, 3), activation = "relu"))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(32, (2, 2), padding = "same", activation = "relu"))
    model.add(MaxPooling2D(pool_size = (2, 2)))
    model.add(Flatten())
    model.add(Dense(64, activation = "relu"))
    model.add(Dropout(0.5))
    model.add(Dense(2, activation = "softmax"))

    return model
```

Listing 5.11: Custom Topology Implementation

Listing 5.12 shows how the model is compiled. On line 1, the *create_model()* function created previously is called and assigned to the *model* variable. On line 2, the model's parameters for compilation are set with arguments for loss, optimizer, and metrics. The initial experiments were carried out with the information of three clusters. However, the results obtained exhibited an accuracy of 48%, a precision of 27%, a recall of 33%, and an F1-score of 25%, with Cluster 2 having very low scores. According to these results, it was necessary to use the information from the two stronger clusters which contained the majority of the images. Due to using only 2 clusters, i.e., the classes in this case, the binary crossentropy loss function was used. The optimizer chosen was Adam with an initial learning rate of 0.001 and epsilon of 1e-08 for preventing division by 0. Adam is a popular optimizer that dynamically adapts the learning rate during training. The metric chosen to be evaluated during training was accuracy. Setting this parameter allows the model's training and validation accuracy to be viewed as it is trained.

```
model = create_model()
model.compile(
    loss = 'binary_crossentropy',
    optimizer = optimizers.Adam(epsilon = 1e-08, learning_rate = 0.001),
    metrics = ["accuracy"])
```

Listing 5.12: Model Compilation

In Listing 5.13, the *ImageDataGenerator* class is used to preprocesses images before use in the neural network. Using the *ImageDataGenerator* class is beneficial in deep learning because it allows images to be preprocessed using rescaling. It also enables the ability to perform on-the-fly augmentation. Both of these aspects help reduce the risk of overfitting. On line 1, an *ImageDataGenerator* object is created for the training images with 4 parameters set which determine how the image is processed. The *rescale* parameter is used to normalize the pixel values between 0 and 1. The *shear_range* parameter, 0.3, is used to augment the shearing of the image up to 30%. The *zoom_range* parameter is used as a form of augmentation to zoom an image up to 30%, given the value of 0.3. The *horizontal_flip* parameter is used to randomly flip some of the images. On line 6, the *test_data_datagen* variable, an instance of the *ImageDataGenerator* class, is created for the validation images. The *rescale* parameter is set to normalize the pixels, but none of the image augmentation values are set.

```
1 train_datagen = ImageDataGenerator(
2     rescale = 1. / 255,
3     shear_range = 0.3,
    zoom_range = 0.3,
    horizontal_flip = True)
test_datagen = ImageDataGenerator(rescale = 1. / 255)
```

Listing 5.13: Data Preprocessing

Listing 5.14 shows how the training and validation generators are created. On line 1, the data generator for the training images is created. The *train_datagen* variable is the first *ImageDataGenerator* object that was created in Listing 5.13. The *flow_from_directory* function reads the images in with specific arguments passed to it. The first parameter, set to *trainingData*, is the path to the directory containing the training images. The value of *target_size*, (180, 180), specifies the size that each image will be resized to. In this case, it will be 180 pixels by 180 pixels. The value of *batch_size* determines the number of images that will be passed into the neural network at a time. This value was defined earlier as 32. The value set for the *class_mode* parameter specifies the label encoding to be used during training. The *shuffle* parameter determines if the training images will be shuffled between epochs. In this case, the value is set to True, so they will be shuffled. It is important to shuffle the images so that each batch the model is trained on contains a random subset of the images. This helps to reduce bias and overfitting in the model during training. If the images are not shuffled, the model may learn to recognize patterns that are specific to the order of the images in the dataset, rather than learning general features that are useful for classifying all images. On line 8, the validation generator is created in the same way that the training generator was created, except it uses the *validationData* and the *Shuffle* parameter is set to *False*. The validation set was not shuffled because no training is done on this data.

```
1 train_generator = train_datagen.flow_from_directory(
2     trainingData,
    target_size =(180, 180),
    batch_size = batch_size,
    class_mode ='categorical',
    shuffle=True)

validation_generator = test_datagen.flow_from_directory(
    validationData,
    target_size =(180, 180),
    batch_size = batch_size,
    shuffle=False)
```

Listing 5.14: Data Generator Creation

Listing 5.15 shows how the model's steps are calculated. On line 1, the number of training steps that will be used per epoch is calculated. It is calculated by dividing the number of training samples by the size of each batch. On line 2, the number of validation steps that will be used is calculated. The calculation divides the number of validation samples by the batch size.

```
1 TRAIN_STEPS = train_generator.samples // batch_size
2 VAL_STEPS = validation_generator.samples // batch_size
```

Listing 5.15: Training and Validation Steps are Calculated

The code in Listing 5.16 is used to train the model. On line 1, the strategy variable created in Listing 5.10 is utilized to create a scope that will execute code on both GPUs. On line 2, the fit method is called on the model to train it. On lines 3-7, the parameter options for the model are specified. The first parameter, set to *train_generator*, determines the training data for the model. The next parameter, set to *steps_per_epoch*, determines the number of batches of data to get from the training generator before finishing an epoch. The *epochs* parameter determines the number of times to iterate over the entire training dataset before finishing. The value for this parameter is 100. The *validation_generator* is passed into the *validation_data* parameter to set the image data used for validation. The *validation_steps* parameter determines the number of batches of validation images to get from the generator before the validation epoch is finished. The callbacks parameter determines functions that will be called during training. The verbose parameter specifies how much information will be printed for each epoch.

```
with strategy.scope():
    trainingmodel = model.fit(
                                train_generator,
                                steps_per_epoch = TRAIN_STEPS,
                                epochs = epochs,
                                validation_data = validation_generator,
                                validation_steps = VAL_STEPS,
                                callbacks=[PlotLossesCallback()],
                                verbose=1)
```

Listing 5.16: Training the Model

In Listing 5.17, the *model.predict()* function applies the trained model to the data passed in and produces the output predictions. In this case, the data passed in is the validation images.

```
predictions = model.predict(validation_generator)
```

Listing 5.17: Model Predictions

Listing 5.18 shows how the model's reports are created. On line 1, the predicted class is determined for each validation image and stored in the *predicted_classes* variable. On line 3, the actual class for each of the validation images is stored in the *true_classes* variable. On line 4, the list of class labels is stored. On line 6, the scikit-learn *classification_report* function is called. This function returns a report of the models precision, recall, F1-score, and accuracy given the *true_classes* and *predicted_classes* values. On line 8, the summary of the model is printed. This includes the layer type and number of parameters for each layer of the model. On line 9, the output of the *classification_report* function is printed.

```
1 predicted_classes = np.argmax(predictions, axis = 1)
2
3 true_classes = validation_generator.classes
4 class_labels = list(validation_generator.class_indices.keys())
5
6 report = classification_report(true_classes, predicted_classes,
    target_names = class_labels)
7
8 print(model.summary())
9 print(report)
```

Listing 5.18: Classification Report and Model Summary

Listing 5.19, shows a piece of code that loads in the trained neural network and then makes predictions on an image. This code is going to used in future work for the classification of new images of *Pectinodon bakkeri* teeth. On line 1, the previously trained neural network model, saved in the H5 format, is loaded into the program. On line 3, an image is loaded in with the size set using the *target_size* parameter. On line 4, the image is converted to a NumPy array. On line 5, the image's pixel values are normalized. On line 6, the image is reshaped in order to have the correct shape for the artificial neural network. The first value passed into the *reshape* function specifies the batch size. In this case, it is set to 1, which indicates that a single image will be processed. The next 2 values, set to 180 each, determine the height and width of the image. The last value specifies the number of color channels. In this case, there are the 3 color channels for red, green, and blue. On line 8, the *predict* function is called on the model with the image passed in as the input. The output is stored in the *predictions* variable. The output for this model is an array with two values that correspond to the likelihood of the image belonging to each cluster. On line 9, the model's output for the image is printed.

```
1 model = load_model('test_model.h5')
2
3 image = load_img('./Clusters/ScaledClusters/validate/Cluster3-
    Validation/02117a.jpg', target_size=(180, 180))
4 img = np.array(image)
5 img = img / 255.0
6 img = img.reshape(1,180,180,3)
7
8 predictions = model.predict(img)
9 print(predictions)
```

Listing 5.19: Image Classifier

5.3.1 Results

The total number of image samples used was 408 with 80% of the images used for training and 20% for validation. There was an even split of 163 images used for training each class and 41 images used for the validation of each class. Originally, the second cluster had 106 training images and 27 validation images, but the image count was evened out using image augmentation as described in Section 3.1.4. The results for training the model with 100 epochs were an average of 71% accuracy, 71% precision, 70.5% recall, and 70.5% F1-score. The specific metrics for each cluster can be seen in Table 5.3. The classification accuracy and cross entropy loss for each epoch can be seen in Figure 5.8 and Figure 5.9. In both of the figures, the orange line represents the validation score and the blue line represents the training score.

Name	Accuracy	Precision	Recall	F1-Score
Cluster 1	71	70	73	71
Cluster 3	71	72	68	70

Table 5.3: Accuracy, Precision, Recall, and F1-Score for Clusters 1 and 3

In the initial experiments, the model's results indicated an overfitting issue, which is when the training accuracy is vastly better than the validation accuracy. This was resolved by tweaking the topology to include regularization layers and by reducing the total number of layers in the model. Also, it is notable that there are peaks in the data, but the overall trend is upward. In the plot of the model's loss, there are spikes toward the end, but the overall trend is still downward.

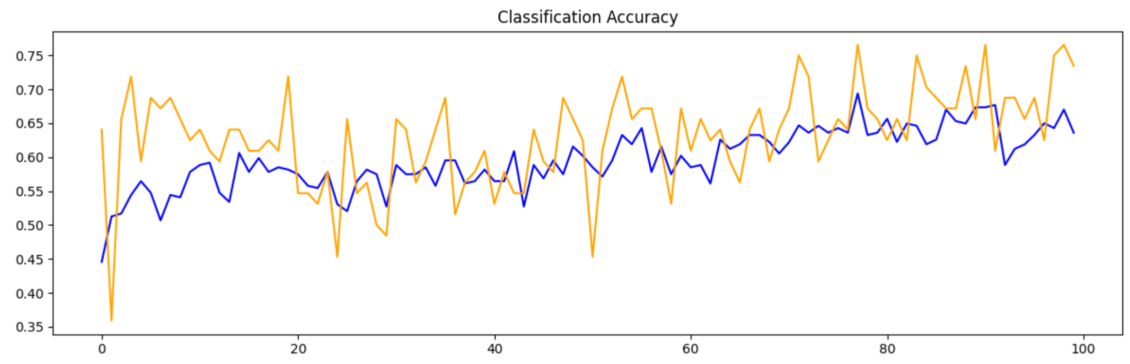


Figure 5.8: The model's accuracy per epoch for classifying images in Cluster 1 and Cluster 3

5.4 Discussion

PCA and K-Means were used to further look at the dimensionality and differences between the samples from the numerical dataset. The PCA findings revealed that

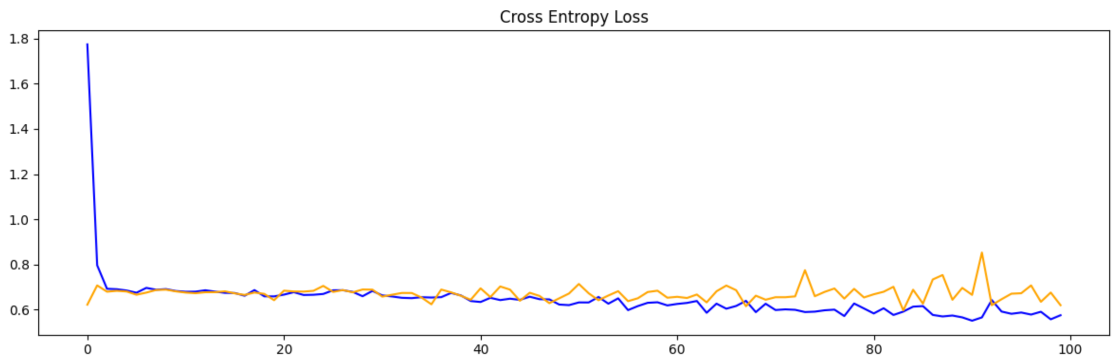


Figure 5.9: The model's loss per epoch for classifying images in Cluster 1 and Cluster 3

most of the data's variation was encompassed in 2 dimensions. This allowed to facilitate the analysis by only focusing on 2 dimensions for the K-Means analysis. The K-Means clustering was performed on dimensions 1 and 2 from the PCA, and the number of clusters was determined with the Elbow Method and Silhoutte plot. Due to the lack of a paleontologist, the clusters determined from the K-Means algorithm were unable to be confirmed. This could have led to a less accurate model since the model's clusters were determined using the K-Means output.

Detecting variations between the tooth images proved to be a challenging task. This is because there appear to be a lot of similarities between the teeth in the clusters, which made it harder to differentiate between them. The issues also could have been due to the images. There were multiple images with reflections and noise. There were also inconsistencies with the background color of the images.

The first topologies tried resulted in models with major overfitting. The final model presented was developed after many iterations with subpar results and experimentation with different techniques for tweaking the model to eliminate overfitting and improve results. One of the tweaks we tried was adding more layers to the topology. Another was to change the output layer's function from softmax to sigmoid. Adding more filters to the convolution layers was also tried. Another modification was changing the optimizer from Adam to SGD. None of these changes resulted in improved performance, and for the most part, they reduced the model's performance. Eventually, the overfitting was resolved by adding dropout layers to the model. Dropout layers help to regularize inputs and were useful with improving the results with this data set. The results also improved by reducing the number of filters in the convolution layers.

Chapter 6

Conclusions and Future Work

As of now, paleontologists are manually classifying dinosaur teeth. This proposed research work aims at saving time for paleontologists by automating the classification of images of microfossil teeth. Although there are deep learning approaches to classify microfossils, none are specifically targeting dinosaur teeth. The goal for this project was to utilize deep learning to classify microfossil *Pectinodon bakkeri* tooth images.

The first necessary step was accomplishing PCA. PCA reduces the dimensionality of a numerical dataset in order to more easily understand the data. The results from PCA were utilized in K-Means clustering, which was the next important step. K-Means clustering was used to determine the clusters that each of the teeth belong to. Following this, a Python program was created to automatically organize images based on the clusters created from K-Means. From the original three clusters, Cluster 2 was not taken into account to train the deep learning model because of low results. Therefore, images in Cluster 1 and Cluster 3 were used to train the model. The results from this model were as follows: 71% accuracy, 71% precision, 70.5% recall, and 70.5% F1-score.

The future work of this thesis is to use the classification model for determining the cluster of new *Pectinodon bakkeri* tooth images that may be collected. Along with this, image segmentation will be applied to the images in order to clearly separate the tooth from the background. A limitation of this research was not considering tooth size during the training process (i.e., all the images had the same size). So, another aspect of future work will be to account for the size of teeth when training the model. Finally, the results are going to be submitted to a paleontologist for validation.

Bibliography

[1] D. W. Larson and P. J. Currie, "Multivariate analyses of small theropod dinosaur teeth and implications for paleoecological turnover through time," *PLoS One*, vol. 8, no. 1, p. e54329, 2013. (document), 2.2, 2.2.2

[2] K. Aguilar, G. H. Alférez, and C. Aguilar, "Detection of difficult airway using deep learning," *Machine Vision and Applications*, vol. 31, pp. 1–11, 2020. (document), 2.3, 2.4, 2.5, 2.2.1

[3] J. Rollins, "Foundational methodology for data science, IBM anal.(2015)." (document), 3, 3.1

[4] J. H. Lipps, "Microfossils." [Online]. Available: https://ucmp.berkeley.edu/fosrec/Lipps1.html 1.1, 2.1.1

[5] K. Mimura, S. Minabe, K. Nakamura, K. Yasukawa, J. Ohta, and Y. Kato, "Automated detection of microfossil fish teeth from slide images using combined deep learning models," *Applied Computing and Geosciences*, vol. 16, p. 100092, 2022. 1.1, 2.2.3

[6] D. G. Demar, "An illustrated guide to latest cretaceous vertebrate microfossils of the hell creek formation of northeastern montana." 1.3, 2.2.2, 3.1

[7] J. O. Farlow, D. L. Brinkman, W. L. Abler, and P. J. Currie, "Size, shape, and serration density of theropod dinosaur lateral teeth," *Modern Geology*, vol. 16, no. 1-2, pp. 161–198, 1991. 1.3, 2.2.2

[8] C. Ding and X. He, "K-means clustering via principal component analysis," in *Proceedings of the twenty-first international conference on Machine learning*, 2004, p. 29. 2.1.2, 2.1.3

[9] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow.* " O'Reilly Media, Inc.", 2022. 2.1.4, 2.1.8

[10] G. H. Alférez, O. A. Esteban, B. L. Clausen, and A. M. M. Ardila, "Automated machine learning pipeline for geochemical analysis," *Earth Science Informatics*, vol. 15, no. 3, pp. 1683–1698, 2022. 2.1.4

[11] K. G. Kim, "Book review: Deep learning," *Healthcare informatics research*, vol. 22, no. 4, pp. 351–354, 2016. 2.1.5

[12] A. Zerium, "Demystifying convolutional neural networks," Sep 2018. [Online]. Available: https://medium.com/@eternalzerodayx/demystifying-convolutional-neural-networks-ca17bdc75559 2.1.6

[13] R. Szeliski, *Computer vision: algorithms and applications.* Springer Nature, 2022. 2.1.7

[14] R. Klette, *Concise computer vision.* Springer, 2014, vol. 233. 2.1.7

[15] Simplilearn, "What is Keras and why it so popular in 2021: Simplilearn," Dec 2022. [Online]. Available: https://www.simplilearn.com/tutorials/deep-learning-tutorial/what-is-keras 2.1.9

[16] G. H. Alférez, E. L. Vázquez, A. M. M. Ardila, and B. L. Clausen, “Automatic classification of plutonic rocks with deep learning,” *Applied Computing and Geosciences*, vol. 10, p. 100061, 2021. 2.2.1

[17] L. G. Olivas, G. H. Alférez, and J. Castillo, “Glaucoma detection in latino population through oct’s rnfl thickness map using transfer learning,” *International Ophthalmology*, vol. 41, pp. 3727–3741, 2021. 2.2.1

[18] T. Itaki, Y. Taira, N. Kuwamori, T. Maebayashi, S. Takeshima, and K. Toya, “Automated collection of single species of microfossils using a deep learning–micromanipulator system,” *Progress in Earth and Planetary Science*, vol. 7, no. 1, pp. 1–7, 2020. 2.2.3

[19] J. Renaudie, R. Gray, and D. B. Lazarus, “Accuracy of a neural net classification of closely-related species of microfossils from a sparse dataset of unedited images,” *PeerJ Preprints*, vol. 6, p. e27328v1, 2018. 2.2.3

[20] T. Itaki, Y. Taira, N. Kuwamori, H. Saito, M. Ikehara, and T. Hoshino, “Innovative microfossil (radiolarian) analysis using a system for automated image collection and ai-based classification of species,” *Scientific reports*, vol. 10, no. 1, pp. 1–9, 2020. 2.2.3