**BAHRIA UNIVERSITY KARACHI**

**CAMPUS**



**CV CCP**

|  |  |
| --- | --- |
| **Group Members** | **Enrollment Number** |
| Noor | 02-136221-001 |
| Emaan Safdar | 02-136221-022 |
| Urooj Moin | 02-136221-036 |
| Zuha Kehar | 02-136221-038 |

[Introduction 2](#_Toc1396023945)

[Problem Statement 2](#_Toc1659364095)

[Dataset 2](#_Toc610475142)

[Class Division 2](#_Toc876869777)

[Data Splitting 3](#_Toc1025251931)

[Model 3](#_Toc985950429)

[Approach 3](#_Toc1778915415)

[Architecture 3](#_Toc237557595)

[Code: 3](#_Toc1775564029)

[Workflow 21](#_Toc1507410609)

[Results Matrix 22](#_Toc798709169)

[Result and Conclusion 24](#_Toc776754794)

**Animal Detection**

# **Introduction**

The Animal Detection Dataset consists of images of three classes of animals: Cats, Dogs, and Pandas. The objective of this project is to classify images into these three classes by employing segmentation and classification techniques using OpenCV and CNN.

# **Problem Statement**

Given an image dataset with three distinct classes (Cats, Dogs, and Pandas), the objective is to:

* Segment regions of interest within each image.
* Train a CNN that classifies the segmented regions accurately into the corresponding categories.
* Assess its performance with suitable metrics.

# **Dataset**

## **Class Division**

The dataset contains labeled images divided into three classes:

* **Class A**: Cats
* **Class B**: Dogs
* **Class C**: Pandas

## **Data Splitting**

The dataset is divided into three subsets:

* **Training Set:** 80% of the dataset.
* **Validation Set**: 10% of the dataset.
* **Testing Set**: 20% of the dataset.

# **Model**

## **Approach**

**Segmentation:**

* Use OpenCV techniques such as thresholding and edge detection for region of interest extraction.

**Classification:**

* Train a CNN on the segmented regions to classify them into the respective classes.

## **Architecture**

**Segmentation:** Implemented segmentation using OpenCVs thresholding and canny edge detection

**CNN:**

* Mobile net model is used
* Global Average Pooling 2D Layer
* Fully Connected Dense layer (128 units, ReLU activation).
* Dropout layer (50% dropout).
* Output layer (3 classes, softmax activation).

## **Code:**

import os

import shutil

import random

def create\_directories(base\_dir, classes, splits):

"""

Create the necessary directories for train, test, and validation splits.

"""

for class\_name in classes:

for split in splits:

# Create base class directories

split\_path = os.path.join(base\_dir, split, class\_name)

os.makedirs(split\_path, exist\_ok=True)

def organize\_dataset(input\_dir, target\_dir, num\_train=200, num\_test=100):

"""

Organize dataset by creating train, test, and validation splits.

"""

classes = ['cats', 'dogs', 'panda']

splits = ['train', 'test', 'validation']

# Create the necessary directories

create\_directories(target\_dir, classes, splits)

for category in classes:

# Get all image files in the category

category\_input = os.path.join(input\_dir, category)

image\_files = os.listdir(category\_input)

# Shuffle the images for random splitting

random.shuffle(image\_files)

# Split images into train, test, and validation

train\_files = image\_files[:num\_train]

test\_files = image\_files[num\_train:num\_train + num\_test]

validation\_files = image\_files[num\_train + num\_test:]

# Copy files to corresponding directories

for img\_name in train\_files:

src = os.path.join(category\_input, img\_name)

dst = os.path.join(target\_dir, 'train', category, img\_name)

shutil.copy(src, dst)

for img\_name in test\_files:

src = os.path.join(category\_input, img\_name)

dst = os.path.join(target\_dir, 'test', category, img\_name)

shutil.copy(src, dst)

for img\_name in validation\_files:

src = os.path.join(category\_input, img\_name)

dst = os.path.join(target\_dir, 'validation', category, img\_name)

shutil.copy(src, dst)

# Example usage

input\_dir = '/content/drive/MyDrive/animals' # Path where original images are stored

target\_dir = '/content/dataset/dataset/organized\_splits' # Path where the organized dataset will be stored

organize\_dataset(input\_dir, target\_dir)

Applying Canny edge detection and Threshold Segmentation:

import os

import cv2

import shutil

def preprocess\_and\_save(input\_dir, output\_dir, method='threshold'):

"""

Preprocess images using either 'threshold' or 'edge' method and save them.

"""

for category in ['cats', 'dogs', 'panda']:

for split in ['train', 'test', 'validation']:

category\_input = os.path.join(input\_dir, split, category)

category\_output = os.path.join(output\_dir, method, split, category)

os.makedirs(category\_output, exist\_ok=True)

for img\_name in os.listdir(category\_input):

img\_path = os.path.join(category\_input, img\_name)

img = cv2.imread(img\_path, cv2.IMREAD\_GRAYSCALE)

# Apply preprocessing (threshold or edge detection)

if method == 'threshold':

\_, processed\_img = cv2.threshold(img, 128, 255, cv2.THRESH\_BINARY)

elif method == 'edge':

processed\_img = cv2.Canny(img, 100, 200)

# Save the processed image

processed\_img\_path = os.path.join(category\_output, img\_name)

cv2.imwrite(processed\_img\_path, processed\_img)

# Define paths

input\_dir = '/content/dataset/dataset/organized\_splits' # Path to the organized dataset

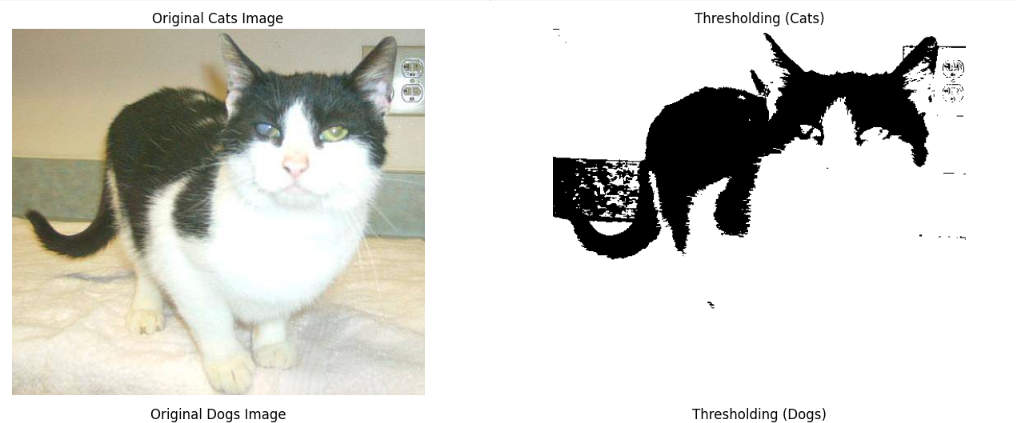
output\_dir = '/content/dataset/dataset/processed' # Path where processed images will be saved

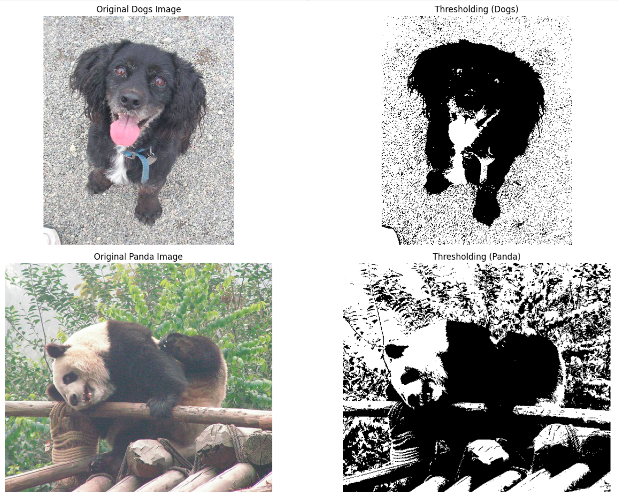
# Apply preprocessing for threshold and edge detection

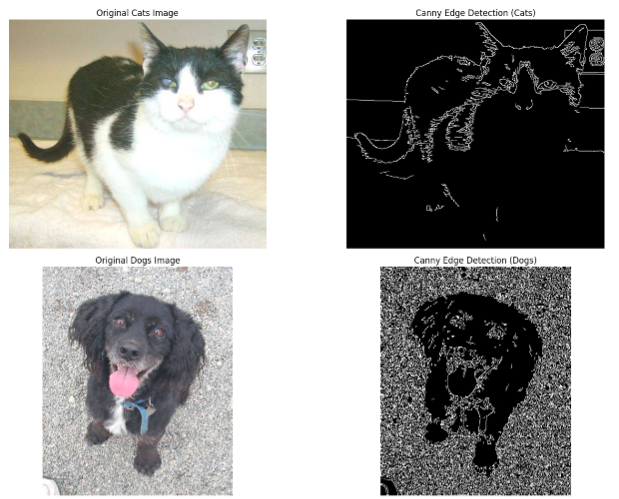
preprocess\_and\_save(input\_dir, output\_dir, method='threshold') # Thresholded images

preprocess\_and\_save(input\_dir, output\_dir, method='edge') # Edge-detected images









import numpy as np

import os

import cv2

from tensorflow.keras import layers, models

from tensorflow.keras.callbacks import EarlyStopping

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.applications import MobileNet

from tensorflow.keras import regularizers

import tensorflow as tf

# Data Augmentation setup

train\_datagen = ImageDataGenerator(

rescale=1./255, # Normalize pixel values to [0, 1]

rotation\_range=40, # Randomly rotate images

width\_shift\_range=0.2, # Randomly shift horizontally

height\_shift\_range=0.2, # Randomly shift vertically

shear\_range=0.2, # Apply random shearing

zoom\_range=0.2, # Apply random zoom

horizontal\_flip=True, # Randomly flip images horizontally

fill\_mode='nearest' # Fill in pixels that appear due to transformations

)

val\_datagen = ImageDataGenerator(rescale=1./255) # For validation, only rescale

# Function to load images and their labels from a directory

def load\_data\_from\_directory(directory):

images = []

labels = []

label\_map = {'cats': 0, 'dogs': 1, 'panda': 2} # Map labels to integers

for category in ['cats', 'dogs', 'panda']:

category\_path = os.path.join(directory, category)

for img\_name in os.listdir(category\_path):

img\_path = os.path.join(category\_path, img\_name)

# Read the image and resize it

img = cv2.imread(img\_path)

img = cv2.resize(img, (200, 200)) # Resize to (200, 200) as per your model input

images.append(img)

labels.append(label\_map[category])

return np.array(images), np.array(labels)

# Define paths for thresholded and edge-detected data

train\_threshold\_path = '/content/dataset/dataset/processed/threshold/train'

test\_threshold\_path = '/content/dataset/dataset/processed/threshold/test'

val\_threshold\_path = '/content/dataset/dataset/processed/threshold/validation'

train\_edge\_path = '/content/dataset/dataset/processed/edge/train'

test\_edge\_path = '/content/dataset/dataset/processed/edge/test'

val\_edge\_path = '/content/dataset/dataset/processed/edge/validation'

# Load the data

X\_train\_thresh, y\_train\_thresh = load\_data\_from\_directory(train\_threshold\_path)

X\_test\_thresh, y\_test\_thresh = load\_data\_from\_directory(test\_threshold\_path)

X\_val\_thresh, y\_val\_thresh = load\_data\_from\_directory(val\_threshold\_path)

X\_train\_edge, y\_train\_edge = load\_data\_from\_directory(train\_edge\_path)

X\_test\_edge, y\_test\_edge = load\_data\_from\_directory(test\_edge\_path)

X\_val\_edge, y\_val\_edge = load\_data\_from\_directory(val\_edge\_path)

# Normalize the data

X\_train\_thresh = X\_train\_thresh / 255.0

X\_test\_thresh = X\_test\_thresh / 255.0

X\_val\_thresh = X\_val\_thresh / 255.0

X\_train\_edge = X\_train\_edge / 255.0

X\_test\_edge = X\_test\_edge / 255.0

X\_val\_edge = X\_val\_edge / 255.0

CNN (Mobilenet)

# Define MobileNet-based model function

def create\_mobilenet\_model(input\_shape=(200, 200, 3)):

base\_model = MobileNet(weights='imagenet', include\_top=False, input\_shape=input\_shape)

base\_model.trainable = False # Freeze the base model layers

model = models.Sequential([

base\_model,

layers.GlobalAveragePooling2D(),

layers.Dense(128, activation='relu', kernel\_regularizer=regularizers.l2(0.01)),

layers.Dropout(0.5),

layers.Dense(3, activation='softmax') # 3 output classes (cats, dogs, pandas)

])

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

# Train and validation generators for thresholded images

train\_generator\_thresh = train\_datagen.flow\_from\_directory(

train\_threshold\_path,

target\_size=(200, 200),

batch\_size=16,

class\_mode='sparse'

)

val\_generator\_thresh = val\_datagen.flow\_from\_directory(

val\_threshold\_path,

target\_size=(200, 200),

batch\_size=16,

class\_mode='sparse'

)



# Train the model with thresholded images

mobilenet\_model\_thresh = create\_mobilenet\_model()

mobilenet\_model\_thresh.fit(

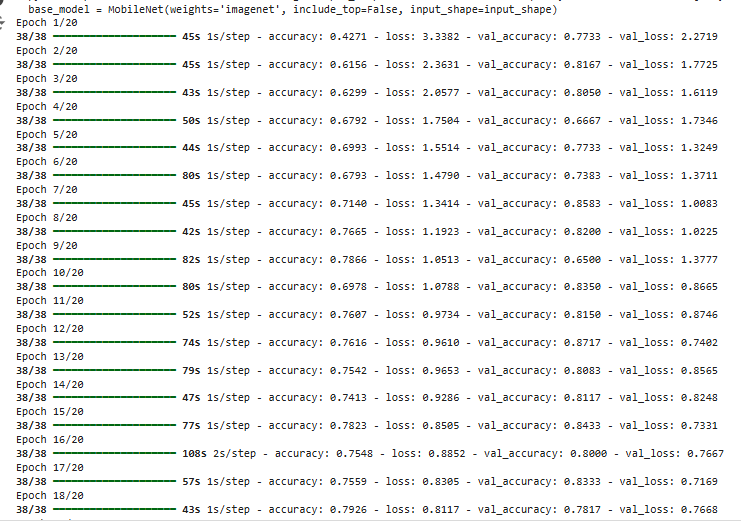
train\_generator\_thresh,

epochs=20,

validation\_data=val\_generator\_thresh,

callbacks=[early\_stopping]

)



from sklearn.metrics import classification\_report, confusion\_matrix

import numpy as np

import matplotlib.pyplot as plt

# Evaluate threshold model on test set

test\_generator\_thresh = val\_datagen.flow\_from\_directory(

test\_threshold\_path,

target\_size=(200, 200),

batch\_size=16,

class\_mode='sparse',

shuffle=False # Ensure predictions match the ground truth

)

test\_loss\_thresh, test\_acc\_thresh = mobilenet\_model\_thresh.evaluate(test\_generator\_thresh)

print(f'Threshold Model - Test Accuracy: {test\_acc\_thresh}, Test Loss: {test\_loss\_thresh}')

# Calculate classification report for threshold model

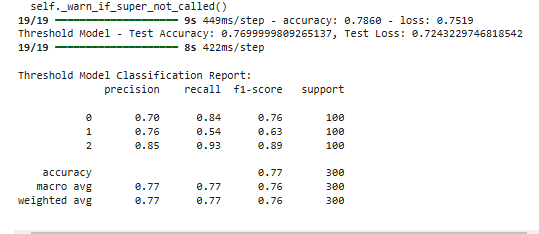
y\_true\_thresh = test\_generator\_thresh.classes

y\_pred\_thresh = mobilenet\_model\_thresh.predict(test\_generator\_thresh, verbose=1)

y\_pred\_thresh = np.argmax(y\_pred\_thresh, axis=1)

print("\nThreshold Model Classification Report:")

print(classification\_report(y\_true\_thresh, y\_pred\_thresh))



# Get true labels and predictions for confusion matrix

y\_true = test\_generator\_thresh.classes # True labels

y\_pred = model.predict(test\_generator\_thresh) # Predictions from the model

y\_pred\_classes = np.argmax(y\_pred, axis=1) # Convert probabilities to predicted class labels

# Generate confusion matrix

cm = confusion\_matrix(y\_true, y\_pred\_classes)

# Plot confusion matrix

plt.figure(figsize=(8, 6))

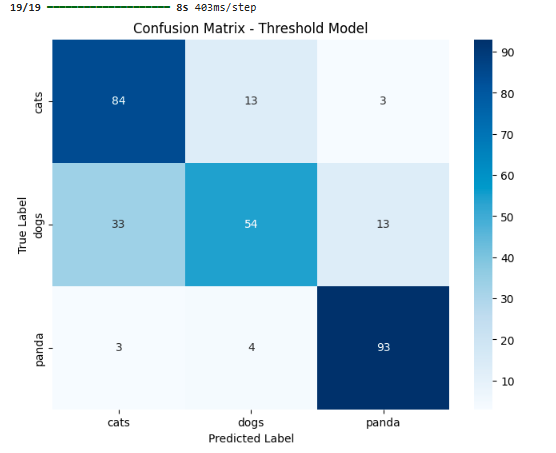
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=test\_generator\_thresh.class\_indices.keys(), yticklabels=test\_generator\_thresh.class\_indices.keys())

plt.title("Confusion Matrix - Threshold Model")

plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.show()



# Train and validation generators for edge-detected images

train\_generator\_edge = train\_datagen.flow\_from\_directory(

train\_edge\_path,

target\_size=(200, 200),

batch\_size=16,

class\_mode='sparse'

)

val\_generator\_edge = val\_datagen.flow\_from\_directory(

val\_edge\_path,

target\_size=(200, 200),

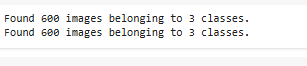
batch\_size=16,

class\_mode='sparse'

)

# Early stopping callback

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)



# Train the model with edge-detected images

mobilenet\_model\_edge = create\_mobilenet\_model()

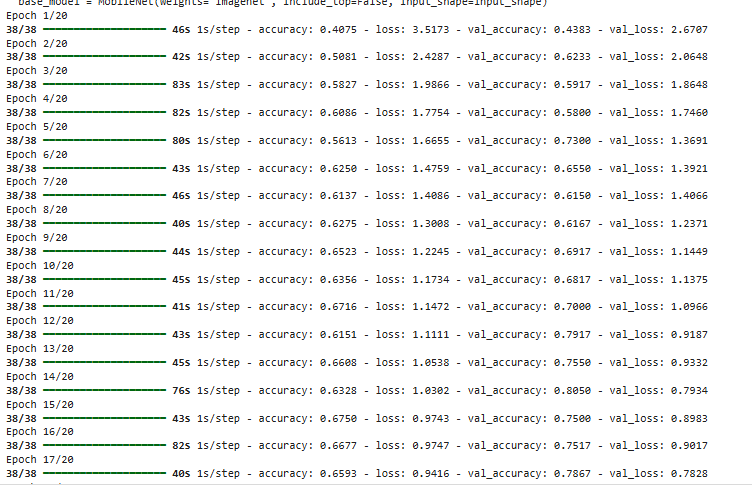
mobilenet\_model\_edge.fit(

train\_generator\_edge,

epochs=20,

validation\_data=val\_generator\_edge

)



# Evaluate edge model on test set

test\_generator\_edge = val\_datagen.flow\_from\_directory(

test\_edge\_path,

target\_size=(200, 200),

batch\_size=16,

class\_mode='sparse',

shuffle=False # Ensure predictions match the ground truth

)

test\_loss\_edge, test\_acc\_edge = mobilenet\_model\_edge.evaluate(test\_generator\_edge)

print(f'Edge Model - Test Accuracy: {test\_acc\_edge}, Test Loss: {test\_loss\_edge}')

# Calculate classification report for edge model

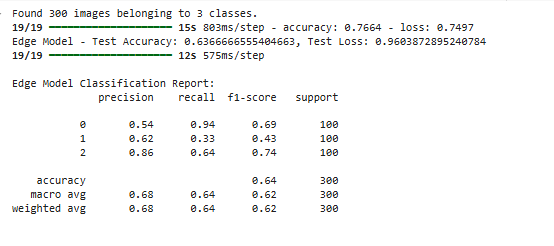
y\_true\_edge = test\_generator\_edge.classes

y\_pred\_edge = mobilenet\_model\_edge.predict(test\_generator\_edge, verbose=1)

y\_pred\_edge = np.argmax(y\_pred\_edge, axis=1)

print("\nEdge Model Classification Report:")

print(classification\_report(y\_true\_edge, y\_pred\_edge))



# Get true labels and predictions for confusion matrix

y\_true = test\_generator\_edge.classes # True labels

y\_pred = model.predict(test\_generator\_edge) # Predictions from the model

y\_pred\_classes = np.argmax(y\_pred, axis=1) # Convert probabilities to predicted class labels

# Generate confusion matrix

cm = confusion\_matrix(y\_true, y\_pred\_classes)

# Plot confusion matrix

plt.figure(figsize=(8, 6))

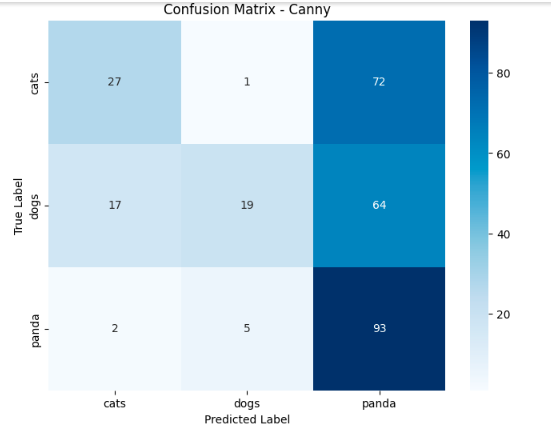
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=test\_generator\_thresh.class\_indices.keys(), yticklabels=test\_generator\_thresh.class\_indices.keys())

plt.title("Confusion Matrix - Canny")

plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.show()





**Prediction of new Image:**

# Function to preprocess the image (resize and normalize)

def preprocess\_image(img\_path):

img = cv2.imread(img\_path, cv2.IMREAD\_COLOR) # Read in color (since model is trained on color images)

img\_resized = cv2.resize(img, (100, 100)) # Resize to match input size of the model

img\_normalized = img\_resized / 255.0 # Normalize the image to [0, 1]

img\_expanded = np.expand\_dims(img\_normalized, axis=0) # Add batch dimension

return img\_expanded

# Function to predict the class of a new image

def predict\_class(model, img\_path):

img = preprocess\_image(img\_path) # Preprocess the image

prediction = model.predict(img) # Make the prediction

predicted\_class = np.argmax(prediction, axis=1)[0] # Get the index of the predicted class

# Map the predicted class index to the corresponding label

labels = ['cats', 'dogs', 'pandas']

predicted\_label = labels[predicted\_class]

return predicted\_label

img\_path = '/content/dataset/dataset/Cat.jpg' # Replace with the path to your new image

# Predict using the threshold model (replace with your model)

predicted\_class\_thresh = predict\_class(mobilenet\_model\_thresh, img\_path)

print(f"Predicted class (Threshold model): {predicted\_class\_thresh}")

plt.imshow(cv2.imread(img\_path))

plt.axis('off')

plt.show()



img\_path = '/content/dataset/dataset/001.jpeg' # Replace with the path to your new image

# Predict using the threshold model (replace with your model)

predicted\_class\_thresh = predict\_class(mobilenet\_model\_thresh, img\_path)

print(f"Predicted class (Threshold model): {predicted\_class\_thresh}")

plt.imshow(cv2.imread(img\_path))

plt.axis('off')

plt.show()



img\_path = '/content/dataset/dataset/pop.jpeg' # Replace with the path to your new image

# Predict using the threshold model (replace with your model)

predicted\_class\_thresh = predict\_class(mobilenet\_model\_thresh, img\_path)

print(f"Predicted class (Threshold model): {predicted\_class\_thresh}")

plt.imshow(cv2.imread(img\_path))

plt.axis('off')

plt.show()





img\_path = '/content/dod1.jpeg'

# Predict using the threshold model (replace with your model)

predicted\_class\_edge = predict\_class(mobilenet\_model\_edge, img\_path)

print(f"Predicted class (edge\_model): {predicted\_class\_edge}")

plt.imshow(cv2.imread(img\_path))

plt.axis('off')

plt.show()



img\_path = '/content/catu.jpeg'

# Predict using the threshold model (replace with your model)

predicted\_class\_thresh = predict\_class(mobilenet\_model\_thresh, img\_path)

print(f"Predicted class (Threshold model): {predicted\_class\_thresh}")

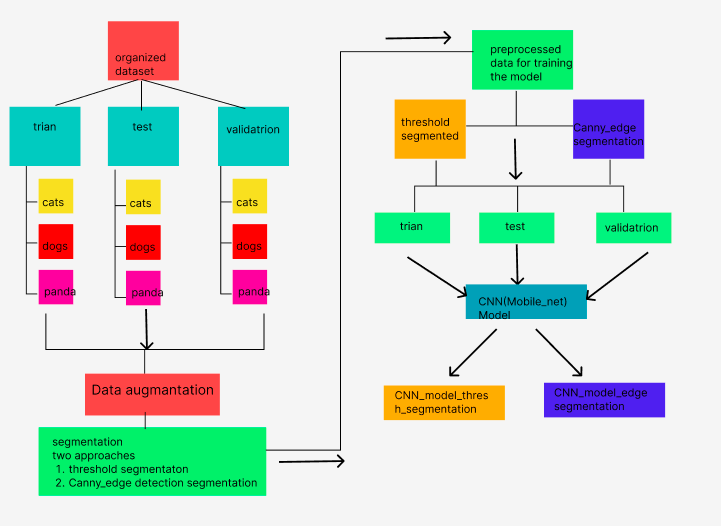
plt.imshow(cv2.imread(img\_path))

plt.axis('off')

plt.show()

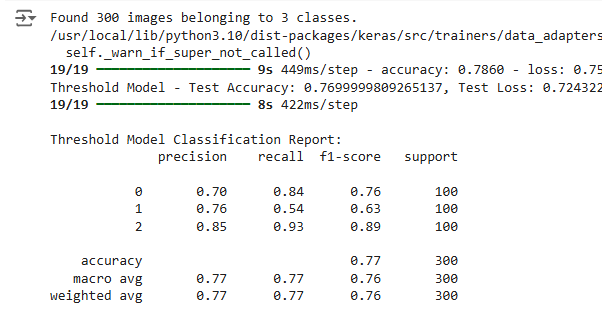


## **Workflow**

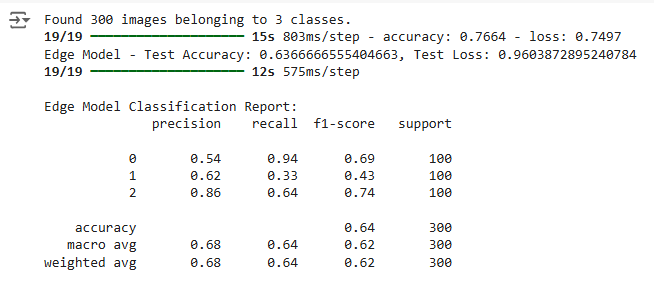


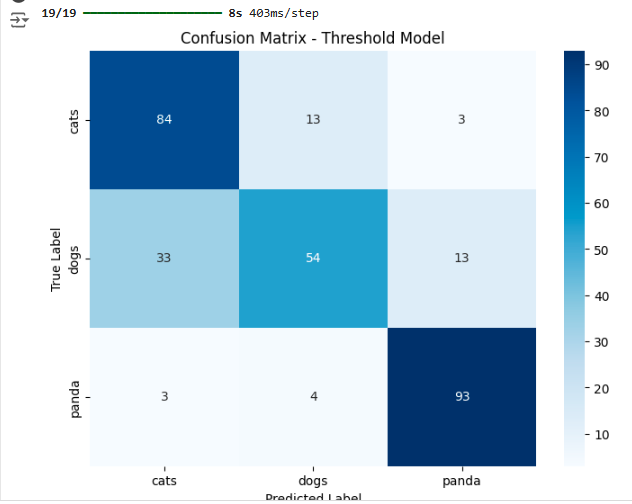
# **Results Matrix**

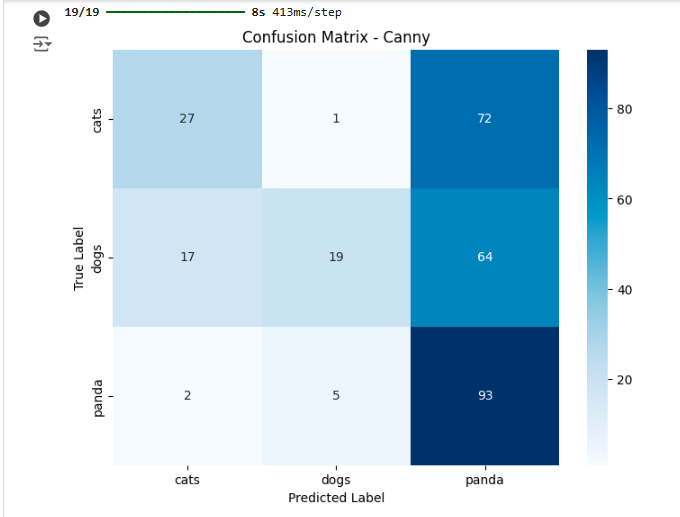
**Classification report threshold model**



**Classification report for Edge Model**

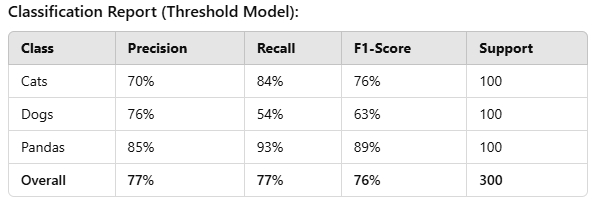




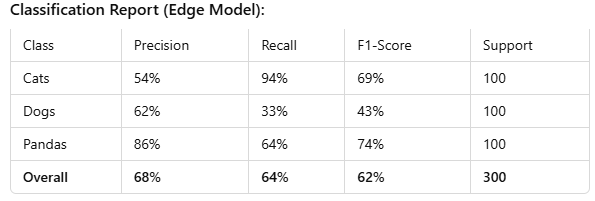


# **Result and Conclusion**

* **Test Accuracy (Threshold Model):** 76.9%
* **Test Loss (Threshold Model):** 0.7243



* **Test Accuracy (Edge Model):** 63.7%
* **Test Loss (Edge Model):** 0.9604



The threshold-based preprocessing method demonstrates higher accuracy and more consistent performance across all three classes compared to the edge detection method. The edge model, while effective for cat classification, struggles with distinguishing dogs, resulting in reduced overall accuracy. Future work may involve combining both thresholding and edge detection techniques to leverage their strengths. Additionally, enhancing data augmentation, fine-tuning the model, and experimenting with more advanced segmentation techniques could improve classification outcomes.