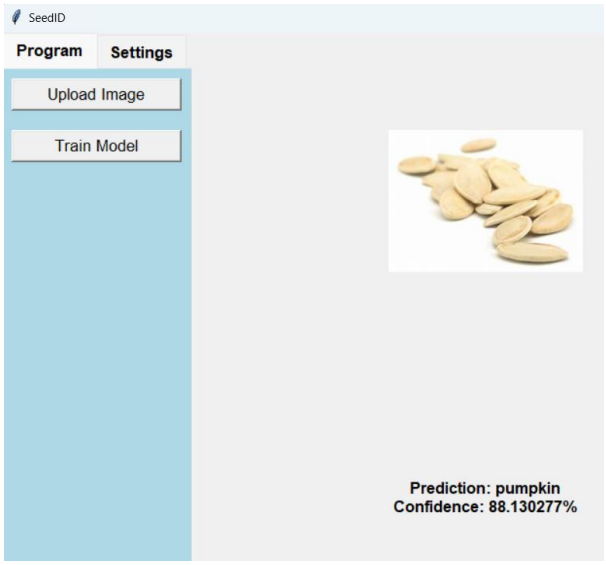
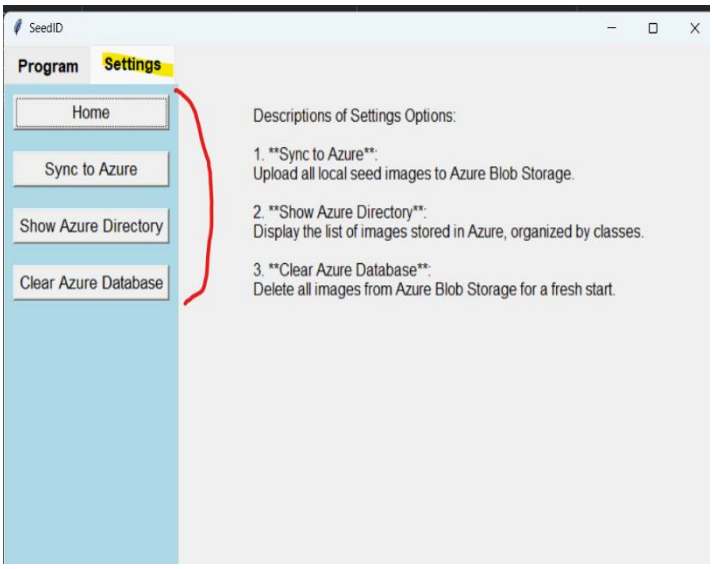
Matt Morrow

CIS5898

Last Interim Progress Report

As of the last check in, the program has come along nicely with new enhancements to the UI, expanded seed training, and even further improvements to identification accuracy. One of the most significant additions during this leg of the project was the integration of Azure into the UI and the ability to store/receive images from the cloud, train the model locally using the machines GPU, and then storing the trained model back into the cloud. Tabs were added to the sidebar of the UI to separate the main program functionality from the administrative settings for managing Azure, as seen below:





There are a few opportunity areas however that need to be addressed. Though the images that are within the classes are very accurately identified, a completely foreign image, such as a baseball which is round in nature, would be identified by the program as a coconut and not immediately dismissed as a non-object. I have tried several different attempts to get this working but without avail. There are also issues with the banner and some of the tab functionality where some of the tkinter object stick and do not disappear. Lastly, the project still needs unit testing coverage and to have the classes broken out into separate files, which I hope to have done by the end of the project.

As requested via email, please see the proceeding pages for the projects code (as PDF). There is one for the main project and then a separate file for the program's logging:

MAIN PROJECT (SEEDID.PY) FILE

```
# %% Import libraries
import os
import shutil
import tkinter as tk
from tkinter import filedialog, messagebox, ttk
import threading
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms, models
from PIL import Image, ImageTk
import numpy as np
from azure.storage.blob import BlobServiceClient
from azure_upload import upload_image_to_azure
from azure_retrieve import retrieve_image_from_azure
import logging
from CSVLogHelper import CSVLogHandler   #Import CSVLogger from CSVLogHelper.py
import shutil


#TODO: FIX THE ALERT BANNER
#TODO: FIX HOME BUTTON AFTER AZURE DIRECOTRY IS PRESSED
#TODO: MAKE HOME GO AWAY AFTER PROGRAM TAB IS PRESSED


#CITATION: Benham, A. (2022, September 30). Deep learning tutorial for beginners | AI neural networks explained. YouTube. https://www.youtube.com/watch?v=r7Am-ZGMef8
#Setting up CUDA and GPU usage
```

#Check for GPU availability

%% Set up device and logger

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

print(f"Using device: {device}")

#Initialize the logger helper class

logger = logging.getLogger("SeedIdentifierApp")

logger.setLevel(logging.INFO)

logger.addHandler(CSVLogHandler())

#CLASS FIXED CONSTANT VARIABLES; Controls for the program

#Number of seeds in the directory to ID

%% Define constants

class_nums = 6

class_names = ['coconut', 'corn', 'flaxseed', 'pumpkin', 'sunflower', 'wheat']

epoch_num = 75

LEARN_RATE = 1e-2 #A smaller learning rate (like 2 or 3) reduces the size of the steps; allowing the model to move more precisely toward the minimum of the loss function.

#Azure Blob Manager for image storage

#CITATION: Microsoft. Connect to and query Azure SQL Database using Python and the pyodbc driver https://learn.microsoft.com/en-us/azure/azure-sql/database/azure-sql-python-quickstart?view=azuresql&tabs=windows%2Csql-inter

%% Azure Blob Manager Class

class AzureImageManager:

  def __init__(self):

    self.connection_string = "DefaultEndpointsProtocol=https;AccountName=fitseedid;AccountKey=P+Gn6AEYlmaEvhMZUkpzOnCVoEKP8kAniBg8YL5coK/ACbvJq9hzXFNys1FBVRbSA8NtZRf3tZYz+ASt/44GrA==;EndpointSuffix=core.windows.net"

    self.container_name = "seedimages"

```
        self.blob_service_client = BlobServiceClient.from_connection_string(self.connection_string)
        self.container_client = self.blob_service_client.get_container_client(self.container_name)
        logger.info("Azure Blob Manager initialized.")

    def upload_image(self, blob_name):
        logger.info(f"Uploading image: {blob_name}")
        upload_image_to_azure(self.connection_string, self.container_name, blob_name)

    def retrieve_image(self, blob_name):
        logger.info(f"Retrieving image: {blob_name}")
        image_tensor = retrieve_image_from_azure(self.connection_string, self.container_name,
blob_name)
        pil_image = Image.fromarray(image_tensor)
        return pil_image

    def delete_all_blobs(self):
        logger.info("Deleting all blobs in the container.")
        blob_list = self.container_client.list_blobs()
        for blob in blob_list:
            self.container_client.delete_blob(blob.name)

    # %% Download all images
    def download_all_images(self, local_directory):
        # Delete the directory if it already exists
        if os.path.exists(local_directory):
            shutil.rmtree(local_directory)
            print(f"Deleted existing directory: {local_directory}")

        # Create the main directory
```

```
    os.makedirs(local_directory)
    print(f"Created main directory: {local_directory}")

    #Pre-create class folders in the main directory
    for class_name in class_names:
        class_folder = os.path.join(local_directory, class_name)
        os.makedirs(class_folder, exist_ok=True)
        print(f"Created folder for class '{class_name}': {class_folder}")

    blobs = self.container_client.list_blobs()
    for blob in blobs:
        blob_client = self.container_client.get_blob_client(blob)

        #Check if the blob name contains a class name as a prefix
        assigned_class = None
        for class_name in class_names:
            if class_name in blob.name:
                assigned_class = class_name
                break

        if assigned_class is None:
            logger.error(f"No matching class found for blob name: {blob.name}")
            continue

        #Define the download path within the appropriate class folder
        download_path = os.path.join(local_directory, assigned_class,
os.path.basename(blob.name))

        #Download the blob to the specified path
```

```
            try:
                with open(download_path, "wb") as file:
                    data = blob_client.download_blob()
                    file.write(data.readall())
                logger.info(f"Downloaded {blob.name} to {download_path}")
                print(f"Downloaded '{blob.name}' to '{download_path}'")
            except Exception as e:
                logger.error(f"Failed to download {blob.name}: {e}")
                print(f"Failed to download {blob.name}: {e}")

        # Print the directory structure for verification
        print("\nContents of temp_azure_images directory after download:")
        for root, dirs, files in os.walk(local_directory):
            print(f"Directory: {root}")
            for name in dirs:
                print(f"  Subdirectory: {name}")
            for name in files:
                print(f"  File: {name}")

    def list_images_by_class(self, class_names):
        class_dict = {class_name: [] for class_name in class_names}

        # List all blobs in the container
        blobs = self.container_client.list_blobs()
        for blob in blobs:
            blob_name = blob.name
            # Check if the blob name contains a class name as a prefix or directory
            for class_name in class_names:
                if class_name in blob_name:
```

```
                class_dict[class_name].append(blob_name)
                break
        return class_dict


    def clear_container(self):
        logger.info("Deleting all blobs in the Azure container.")
        blob_list = self.container_client.list_blobs()
        for blob in blob_list:
            try:
                self.container_client.delete_blob(blob.name)
                logger.info(f"Deleted blob: {blob.name}")
                print(f"Deleted blob: {blob.name}")
            except Exception as e:
                logger.error(f"Failed to delete blob {blob.name}: {e}")
                print(f"Failed to delete blob {blob.name}: {e}")
        logger.info("Container cleared successfully.")
        print("Azure container cleared successfully.")


    #Downloads the trained model from Azure to a temporary location.
    #CITATION: Microsoft. Download a blob with Python https://learn.microsoft.com/en-
us/azure/storage/blobs/storage-blob-download-python
    # %% Download the model from Azure
    def download_model_from_azure(self, model_path, blob_name="trained_model.pth"):
        #Downloads the trained model from Azure to a temporary location.
        try:
            blob_client = self.container_client.get_blob_client(blob_name)
            with open(model_path, "wb") as file:
                data = blob_client.download_blob()
                file.write(data.readall())
```

```
            logger.info(f"Model downloaded from Azure as {blob_name}")
            print(f"Model downloaded from Azure as {blob_name}")

        except Exception as e:
            logger.error(f"Failed to download model from Azure: {e}")
            print(f"Failed to download model from Azure: {e}")

    #Uploads an image file to Azure with a specified blob name.
    def upload_image_with_path(self, file_path, blob_name):
        try:
            blob_client = self.container_client.get_blob_client(blob_name)
            with open(file_path, "rb") as data:
                blob_client.upload_blob(data, overwrite=True)
            logger.info(f"Successfully uploaded {blob_name} to Azure.")
            print(f"Uploaded {blob_name} to Azure.")
        except Exception as e:

            logger.error(f"Failed to upload {file_path} to Azure: {e}")
            print(f"Failed to upload {file_path} to Azure: {e}")

#CITATION: GeeksForGeeks. Residual Networks (ResNet) – Deep Learning
https://www.geeksforgeeks.org/residual-networks-resnet-deep-learning/
# %% TrainModel Class
class TrainModel:
    def __init__(self, num_classes=class_nums, azure_manager=None):
        self.num_classes = num_classes
        self.model = None
        self.azure_manager = azure_manager  # Pass AzureImageManager instance here
```

```
  def load_data(self, data_dir, img_size=256, batch_size=32):
    logger.info(f"Loading data from directory: {data_dir}")
    transform = transforms.Compose([
      transforms.RandomResizedCrop(256, scale=(0.8, 1.0)),
      transforms.RandomHorizontalFlip(),
      transforms.RandomVerticalFlip(),
      transforms.RandomRotation(20),
      transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1),
      transforms.ToTensor(),
      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])  # Standard normalization
for ResNet
    ])

    train_ds = datasets.ImageFolder(root=data_dir, transform=transform)
    val_size = int(0.2 * len(train_ds))
    train_size = len(train_ds) - val_size
    train_ds, val_ds = torch.utils.data.random_split(train_ds, [train_size, val_size])

    train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_ds, batch_size=batch_size, shuffle=False)

    class_names = train_ds.dataset.classes

    return train_loader, val_loader, class_names

  def build_model(self, input_shape=(256, 256, 3), num_classes=class_nums):
    print("Building model...")
    self.model = models.resnet50(pretrained=True)
      # Freeze all layers initially
```

```
    for param in self.model.parameters():
        param.requires_grad = False

    # Unfreeze the last few layers
    for param in list(self.model.parameters())[-10:]:
        param.requires_grad = True

    # Replace the final layer
    self.model.fc = nn.Sequential(
        nn.Linear(self.model.fc.in_features, 512),
        nn.ReLU(),
        nn.Dropout(0.4),
        nn.Linear(512, self.num_classes)
    )

    self.model = self.model.to(device)
    logger.info("Model built successfully.")

def evaluate(self, val_loader):
    self.model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = self.model(images)
            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
    accuracy = correct / total
    logger.info(f"Validation Accuracy: {accuracy:.4f}")
    return correct / total

def load_model(self, path="temp/trained_model.pth"):
    #Check if the model exists locally; if not, download it from Azure
    if not os.path.exists(path) and self.azure_manager:
        print("Downloading model from Azure...")
        self.azure_manager.download_model_from_azure(path)

    try:
        print(f"Attempting to load model from {path}...")
        self.model = models.resnet50(pretrained=True)
        self.model.fc = nn.Sequential(
            nn.Linear(self.model.fc.in_features, 512),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(512, self.num_classes)
        )
        self.model.load_state_dict(torch.load(path))
        self.model = self.model.to(device)
        self.model.eval()

        logger.info(f"Model loaded successfully from {path}")
        print(f"Model loaded successfully from {path}")
    except Exception as e:
        logger.error(f"Failed to load model from {path}: {e}")
        print(f"Error loading model from {path}: {e}")
```

```
    def save_model(self, path):
        #Ensure the parent directory exists
        directory = os.path.dirname(path)
        if not os.path.exists(directory):
            os.makedirs(directory)

        try:
            #Save only the model's parameters (state dictionary)
            torch.save(self.model.state_dict(), path)
            logger.info(f"Model saved to {path}")
            print(f"Model saved successfully to {path}")

            #Upload to Azure if the AzureImageManager is available
            if self.azure_manager:
                self.azure_manager.save_model_to_azure(path)
        except Exception as e:
            logger.error(f"Failed to save model: {e}")
            print(f"Error saving model: {e}")

    # %% Train the model
    def train_model(self, train_loader, val_loader, epochs=epoch_num, learning_rate=LEARN_RATE):
        self.build_model()
        criterion = nn.CrossEntropyLoss(label_smoothing=0.1)  #Apply slight smoothing (adjusts
probabilities slightly to introduce uncertainty)
        optimizer = optim.SGD(self.model.fc.parameters(), lr=learning_rate, momentum=0.9)

        for epoch in range(epochs):
            self.model.train()
```

```
        running_loss = 0.0
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = self.model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        #Validation accuracy
        val_accuracy = self.evaluate(val_loader)
        print(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.4f}, Val Accuracy: {val_accuracy:.4f}")
        logger.info(f"Epoch {epoch+1}/{epochs}, Loss: {running_loss/len(train_loader):.4f}, Val Accuracy: {val_accuracy:.4f}")

    # %% Save the model to azure
    def save_model_to_azure(self, model_path, blob_name="trained_model.pth"):
        """Uploads the trained model to Azure."""
        try:
            blob_client = self.container_client.get_blob_client(blob_name)
            with open(model_path, "rb") as data:
                blob_client.upload_blob(data, overwrite=True)
            logger.info(f"Model saved to Azure as {blob_name}")
            print(f"Model saved to Azure as {blob_name}")
        except Exception as e:
            logger.error(f"Failed to save model to Azure: {e}")
            print(f"Failed to save model to Azure: {e}")
```

```
#Main application class

#CITATION: Graphical User Interface (GUI) with Tkinter
https://docs.python.org/3/library/tkinter.html

# %% SeedIdentifierApp Class

class SeedIdentifierApp:

  def __init__(self, root):

    self.root = root

    self.root.title("SeedID")

    self.root.geometry("800x600")

    self.azure_manager = AzureImageManager()


    # Initialize the model

    self.tm = TrainModel(azure_manager=self.azure_manager)


    # Alert bar at the top for displaying error or success messages

    self.alert_bar = tk.Label(

      self.root,

      text="",

      bg="yellow",

      fg="black",

      font=("Helvetica", 12, "bold"),

      anchor="center"

    )

    self.alert_bar.pack(side=tk.TOP, fill=tk.X, padx=0, pady=0)  # Attach at the top and stretch
horizontally

    self.alert_bar.pack_forget()  # Initially hide the alert bar


    #Style for the tabs
```

```
        style = ttk.Style()
        style.configure("TNotebook.Tab", font=("Helvetica", 13, "bold"), padding=[10, 5])  # Set font size and padding

        # Create sidebar frame for tabs
        self.sidebar = tk.Frame(self.root, width=200, bg="#ADD8E6")
        self.sidebar.pack(side=tk.LEFT, fill=tk.Y)

        # Create a Notebook (tabs) for the sidebar
        self.notebook = ttk.Notebook(self.sidebar, style="TNotebook")
        self.notebook.pack(fill=tk.BOTH, expand=True)

        # Create frames for each tab
        #CITATION: GeeksForGeeks. Creating Tabbed Widget With Python-Tkinter https://www.geeksforgeeks.org/creating-tabbed-widget-with-python-tkinter/
        self.main_tab = tk.Frame(self.notebook, bg="#ADD8E6")
        self.settings_tab = tk.Frame(self.notebook, bg="#ADD8E6")

        #Add tabs to the Notebook
        self.notebook.add(self.main_tab, text="Program")
        self.notebook.add(self.settings_tab, text="Settings")

        #Bind tab change event to handle result label visibility
        self.notebook.bind("<<NotebookTabChanged>>", self.on_tab_change)

        #Main content area frame
        self.main_area = tk.Frame(self.root)
        self.main_area.pack(side=tk.RIGHT, expand=True, fill=tk.BOTH)
```

```
    #Loading label for showing training in progress when model is being trained
    self.loading_label = tk.Label(self.main_area, text="", font=("Helvetica", 12))
    self.loading_label.pack(pady=10)
    self.home_info_label_main = tk.Label(
      self.main_area,
      text="",  # Initially empty
      font=("Helvetica", 12),
      justify="left",
      wraplength=600,
    )
    self.home_info_label_main.pack(pady=10, padx=10, fill=tk.BOTH)

    #MAIN PROGRAM TAB
    #Sidebar buttons
    self.upload_button = tk.Button(self.main_tab, text="Upload Image", font=("Helvetica", 13), command=self.upload_image)
    self.upload_button.pack(pady=10, padx=10, fill=tk.X)

    self.train_button = tk.Button(self.main_tab, text="Train Model", font=("Helvetica", 13), command=self.train_model_gui)
    self.train_button.pack(pady=10, padx=10, fill=tk.X)

    #SETTINGS TAB
    #Add the new "Home" button to the settings tab
    self.home_button = tk.Button(self.settings_tab, text="Home", font=("Helvetica", 13), command=self.display_home_info)
    self.home_button.pack(pady=10, padx=10, fill=tk.X)

    self.sync_button = tk.Button(self.settings_tab, text="Sync to Azure", font=("Helvetica", 13), command=self.sync_to_azure_threaded)
```

```
        self.sync_button.pack(pady=10, padx=10, fill=tk.X)


        self.show_azure_button = tk.Button(self.settings_tab, text="Show Azure Directory", font=("Helvetica", 13), command=self.show_azure_directory)
        self.show_azure_button.pack(pady=10, padx=10, fill=tk.X)


        self.clear_button = tk.Button(self.settings_tab, text="Clear Azure Database", font=("Helvetica", 13), command=self.clear_azure_database_threaded)
        self.clear_button.pack(pady=10, padx=10, fill=tk.X)


        #Image display label
        self.img_label = tk.Label(self.main_area)
        self.img_label.pack(pady=10)


        #Result display frame at the bottom of the main area
        self.result_frame = tk.Frame(self.main_area)
        self.result_frame.pack(side=tk.BOTTOM, pady=100)  # Anchor it to the bottom


        #Result label inside the result frame
        self.result_label = tk.Label(self.result_frame, text="Prediction results will appear here.", font=("Helvetica", 12, "bold"))
        self.result_label.pack()


        #Azure directory display frame within the main app
        #CITATION: PythonTutorial. Tkinter Scrollbar https://www.pythontutorial.net/tkinter/tkinter-scrollbar/
        self.azure_dir_frame = tk.Frame(self.main_area)
        self.azure_dir_frame.pack(fill=tk.BOTH, expand=True)
        self.azure_dir_canvas = tk.Canvas(self.azure_dir_frame)
```

```
        self.scrollbar = tk.Scrollbar(self.azure_dir_frame, orient="vertical",
command=self.azure_dir_canvas.yview)
        self.scrollable_frame = tk.Frame(self.azure_dir_canvas)

        self.scrollable_frame.bind(
            "<Configure>",
            lambda e: self.azure_dir_canvas.configure(scrollregion=self.azure_dir_canvas.bbox("all"))
        )

        self.azure_dir_canvas.create_window((0, 0), window=self.scrollable_frame, anchor="nw")
        self.azure_dir_canvas.configure(yscrollcommand=self.scrollbar.set)

        self.azure_dir_canvas.pack(side="left", fill="both", expand=True)
        self.scrollbar.pack(side="right", fill="y")

        #Load the trained model if available
        try:
            self.tm.load_model('model/trained_model.pth')
            print("Model loaded successfully.")
        except Exception as e:
            print("Model not found or failed to load.")
            logger.warning(f"Model not found or failed to load: {e}")
            self.tm.model = None  # Explicitly set to None if loading fails

    def on_tab_change(self, event):
        #Check which tab is selected
        selected_tab = self.notebook.index(self.notebook.select())

        if selected_tab == 0:  #Program tab
```

```
            #Show the prediction result label text
            self.result_label.config(text="Prediction results will appear here.")

            #Show the image label if it's hidden
            self.img_label.pack_forget()

            #Hide the Azure directory frame
            self.azure_dir_frame.pack_forget()

        elif selected_tab == 1:  #Settings tab
            #Hide the prediction result label text
            self.result_label.config(text="")

            #Hide the image label when switching to settings
            self.img_label.pack_forget()

            #Show the Azure directory frame
            self.azure_dir_frame.pack(fill=tk.BOTH, expand=True)
            self.display_home_info()

    #Display descriptions of the settings buttons.
    def display_home_info(self):
        self.azure_dir_frame.pack_forget()
        descriptions = (
            "Descriptions of Settings Options:\n\n"
            "1. **Sync to Azure**: \nUpload all local seed images to Azure Blob Storage.\n\n"
            "2. **Show Azure Directory**: \nDisplay the list of images stored in Azure, organized by
classes.\n\n"
```

```
        "3. **Clear Azure Database**: \nDelete all images from Azure Blob Storage for a fresh start.\n\n"
    )
    self.home_info_label_main.config(text=descriptions)
    self.home_info_label_main.pack()  #Show the label

  def show_azure_directory(self):
    #Hide the home description label
    self.home_info_label_main.pack_forget()

    #Ensure the Azure directory frame is visible
    self.azure_dir_frame.pack(fill=tk.BOTH, expand=True)

    #Clear any existing content in the scrollable frame
    for widget in self.scrollable_frame.winfo_children():
      widget.destroy()

    #Fetch the list of images organized by class from Azure
    class_images = self.azure_manager.list_images_by_class(class_names)

    #Display class names and their images in the scrollable frame
    for class_name, images in class_images.items():
      #Class Label
      tk.Label(self.scrollable_frame, text=class_name, font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=5)

      #List of images in this class
      for image_name in images:
        tk.Label(self.scrollable_frame, text=image_name, font=("Helvetica", 10)).pack(anchor="w", padx=20)
```

```
def show_alert(self, message):
    self.alert_bar.config(text=message, anchor="center")
    #Repack to ensure alignment
    self.alert_bar.pack(side=tk.TOP, fill=tk.X, padx=0, pady=0)

def hide_alert(self):
    self.alert_bar.pack_forget()

# %% Upload Image
def upload_image(self):
    if not self.tm.model:
        self.show_alert("No model found. Please train the model first.")
        return

    file_path = filedialog.askopenfilename(
        title="Select an Image",
        filetypes=(("Image Files", "*.jpg;*.jpeg;*.png"), ("All Files", "*.*"))
    )
    if file_path:
        logger.info(f"Image uploaded: {file_path}")
        img = Image.open(file_path)
        img.thumbnail((200, 200))
        img_tk = ImageTk.PhotoImage(img)
        self.img_label.pack(pady=10)
        self.img_label.config(image=img_tk)
        self.img_label.image = img_tk
        self.predict_image(file_path)
```

```
    # %% Predict Image
    #CITATION: GeeksForGeeks. Loading Images in Tkinter using PIL https://www.geeksforgeeks.org/loading-images-in-tkinter-using-pil/
    def predict_image(self, file_path):
        self.hide_alert()  #Hide any previous alerts
        if not self.tm.model:
            self.show_alert("Model not loaded. Please train the model first.")
            logger.warning("Attempted prediction without loaded model.")
            return
        img = Image.open(file_path).convert('RGB')
        transform = transforms.Compose([
            transforms.Resize((256, 256)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])

        img_tensor = transform(img).unsqueeze(0)  # Add batch dimension

        #Move the tensor to the same device as the model and ensure the input tensor is on the same device
        img_tensor = img_tensor.to(device)
        self.tm.model.to(device)

        try:
            self.tm.model.eval()  #Set the model to evaluation mode
            with torch.no_grad():  #Disable gradient calculation for inference
                output = self.tm.model(img_tensor)  #Forward pass
                _, pred_class = torch.max(output, 1)  #Get the predicted class
                confidence = (torch.softmax(output, dim=1)[0][pred_class].item()) * 100  #Get confidence
```

```
            self.display_results(pred_class.item(), confidence)
            logger.info(f"Prediction: {pred_class}, Confidence: {confidence:0f}%")

    except Exception as e:
        self.show_alert(f"Prediction failed: {e}")
        logger.error(f"Prediction failed: {e}")

# %% Display the results
def display_results(self, pred_class, confidence):
    if confidence < 85: #If the confidence score is less than 85% show as "no results found"
        result = "No results found"
    else:
        result = f"Prediction: {class_names[pred_class]}\nConfidence: {confidence:0f}%"

    self.result_label.config(text=result)

#Show a loading message.
def show_loading(self, message):
    self.loading_label.config(text=message)
    self.loading_label.pack()
    self.root.update_idletasks()

#Hide the loading message.
def hide_loading(self):
    self.loading_label.config(text="")
    self.loading_label.pack_forget()
    self.root.update_idletasks()

def train_model_gui(self):
```

```
    #BUG PATCH: Prevent duplicate model training
    if hasattr(self, 'training_thread') and self.training_thread.is_alive():
        messagebox.showinfo("Training in Progress", "Model is already training. Please wait.")
        return

    #Hide any previous alerts
    self.hide_alert()
    self.show_loading("Training... Please wait.")
    threading.Thread(target=self.run_training_process).start()
    #DL all images from azure to train the model.
    temp_training_dir = './temp_azure_images'
    self.azure_manager.download_all_images(temp_training_dir)



    self.hide_loading()

    messagebox.showinfo("Training", "Model trained and saved successfully.")
    logger.info(f"Model training completed and saved")

    # Clean up the temporary directory after training
    shutil.rmtree(temp_training_dir)

def run_training_process(self):
    temp_training_dir = './temp_azure_images'
    self.azure_manager.download_all_images(temp_training_dir)A

    train_loader, val_loader, class_names = self.tm.load_data(temp_training_dir)

    #Run training; ensure training completes before moving forward
```

```
        self.tm.train_model(train_loader, val_loader)

        #Save the trained model (if necessary)
        #Only after training completes, hide loading and show the completion message
        self.hide_loading()
        messagebox.showinfo("Training", "Model trained and saved successfully.")
        logger.info("Model training completed and saved")

        #Clean up the temporary directory after training
        #CITATION: Delete an entire directory tree using Python | shutil.rmtree() method https://www.geeksforgeeks.org/delete-an-entire-directory-tree-using-python-shutil-rmtree-method/
        shutil.rmtree(temp_training_dir)

    #Start a thread to sync to Azure
    def sync_to_azure_threaded(self):
        threading.Thread(target=self.sync_to_azure).start()

    #CITATION: Microsoft. Upload a block blob with Python https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blob-upload-python
    def sync_to_azure(self):
        #Syncs the local images to Azure
        print("Starting sync to Azure...")
        self.show_loading("Syncing to Azure... please wait.")

        #Clear all existing blobs in the container first
        print("Clearing Azure container...")
        self.azure_manager.clear_container()

        #Define the local folder containing images organized by class
```

```
    local_directory = './temp_images'

    #Walk through all files and folders in the directory
    for root, dirs, files in os.walk(local_directory):
        for file_name in files:
            file_path = os.path.join(root, file_name)
            blob_name = os.path.relpath(file_path, local_directory)
            print(f"Uploading {file_path} as {blob_name}")

            try:
                # Upload file with folder structure
                self.azure_manager.upload_image_with_path(file_path, blob_name)
                print(f"Uploaded {file_name} to Azure.")
            except Exception as e:
                print(f"Failed to upload {file_path} to Azure: {e}")

    print("Sync to Azure completed.")
    self.show_alert("Sync to Azure completed successfully.")
    self.hide_loading()

#Start a thread to clear the Azure database
def clear_azure_database_threaded(self):
    threading.Thread(target=self.clear_azure_database).start()

def clear_azure_database(self):
    #Clear all blobs in Azure container.
    self.show_loading("Clearing Azure database... please wait.")

    try:
```

```
        #Call the clear_container method to delete all blobs in Azure
        self.azure_manager.clear_container()
        print("All images have been cleared from the Azure container.")
        messagebox.showinfo("Azure Database", "All images have been cleared from the Azure database.")

        for widget in self.scrollable_frame.winfo_children():
            widget.destroy()
        tk.Label(self.scrollable_frame, text="Azure directory is now empty.", font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=10)

    except Exception as e:
        print(f"Failed to clear Azure database: {e}")
        messagebox.showerror("Error", f"Failed to clear Azure database: {e}")

    finally:
        #Hide the loading message
        self.hide_loading()

  def clear_azure_database(self):
    #Call the clear_container method to delete all blobs in Azure
    self.azure_manager.clear_container()
    print("All images have been cleared from the Azure container.")
    messagebox.showinfo("Azure Database", "All images have been cleared from the Azure database.")

    for widget in self.scrollable_frame.winfo_children():
        widget.destroy()
    tk.Label(self.scrollable_frame, text="Azure directory is now empty.", font=("Helvetica", 12, "bold")).pack(anchor="w", padx=10, pady=10)
```

```
# %% Run the application
if __name__ == "__main__":
    root = tk.Tk()
    app = SeedIdentifierApp(root)
    root.mainloop()
```

LOGGER CLASS FILE

```
import pandas as pd
import logging
from datetime import datetime
import os

#CITATION: GeeksforGeeks. Log File to Pandas Dataframe. https://www.geeksforgeeks.org/log-file-to-pandas-dataframe/
class CSVLogHandler(logging.Handler):
    def __init__(self, log_file="application_logs.csv"):
        super().__init__()
        self.log_file = log_file
        self._initialize_log_file()

    def _initialize_log_file(self):
        if not os.path.exists(self.log_file):
            pd.DataFrame(columns=["Timestamp", "Level", "Message"]).to_csv(self.log_file, index=False)

    def log_to_dataframe(self, level, message):
        log_entry = {
            "Timestamp": datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
```

```
        "Level": level,
        "Message": message
    }
    df = pd.DataFrame([log_entry])
    df.to_csv(self.log_file, mode='a', index=False, header=False)

def emit(self, record):
    self.log_to_dataframe(record.levelname, record.getMessage())
```