Recycle Sort: Image Classifier for Recycling Identification

CPE 462: Introduction to Image Processing and Coding

Arden Sentak

I pledge my honor that I have abided by the Stevens Honor System.

Abstract

Within the area of image processing and coding, this project aims to explore the topic of image classification. Most people know to recycle items like bottles and cans, but beyond these basics, there's a lack of awareness about what can and cannot be recycled. The goal of this project was to design a binary image classifier that can help people to determine whether or not an item is recyclable. This project was implemented in Python using TensorFlow and Keras to develop an image classification model, which was trained on a dataset consisting of 12,615 images. Then, a prediction script was written in Python in order to use the model. This script accepts user-uploaded images and returns a recyclability prediction for the item in the image based on the trained model. To provide an accessible user experience, a web interface was designed using Next.js framework with TypeScript in order to allow users to interact with the classification model. The model achieved an accuracy of 93% on the training set and an accuracy of 82% on the validation set, demonstrating its ability to classify items accurately. Therefore, this project resulted in a functional, working tool being developed to predict whether or not an item is recyclable.

Description of Work Accomplished

1. Dataset Preparation

To develop a binary image classifier capable of distinguishing between recyclable and non-recyclable items, a custom dataset was assembled through web scraping. All images were collected from publicly available datasets on Kaggle or from Google Images. The dataset was split into two sections: training data and validation data. Within both of these sections, the images were split into two folders: recyclable and non-recyclable. The training data was made to be larger than the validation data so that the image classifier had the best chance to learn how to distinguish recyclable and non-recyclable items. The final data set consisted of 12,615 images with the following distribution:

Section	Non-Recyclable (# of Images)	Recyclable (# of Images)
Training Data	4,667	3,593
Validation Data	2,284	2,071

2. Image Preprocessing + Data Cleaning (training.py)

In order to produce a reliable and accurate image classifier, the dataset needed to be preprocessed before being fed into the classification model. In order to preprocess the data, several image processing techniques were used. The function I created to preprocess the dataset is pictured below.

```
#function to preprocess data using image processing techniques
def preprocess_image(img):
    img = tf.image.resize(img, [224, 224])  #resizes to 224 x 224
    img = img / 255.0 #normalizes between 0 and 1
    img = (img - 0.5) * 2.0 #scales from -1 to 1
    img = tf.image.random_brightness(img, max_delta=0.1) #randomly adjusts image brightness
    img = tf.image.random_contrast(img, lower=0.9, upper=1.1) # randomly adjusts image contrast
    img = tf.image.random_flip_left_right(img) #randomly rotates (flips) images
    return img
```

This function takes in an image and utilizes TensorFlow to perform image processing. First, the image is resized to be 224 x 224. It normalizes the image pixel values between 0 and 1, and then scales them to range from -1 to 1. To add variation into the dataset, this function randomly adjusts the brightness, contrast, and orientation of the image. Before creating the image classifier, it was necessary to ensure that all files in the dataset were usable. Therefore, I created two functions to clean the dataset.

```python
#creates a directory for corrupted files
corrupted_files_dir = "corruptedFiles"
os.makedirs(corrupted_files_dir, exist_ok=True)

# Function to move corrupted files out of the dataset
def move_to_corrupted_files(img_path):
    try:
        file_name = os.path.basename(img_path)
        destination = os.path.join(corrupted_files_dir, file_name)
        shutil.move(img_path, destination)
        print(f"Moved corrupted image {file_name} to {corrupted_files_dir}")
    except Exception as e:
        print(f"Error moving corrupted file {img_path}: {e}")

# Function to clean the dataset of invalid image files (.jpg, .jpeg, .png are acceptable)
def clean_directory(directory):
    valid_extensions = (".jpg", ".jpeg", ".png")
    for root, _, files in os.walk(directory):
        for file in files:
            if file.lower().endswith(valid_extensions):
                file_path = os.path.join(root, file)
                try:
                    with Image.open(file_path) as img:
                        img.verify()  # Verify the file integrity
                    with Image.open(file_path) as img:
                        img.convert("RGB")  # Try to actually open and convert image
                except Exception as e:
                    print(f"Corrupted or unsupported file: {file_path} - {e}")
                    move_to_corrupted_files(file_path)
```

Clean_directory first checks to see if each file in a directory has an acceptable file extension. For my image classifier I accepted the following extensions: jpg, jpeg, and png. This function also does a quick integrity check on each file by using .verify() from the PIL library, and then attempts to convert the image to RGB for a deeper check. If any of these checks are failed, the function prints a message indicating that a file is corrupt and then calls move_to_corrupted files. The function, move_to_corrupted_files, uses shutil to move a file out of its current path and into the corrupted files directory. If the file is successfully moved a message will indicate so. However, if the file is not moved an error message will be printed out. The error message is

printed so that if a file could not be moved, I would be able to easily locate the file and manually remove it.

3. CNN-Based Image Classifier: Development and Training (training.py)

To prepare the dataset to be fed into an image classification model I created two iterators – one for the training data (train_gen) and one for the validation data (val_gen).

```
#initializing image size to 224 x 224 and batch size to 32
IMAGE_SIZE = (224, 224)
BATCH_SIZE = 32

#iterator for training dataset --> used to load images in batches instead of all at once
train_gen = train_datagen.flow_from_directory(
    train_dir, #path to directory
    target_size=IMAGE_SIZE, #size of image (224 x 224)
    batch_size=BATCH_SIZE, #number of images in the batch (32)
    class_mode='binary' #binary since dataset has two classes: recyclable and non_recyclable
)

#iterator for validation dataset
val_gen = val_datagen.flow_from_directory(
    val_dir,
    target_size=IMAGE_SIZE,
    batch_size=BATCH_SIZE,
    class_mode='binary'
)
```

Essentially, these iterators will tell the classification model how to process through the dataset. Each iterator specifies the path to the directory that holds the desired data. For example, train_dir was previously initialized in the code to be *train_dir = "data/trainingData"* since the training data iterator needs to access the training data. In addition, each iterator specifies the target image size, the number of images that should be contained in each batch, and the class mode. The target image size I used is 224 x 224 and the batch size I used is 32. The class mode is set to binary since the training data and validation data only have two classes: recyclable and non_recyclable.

After the iterators were created, I set up the structure of my image classification model by using commands from the TensorFlow Keras library. Since my dataset was relatively small for deep learning, I used the concept of transfer learning to achieve higher accuracy. Transfer learning is a technique in which a model that has been previously trained for another task is used as a starting point for building another model with a semi-related task.

```
#loading in a pretrained model for transfer learning: enhances feature learning
    #MobileNetV2 is a CNN used for extracting image features (edges, textures, objects...)
    #imagenet is a super large dataset
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(224, 224, 3))
base_model.trainable = False  # Freeze base model initially so that these layers dont change
```

I used MobileNetV2 as the base of my model. MobileNetV2 is a pre-trained convolutional neural network that was trained for computer vision tasks such as extracting image features like edges, textures, and objects. Note that I did not create MobileNetV2, it is a pre-trained publicly available model that I accessed through the TensorFlow Keras library. I used the following import statement in my code to access it: *"from keras._tf_keras.keras.applications import MobileNetV2."* The weights parameter for the base of my model was set to "imagenet," which is a large, publicly available image dataset. Setting the weights parameter to this dataset allowed my classifier to start with pre-learned patterns obtained from millions of labeled images in the ImageNet dataset. Furthermore, the include_top parameter was set to false so that the final classification layers of MobileNetV2 were removed so that I could add my own custom layers. Then, the input_shape parameter was set to (224, 224, 3) so that the images inputted into the classifier would be 224 x 224 and RGB, which matched my preprocessing function. Lastly, I froze the base model by setting base_model.trainable to false so that any pretrained features would not be initially overwritten.

After my base model was configured, I designed my own custom layers to add on top of it.

```
#custom layers
x = base_model.output #stores last layer of frozen base model so customization can go on top
x = GlobalAveragePooling2D()(x) #pooling layer --> averages feature maps to a single number
x = Dropout(0.2)(x) #dropout layer --> randomly drops 20% of units to help the model generalize better
x = Dense(64, activation='relu')(x) #fully connected layer w/ 64 neurons; relu used so model can learn recyclability features
output = Dense(1, activation='sigmoid')(x) #will output the probability an image is recyclable (recyclable = 1)

#combines pre-trained model and my output layers
model = Model(inputs=base_model.input, outputs=output)

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

The variable x was created to store the last layer of the frozen base model so that I could add my customization layers directly on top. I added the following custom layers: a global pooling layer, a dropout layer, and two dense layers. The global pooling layer averaged feature maps to a single

number in order to reduce overfitting and size. The dropout layer randomly drops 20% of units during training to help the model generalize better. The first dense layer is a fully connected layer with 64 neurons and ReLU activation (a commonly used activation function for deep learning) to help the model learn patterns regarding the recyclability of an item. Lastly, the output layer is a dense layer with 1 output and sigmoid activation which is used to output a binary probability with recyclable = 1 and non-recyclable = 0. After each layer was designed, they were combined into one model with the base model being the input (bottom layers) and my custom layers being the output (top layers).

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()

#trains only on my custom layers
history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=5  #goes through the dataset 5 times
)

# Unfreezes the last 30 layers of MobileNetV2 for fine tuning
for layer in base_model.layers[-30:]:
    layer.trainable = True

model.compile(optimizer=tf.keras.optimizers.Adam(1e-5),  # lower Learing Rate for fine tuning
              loss='binary_crossentropy',
              metrics=['accuracy'])

#trains again
fine_tune_history = model.fit(
    train_gen,
    validation_data=val_gen,
    epochs=7  #goes through the dataset 7 times
)

#save trained model
model.save("models/recyclableClassifierFinalModel.keras")
```

The model was then compiled to be prepared for training. I compiled it with an Adam optimizer, a binary cross entropy loss function, and an accuracy metric so that I could measure my results. After compiling, I trained my model on only my custom layers for 5 epochs, or 5 complete pass throughs of the data. After the initial training, to boost accuracy and performance of my model I unfroze the last 30 layers of my base model and enabled these layers to be trainable. I recompiled my model to have a lower learning rate since I was just fine tuning rather than training from scratch. Then, I trained the model again for 7 epochs. After doing this, the model was done training and was saved so that I could access it.

4. Prediction Algorithm (classifier.py)

After the image classification model was trained, a prediction algorithm was created so that the model could be utilized. To write my prediction algorithm I loaded my trained model into a variable called model and then wrote a function called classify_image. Within classify_image I used commands from the TensorFlow Keras library and from NumPy to simplify my code.

```
model = tf.keras.models.load_model("models/recyclableClassifierFinalModel.keras")

#prediction script
def classify_image(img_path):
    """Loads an image and predicts if it's recyclable or not."""
    img = image.load_img(img_path, target_size=(224, 224))
    img_array = image.img_to_array(img) / 255.0
    img_array = np.expand_dims(img_array, axis=0)

    prediction = model.predict(img_array)[0][0]
    print(prediction)
    print(model.predict(img_array))
    return "Recyclable" if prediction > 0.5 else "Non-Recyclable"
```

Classify_image takes in the path of an image as a parameter. Based on this path, it loads the image and ensures that it is of the proper size (224 x 224). Then, it converts the image to an array of pixels and divides by 255 to normalize the pixels to have values between 0 and 1. Lastly, it expands the array shape to be (1, 224, 224, 3) rather than (224, 244, 3) by adding a new axis at position 0. This expansion was needed so that the array represented the batch size (1 image), as well as the shape (224 x 224) and color (RGB) of the image. After the image array was set up, it was fed into the model as a scalar value so that the model could predict the probability of the image being recyclable or non-recyclable. After .predict() is used to predict the probability that an item is recyclable or not, the prediction results are printed out. The prediction probability is printed out as well as the actual prediction (recyclable or non-recyclable). As mentioned earlier, my model is a binary image classifier in which values closer to 1 are associated with recyclable items and values closer to 0 are associated with non-recyclable items. Therefore, this algorithm predicts an image to be recyclable if its prediction probability is greater than 0.5 or non-recyclable if its prediction probability is less than 0.5.

5. Web Interface

A web interface was designed in order to make my image classifier usable. I developed a FastAPI backend using Python and a frontend using Next.js framework with TypeScript. I created the Next.js framework by typing the following command into the terminal: "*npx create-next-app@latest --ts"*. Note that when a Next.js application is created, the computer automatically sets up the basic project structure and the necessary foundational files. Therefore, I did not write many of the files for my web interface and will only be discussing any files I changed or created in order to customize my application.

Backend

In order to develop my backend I used the following dependencies: TensorFlow, FastAPI, Uvicorn, and Pillow. I created three files: main.py, classifier.py, and predict.py. The main file initializes and runs my backend API.

```python
from app.api import predict  # Importing my predict router

app = FastAPI(
    title="Image Classification API",
    description="API to classify uploaded images using a trained ML model.",
    version="1.0.0"
)

# Allow CORS (important for frontend to connect during dev)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # In production, specify allowed domains
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Register routers
app.include_router(predict.router, prefix="/api")
```

To accomplish this my predict router, which is in the predict file, was imported. Then, a FastAPI application was initialized with basic labelling info. I added CORS middleware to the application so that the frontend would be able to make requests to the backend. Lastly, I registered the predict router under the /api path for organization purposes.

The predict file handles image uploads and initializes the predict router used in the main file.

```python
router = APIRouter()

@router.post("/predict")
async def predict(file: UploadFile = File(...)):
    try:
        with tempfile.NamedTemporaryFile(delete=False, suffix=".jpg") as tmp:
            tmp.write(await file.read())
            tmp_path = tmp.name

        result = classify_image(tmp_path)
        return {"prediction": result}

    except Exception as e:
        print("Backend Error:", e)
        return {"error": str(e)}

    finally:
        if os.path.exists(tmp_path):
            os.remove(tmp_path)
```

In this file, a router object is created to define API routes. Then, to initialize the predict router I registered a POST route at /predict. Then I created a predict function that takes in an uploaded image and feeds it to the image classifier so that a recycling prediction can be made. The predict function takes in an uploaded file and tries to temporarily save it as a jpg file. If it can successfully do so, my prediction algorithm (classify_image) is called so that my image classifier classifies the uploaded image as recyclable or non-recyclable. The result is returned and then the temporary file is deleted. Note that if the uploaded file can't be saved as a jpg an error exception will be thrown.

Lastly, the classifier file contains the logic needed to predict whether an item is recyclable or not. This logic was previously discussed in section 4: *Prediction Algorithm*.

Frontend

In order to develop the frontend a good amount of dependencies were needed. These were included in a package.json file and are shown below.

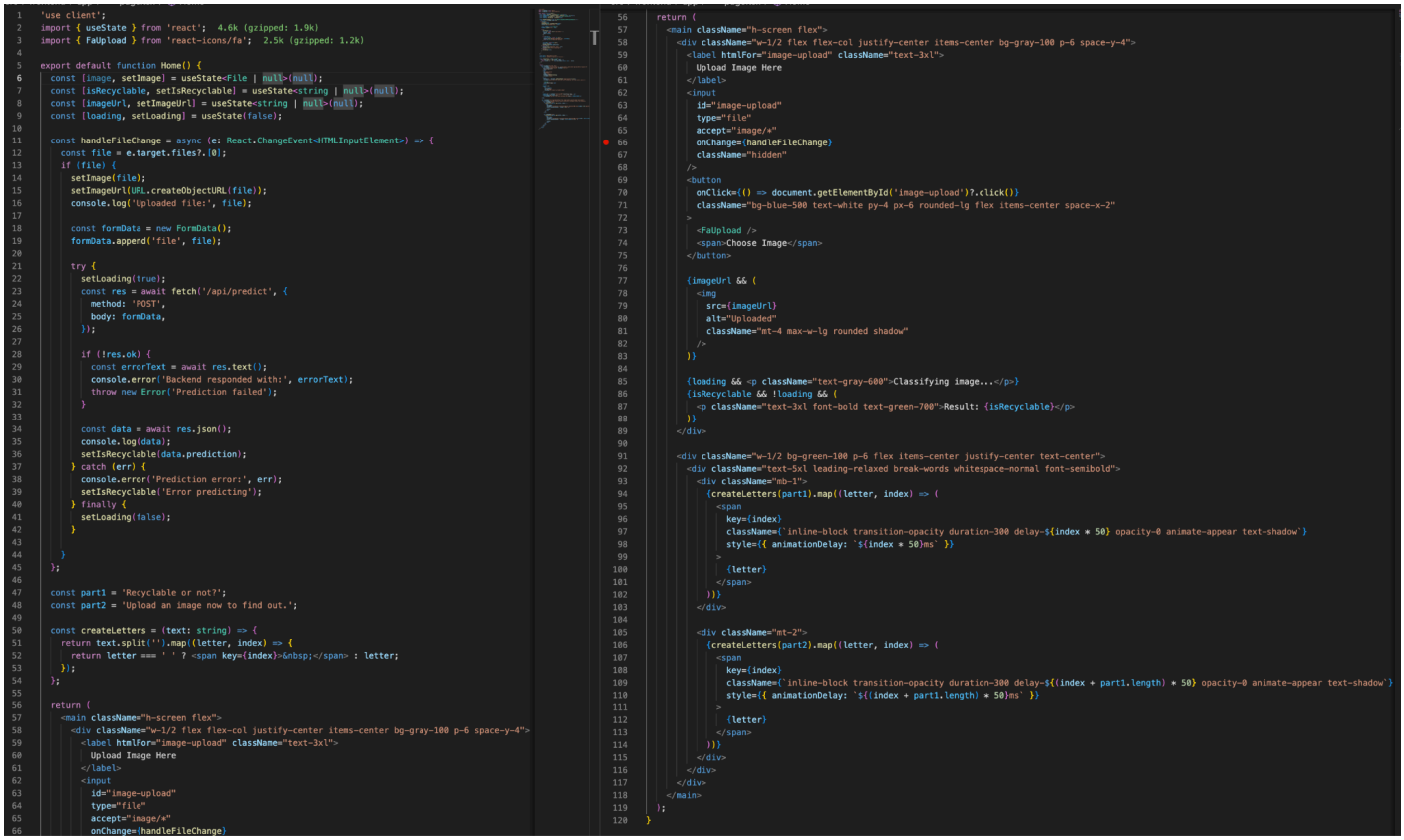
```
  "dependencies": {
    "@radix-ui/react-toolbar": "^1.1.3",
    "next": "^15.3.0",
    "react": "^19.1.0",
    "react-dom": "^19.1.0",
    "react-icons": "^4.3.1"
  },
  "devDependencies": {
    "@eslint/eslintrc": "^3",
    "@tailwindcss/postcss": "^4",
    "@types/node": "^20",
    "@types/react": "^19",
    "@types/react-dom": "^19",
    "eslint": "^9",
    "eslint-config-next": "15.3.0",
    "tailwindcss": "^4",
    "typescript": "^5"
  }
}
```

Since I developed my frontend by creating a Next.js application, as previously mentioned, many default files were created for me. The files I created or edited are as follows: layout.tsx, page.tsx, header.tsx, and footer.tsx. The layout file specifies my website layout. I made a couple changes to this file in order to customize it for my website.

```
import type { Metadata } from "next";
import { Inter } from "next/font/google";  643 (gzipped: 377)
import "./globals.css";
import Header from "../components/ui/header";
import Footer from "../components/ui/footer";

const inter = Inter({subsets: ["latin"]});

export const metadata: Metadata = {
  title: "Image Classifier",
  description: "Recyclable or Not?",
};


export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body
        className={inter.className}
      >
        <Header/>
        {children}
        <Footer/>
      </body>
    </html>
  );
}
```

First, I edited the title and description to fit my website. These represent the title and description that show up in a browser tab. Then, I added the line *className = {inter.className}* so that I could use the font, Inter, on my website. Lastly, I changed the website structure to have a header before my webpage and a footer at the bottom of my webpage.

The page file, which I created, defines my webpage. Essentially, my website allows users to upload an image and the website will return a prediction stating whether or not the item in the image is recyclable or not.



```
'use client';
import { useState } from 'react';  4.6k (gzipped: 1.9k)
import { FaUpload } from 'react-icons/fa';  2.5k (gzipped: 1.2k)

export default function Home() {
  const [image, setImage] = useState<File | null>(null);
  const [isRecyclable, setIsRecyclable] = useState<string | null>(null);
  const [imageUrl, setImageUrl] = useState<string | null>(null);
  const [loading, setLoading] = useState(false);

  const handleFileChange = async (e: React.ChangeEvent<HTMLInputElement>) => {
    const file = e.target.files?.[0];
    if (file) {
      setImage(file);
      setImageUrl(URL.createObjectURL(file));
      console.log('Uploaded file:', file);

      const formData = new FormData();
      formData.append('file', file);

      try {
        setLoading(true);
        const res = await fetch('/api/predict', {
          method: 'POST',
          body: formData,
        });

        if (!res.ok) {
          const errorText = await res.text();
          console.error('Backend responded with:', errorText);
          throw new Error('Prediction failed');
        }

        const data = await res.json();
        console.log(data);
        setIsRecyclable(data.prediction);
      } catch (err) {
        console.error('Prediction error:', err);
        setIsRecyclable('Error predicting');
      } finally {
        setLoading(false);
      }

    }
  };

  const part1 = 'Recyclable or not?';
  const part2 = 'Upload an image now to find out.';

  const createLetters = (text: string) => {
    return text.split('').map((letter, index) => {
      return letter === ' ' ? <span key={index}> </span> : letter;
    });
  };

  return (
    <main className="h-screen flex">
      <div className="w-1/2 flex flex-col justify-center items-center bg-gray-100 p-6 space-y-4">
        <label htmlFor="image-upload" className="text-3xl">
          Upload Image Here
        </label>
        <input
          id="image-upload"
          type="file"
          accept="image/*"
          onChange={handleFileChange}
          className="hidden"
        />
        <button
          onClick={() => document.getElementById('image-upload')?.click()}
          className="bg-blue-500 text-white py-4 px-6 rounded-lg flex items-center space-x-2"
        >
          <FaUpload />
          <span>Choose Image</span>
        </button>

        {imageUrl && (
          <img
            src={imageUrl}
            alt="Uploaded"
            className="mt-4 max-w-lg rounded shadow"
          />
        )}

        {loading && <p className="text-gray-600">Classifying image...</p>}
        {isRecyclable && !loading && (
          <p className="text-3xl font-bold text-green-700">Result: {isRecyclable}</p>
        )}
      </div>

      <div className="w-1/2 bg-green-100 p-6 flex items-center justify-center text-center">
        <div className="text-5xl leading-relaxed break-words whitespace-normal font-semibold">
          <div>
            {createLetters(part1).map((letter, index) => (
              <span
                key={index}
                className={`inline-block transition-opacity duration-300 delay-${index * 50} opacity-0 animate-appear text-shadow`}
                style={{ animationDelay: `${index * 50}ms` }}
              >
                {letter}
              </span>
            ))}
          </div>

          <div>
            {createLetters(part2).map((letter, index) => (
              <span
                key={index}
                className={`inline-block transition-opacity duration-300 delay-${(index + part1.length) * 50} opacity-0 animate-appear text-shadow`}
                style={{ animationDelay: `${(index + part1.length) * 50}ms` }}
              >
                {letter}
              </span>
            ))}
          </div>
        </div>
      </div>
    </main>
  );
}
```

Since my website is only one page all logic is contained in the Home function. First I initialize state variables that will be used throughout the file. These variables will store the uploaded file, the prediction result, the temporary image URL, and whether or not a prediction is loading. The function, handleFileChange, is triggered when the user selects an image to upload. When an image is uploaded, the function stores it and its URL. It then creates a FormData object so that the file matches the form submission type that the backend was set up to accept. After that, the

function tries to send the image file in a POST request to my backend predict router. It does this so that the recyclability of the image can be predicted. Before awaiting a response, the function checks if there are any HTTP errors. If there are errors, the website will let the user know that the prediction has failed. If there are no errors, the function will wait for the backend to make a prediction and then store the prediction result. The prediction result should be recyclable or non-recyclable but it could be "error predicting" if some other error occurs during the prediction process. Lastly, the function sets the loading variable to false so that the website knows a prediction is no longer loading. The function, createLetters, was created to style my website with a pop up animation. When the user first goes onto the website, the letters from the variables part1 and part2 will pop up one at a time to form the phrases. The rest of the code in the page file defines the structure of my webpage. My webpage is designed to have two side by side columns. The first container, indicated by "* ,"* represents the left half of my site. The left half has a gray background and its contents are centered. A text prompt saying "Upload Image Here" appears as well as a blue button that allows the user to choose an image to upload. When the button is clicked, the user's file manager will pop up so that they can select a file to upload. Once they select a file, their image will be uploaded onto the website underneath the choose file button as well as text that says "classifying image." Once the image classifier is done predicting the recyclability of the image, the "classify image" text will change to say "Result: Recyclable" or "Result: Non-Recyclable" depending on the prediction result from my image classifier. The second container, indicated by "* ,"* represents the right half of my site. The right half is green and its contents are centered. Text saying "Recyclable or not?" and "Upload an image now to find out." appears on this side. When the user first goes onto the site these phrases are animated to pop up letter by letter, but after that the text just stays stationary.

The header file creates a custom header for my website.

```
import React from 'react'  7.9k (gzipped: 3.1k)
import { FaRecycle } from 'react-icons/fa'  3.1k (gzipped: 1.6k)
import * as Toolbar from '@radix-ui/react-toolbar'  14.3k (gzipped: 5.1k)

const Header = () => {
  return (
    <Toolbar.Root className='w-full border-b-3 border-green-700 p-2 flex items-center relative'>
      <div className='absolute left-6 flex items-center space-x-2'>
        <FaRecycle className='text-green-600 text-4xl' />
      </div>

      <div className='flex-grow text-center'>
        <h1 className='text-4xl text-green-600'>Recycle Sort</h1>
      </div>
    </Toolbar.Root>
  )
}

export default Header
```

In the Header component is where I customized a website header. My header has full-screen width and is white with a small green border at the bottom. On the left of the header is a green recycling icon imported from react-icons/fa. In the center of the header is green text that says “Recycle Sort” which represents the name of my project. The line *export default Header* is written at the bottom of my file so that I could use my header in other files.

The footer file creates a custom footer for my website.

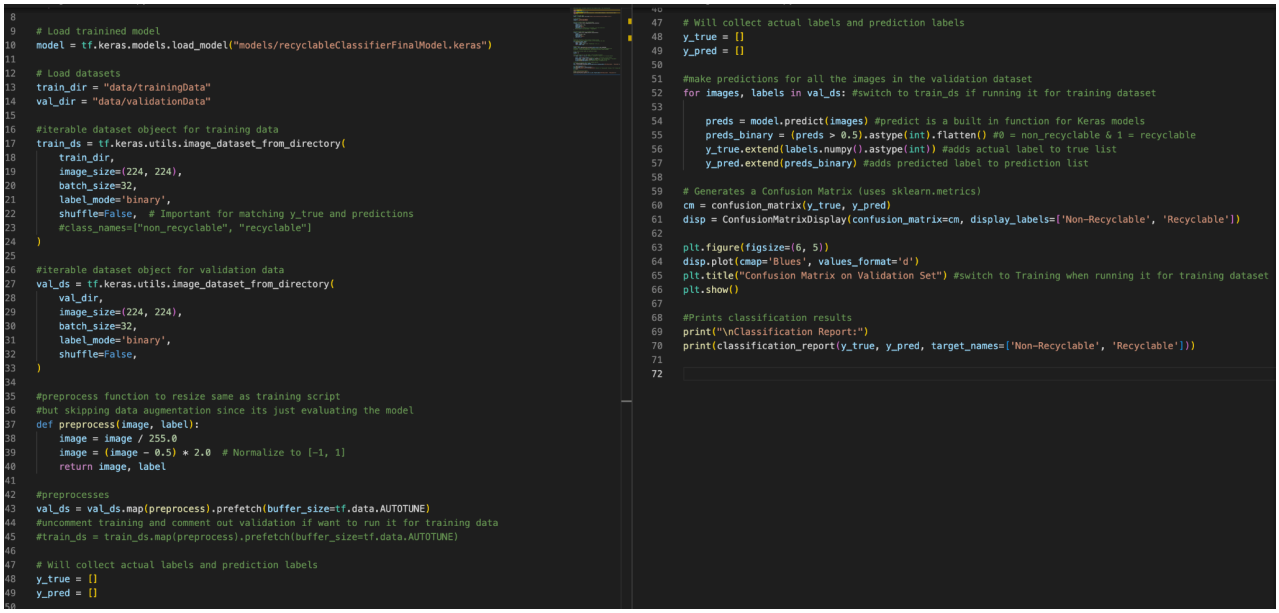
```
import React from 'react';  7.9k (gzipped: 3.1k)

const Footer = () => {
  return (
    <footer className="text-center text-sm text-gray-500 py-2 mt-2">
      <p>&copy; {new Date().getFullYear()} Arden Sentak</p>
    </footer>
  );
};

export default Footer;
```

In the Footer component is where I customized my website footer. My footer consists of small centered gray text that reads “© 2025 Arden Sentak.” The line *export default Footer* is written at the bottom of my file so that I could use my footer in other files.

Results

After I finished training my model (by running training.py) the final accuracy for the training and validation datasets were printed out. The training set achieved 93% accuracy with a loss of 0.17, while the validation set achieved 82% accuracy with a loss of 0.44. The drop in accuracy and increase in loss from training to validation suggest that the model is somewhat overfitting to the training data — it performs better on data it has seen before than on new, unseen data. This is expected to some degree, especially since the model was trained on a relatively small dataset. It's also worth noting that the model was trained to classify typical household items as recyclable or non-recyclable. Therefore, in cases where images are non-household items, such as buildings or cars, the model may struggle, contributing to the higher validation loss. Despite this, the model demonstrates solid performance for its intended use. The accuracy and loss levels are acceptable for a household item classifier and indicate that the model can correctly predict the recyclability of common household items the majority of the time.

To confirm the accuracy statistics, I wrote an evaluation script (evaluate.py) to generate a confusion matrix for each set.

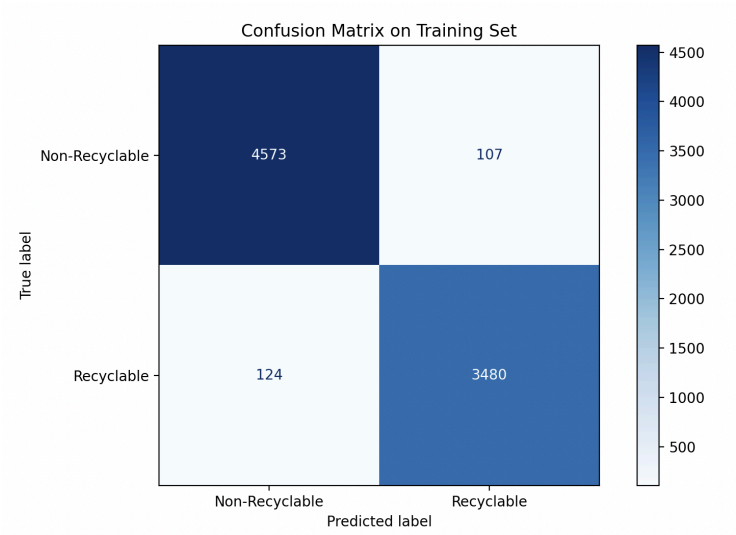


```python
# Load trainined model
model = tf.keras.models.load_model("models/recyclableClassifierFinalModel.keras")

# Load datasets
train_dir = "data/trainingData"
val_dir = "data/validationData"

#iterable dataset objeect for training data
train_ds = tf.keras.utils.image_dataset_from_directory(
    train_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='binary',
    shuffle=False,  # Important for matching y_true and predictions
    #class_names=["non_recyclable", "recyclable"]
)

#iterable dataset object for validation data
val_ds = tf.keras.utils.image_dataset_from_directory(
    val_dir,
    image_size=(224, 224),
    batch_size=32,
    label_mode='binary',
    shuffle=False,
)

#preprocess function to resize same as training script
#but skipping data augmentation since its just evaluating the model
def preprocess(image, label):
    image = image / 255.0
    image = (image - 0.5) * 2.0  # Normalize to [-1, 1]
    return image, label

#preprocesses
val_ds = val_ds.map(preprocess).prefetch(buffer_size=tf.data.AUTOTUNE)
#uncomment training and comment out validation if want to run it for training data
#train_ds = train_ds.map(preprocess).prefetch(buffer_size=tf.data.AUTOTUNE)

# Will collect actual labels and prediction labels
y_true = []
y_pred = []
```

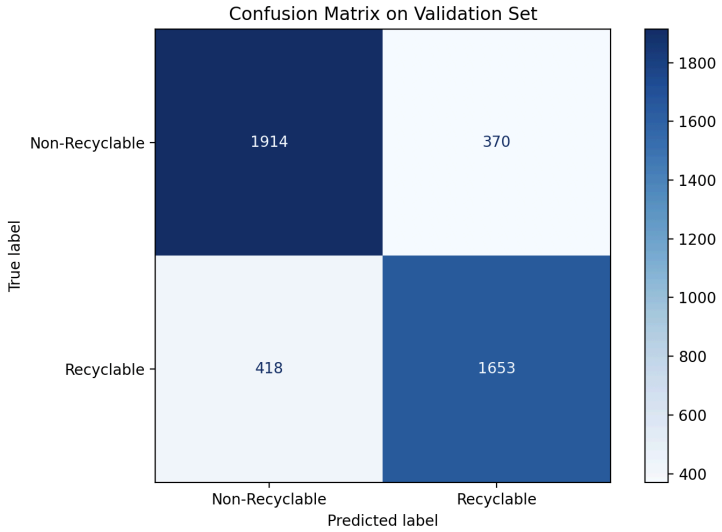
```python
# Will collect actual labels and prediction labels
y_true = []
y_pred = []

#make predictions for all the images in the validation dataset
for images, labels in val_ds: #switch to train_ds if running it for training dataset

    preds = model.predict(images) #predict is a built in function for Keras models
    preds_binary = (preds > 0.5).astype(int).flatten() #0 = non_recyclable & 1 = recyclable
    y_true.extend(labels.numpy().astype(int)) #adds actual label to true list
    y_pred.extend(preds_binary) #adds predicted label to prediction list

# Generates a Confusion Matrix (uses sklearn.metrics)
cm = confusion_matrix(y_true, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=['Non-Recyclable', 'Recyclable'])

plt.figure(figsize=(6, 5))
disp.plot(cmap='Blues', values_format='d')
plt.title("Confusion Matrix on Validation Set") #switch to Training when running it for training dataset
plt.show()

#Prints classification results
print("\nClassification Report:")
print(classification_report(y_true, y_pred, target_names=['Non-Recyclable', 'Recyclable']))
```

This script loads in my trained model as well as the training and validation datasets. Then it creates two iterable dataset objects – one for training and one for validation. These objects specify the location of the data (the path), the image size of the data (224 x 224), the batch size (how many images to load at once), the label type (binary since my labels are recyclable or non-recyclable), and whether or not the data should be shuffled. Note, I have shuffle set to false so that my true values and predicted values alway map to the same image each time. Similar to my training script, I pre-process the image to have each pixel value normalized between -1 and 1. However, I did not perform any data augmentation in my evaluation script since the purpose of this script is just to confirm that my image classifier works. I created two lists – one to store true label values of each image and one to store the predicted label values of each image. Essentially, the true values represent how the data is actually labelled, while the predicted values represent my image classifier's prediction of the image. Then for every image in a dataset, I have my model predict the result and interpret it. Any prediction probability greater than 0.5 will be converted to a 1 to represent a recyclable item and any prediction probability less than 0.5 will be converted to a 0 to represent a non-recyclable item. Both the true and predicted labels are added to their associated list for each image. After this is complete, I used scikit-learn to generate an image of a confusion matrix. The matrix plots the predicted labels versus the true labels for both recyclable and non-recyclable items. The idea of it is to picture how many labels of each type were predicted correctly and incorrectly. Additionally, this script prints the results in the terminal as well just for another view.

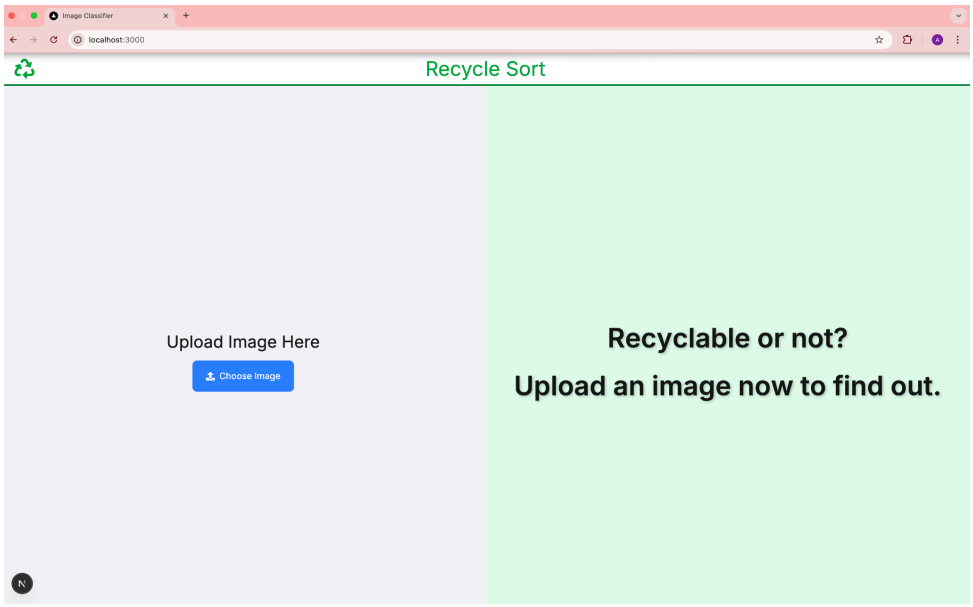
The confusion matrix for the training set is shown below. Out of 4,697 non-recyclable images, my image classifier correctly predicted 4,573 non-recyclable images as non-recyclable and incorrectly predicted 124 non-recyclable images as recyclable. Similarly, out of 3,587 recyclable images, my image classifier correctly predicted 3,480 recyclable images as recyclable and incorrectly predicted 107 recyclable images as non-recyclable. Overall, these results are very good and prove that my model can correctly predict the recyclability of an item a majority of the time.



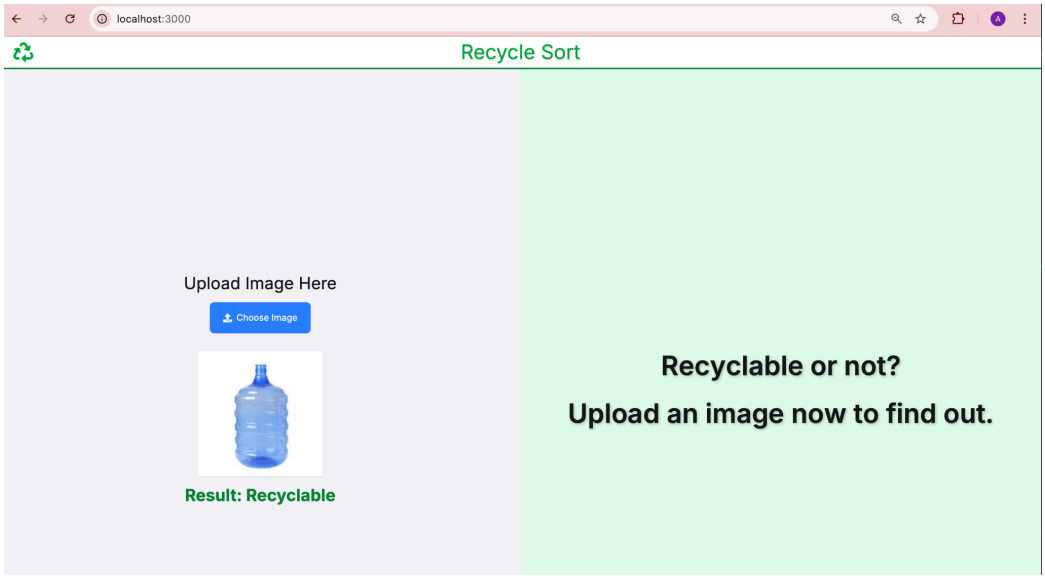
Similarly, the confusion matrix for the validation set is shown below. Out of 2,332 non-recyclable images, my image classifier correctly predicted 1,914 non-recyclable images as non-recyclable and incorrectly predicted 418 non-recyclable images as recyclable. Similarly, out of 2,023 recyclable images, my image classifier correctly predicted 1,653 recyclable images as recyclable and incorrectly predicted 370 recyclable images as non-recyclable. Although not as accurate as the training set, these results are still good and prove that my model can correctly predict the recyclability of an item a majority of the time, even when the image is not something it's been directly trained on.

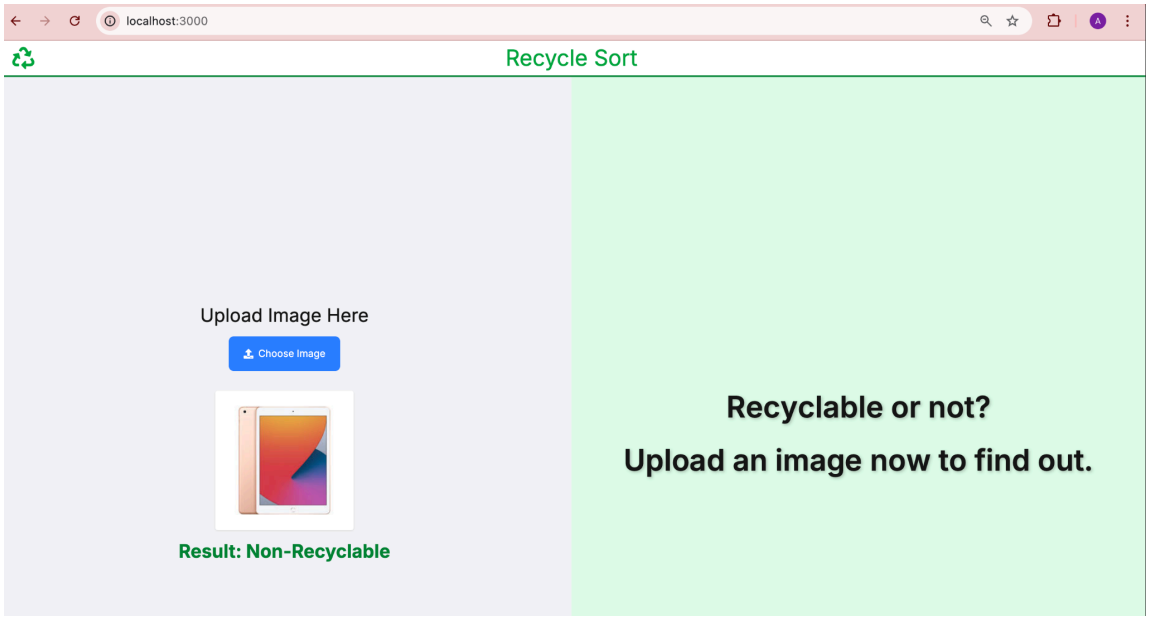


Lastly, to evaluate the usability of my project I tested my web server. The home page of my website is depicted below.



To ensure that my image classification model worked for new predictions I uploaded an image of a recyclable water jug and a non-recyclable iPad. The results are shown below.





The website worked as intended which indicated that my image classification model could accurately predict whether a user-uploaded image is recyclable or non-recyclable.

How to Compile & Run the Code

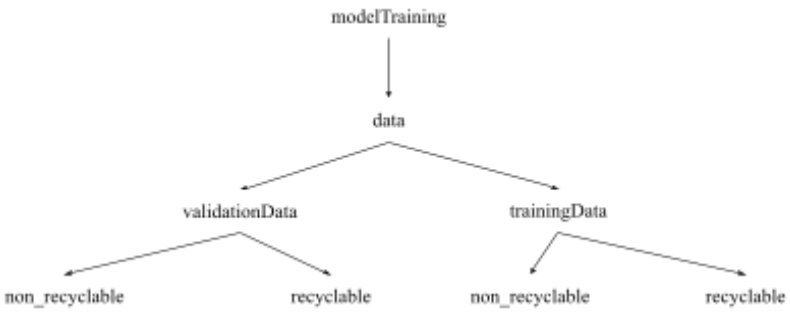
The first step to compiling and running the code is to clone my github repository from https://github.com/ardensentak/CPE462Project.git.

1. **Model Training + Evaluation**

Since the model is already trained, the training script does not have to be run again as it will overwrite my trained model with a new version. Similarly, since the dataset has not changed, the results of the evaluation script will not change either making it unnecessary to run. However, I am still going to provide the instructions on how to run these scripts.

Once the repository is cloned and opened in an IDE (I used VSCode as my IDE) the next step is to activate a python virtual environment and download my dataset.

1. To do this open your terminal and change directories (cd modelTraining) to the modelTraining folder of the project. If not already in the project folder you will first have to change directories into the project folder then use cd modelTraining.
2. Type the following command to create a new virtual environment: "*python -m venv myenv*". You only have to create a new environment once.
3. To activate your environment type: "*source myenv/bin/activate*".
4. Then go to the following link to download my dataset: https://www.kaggle.com/datasets/3d4ff694190d7a677ab09e96706aa76816e76d9ef59f357f233f837317fc5250. Once the dataset is downloaded, unzip the file and move the data folder into the modelTraining folder of the project. The structure of the dataset folders should be as followed:



After the setup is accomplished, the training and evaluation scripts are ready to be run.

5. Ensure your terminal is still in the modelTraining directory (use cd command to get there if not already there)
6. Type "*pip install -r requirements.txt*" in the terminal to download the necessary dependencies for the scripts
7. Run the desired script by typing "*python training.py*" in your terminal to run the training file or type "*python evaluate.py*" in your terminal to run the evaluation file.
8. Once finished running, you can deactivate your virtual environment by typing *deactivate* in the terminal.

2. Running the Application

To run my actual image classifier application you will need to run the frontend and backend at the same time in two separate terminals.

To get the backend running follow these steps:

1. In your terminal, change directories (cd) into the backend app folder of the project. If you are currently in the project directory (CPE462Project) you would type *cd src/backend/app* to get to the backend folder. If not in the project directory, change directories to that first then type *cd src/backend/app*.
2. Install necessary dependencies by typing "*pip install -r requirements.txt*" in the terminal.
3. Change directories into the backend folder. To do this type "*cd ..*".
4. Run the backend server by typing "*uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload*". You will get some terminal output that indicates the backend is running if done properly.

While the backend is running, open another terminal to run the frontend. Follow these steps:

1. In your terminal, change directories (cd) into the frontend folder of the project. If you are currently in the project directory (CPE462Project) you would type *cd src/frontend* to get to the frontend folder. If not in the project directory, change directories to that first then type *cd src/frontend*.
2. Install necessary dependencies by typing "*npm install*" in the terminal. Then, ensure your computer is using node version 20 or later by typing "*nvm install 20*" then "*nvm use 20*" in the terminal.
3. Run the frontend server by typing "*npm run dev*" in the terminal. You will get some terminal output that indicates the frontend is running. Go to the local server link: http://localhost:3000 and you will be able to use my web interface for my project. You can upload images to test my image classification model. Note, that there is a folder in the repository called WebAppTestingImages that contains images of various items that can be used to test my image classifier.

When done with the website, you can close the tab. Then click Ctrl-C in both the frontend and backend terminals to stop them from running.

Source Code

The source code for my project is posted on my GitHub and can be accessed with the following link: https://github.com/ardensentak/CPE462Project.git