Determining Waste Type Using Computer Vision and Machine Learning Models

Jashwanth Sompalli, Samuel Ha, Justin Wong, Aaron Tang*
School of Information - University of California, Berkeley.

(Data Science 281 - Spring 2023)

The rise of single-stream recycling has led to a substantial increase in the quantity of material recycling centers receive, however the quality of material has declined. A high level of contamination has forced recycling centers to throw out entire loads of recycled material due to the high cost of sorting through the waste to find recyclable material. Our study attempts several strategies to classify waste into six classes: cardboard, paper, metal, plastic, glass, and trash. We use XG-Boosted Random Forest and transfer learning with convolutional neural networks such as ResNet-50, ConvNeXt, DenseNet-121, and ResNet-101 for classification. Prior to model training, we extract edge features and color histograms from each image before inputting them into the classification algorithm. In addition to these features, for the convolutional neural networks, we use common data augmentation techniques such as rotating, changing radiation variation, and adjusting image brightness, contrast, and saturation in order to provide the neural network with more randomized data and make it robust to these types of differences in the test data. With a dataset of 2527 high quality and clear images, our model evaluations show that an ensemble trained using the DenseNet-121, ConvNeXt, and ResNet 101 results in the highest level of accuracy, achieving 99.6% test accuracy and 98.6% validation accuracy.

I. INTRODUCTION

Proper disposal and handling of waste is a significant environmental challenge that society faces today. To combat excess waste and lack of reuse, recycling is one of the most common methods. Recycling reduces the amount of waste that ends up in landfills, conserves natural resources, and reduces greenhouse gas emissions which are a byproduct of controlled landfill burns.

The first step of the recycling process involves sorting materials by type and reprocessing them into new products. One of the challenges of recycling is identifying the different types of waste accurately. The most common waste categories are cardboard, paper, trash, glass, plastic, and metal. Due to the nature of how recycling is collected in the United States, sorting these different categories of waste can be a monumental task. The method that is used today is known as single-stream recycling.

Single-stream recycling is a popular method of waste collection. In this system, residents place all recyclable materials, including paper, cardboard, glass, plastic, and metal, in one container for collection. The idea behind single-stream recycling is to make recycling more convenient for residents and increase recycling rates [1].

Despite the benefits of recycling, there are still many obstacles to overcome. One of the main issues is the high level of contamination in recycling streams, which can reduce the value of recycled materials and make them harder to process. Contamination occurs when non-recyclable materials are mixed with recyclable materials, reducing the quality and value of the recycled materials.

This issue is particularly prevalent in single-stream recycling. Contamination happens because people incorrectly categorize non-recyclable objects and waste as recyclable. For example, one of the main causes of contamination in single-stream recycling is the misconception that all plastics are recyclable. While some plastics, such as bottles and jugs, are commonly accepted for recycling, others, such as plastic bags, straws, and utensils, are not. These items can jam sorting equipment or contaminate other materials, making them non-recyclable [2].

To address the challenges created by single-stream recycling methods, we have developed a machine learning pipeline to automate the identification and sorting of the different types of waste commonly included in single-stream recycling. We trained our models on a dataset that contains images of paper, glass, trash, plastic, cardboard, and metal and test the classification model using test data. We ultimately developed an ensemble of deep learning models that classifies waste properly 99.6% of the time among these 6 categories.

II. LITERATURE REVIEW

After the publication of the TrashNet dataset, a number of teams have used the dataset to classify the six different waste classes and published their findings.

In the original study, the authors, Yang and Thung, outline their data collection of the 2,527 images that comprise the TrashNet dataset. The authors mimicked a stream of materials of a recycling plant or a consumer taking an image of a material by taking photos of single objects against a white background. They then use a SVM and CNN to classify the images into six categories of waste. Their training models showed that SVM using SIFT features performed better than CNN, but they acknowledge that the CNN was not trained to its full capability because they had difficulty identifying the opti-

* jashsompalli@berkeley.edu, samqvha@berkeley.edu
justinryanwong@berkeley.edu, ajt385@berkeley.edu

mal hyperparameters. Using SVM, the authors achieved 63% accuracy on test data, and using CNN, they only achieved a test accuracy of 22%, which is marginally better than random classification [3]. Their dataset was released June 6, 2017, one week before June 12, 2017 when the now prominent network architecture, Transformers, [4] was released. Transformers have become prominent because they have a simple and efficient design, yet have successful applications in a variety of areas from machine translation tasks to biomedical imaging [5] among many others.

Aral et al. use a number of well-known deep learning architectures for their classification on the TrashNet dataset. These architectures included DenseNet121, DenseNet169, InceptionResNetV2, MobileNet, Xception. The architectures were fine-tuned using the images from the TrashNet dataset which were augmented to make up for the limited samples. The authors found that the best results were found in DenseNet121 with a test accuracy of 95% followed by InceptionResNetv2 with a test accuracy of 94% [6].

Puspaningrum et al. take a different approach with the TrashNet dataset by using Scale Invariant Feature Transform - Principal Component Analysis (SIFT-PCA) feature extraction and a Support Vector Machine (SVM) classification algorithm. During feature extraction, SIFT is used to extract feature data, and PCA is used to reduce the dimensionality of the SIFT data. The authors find that using the SIFT features without PCA leads to higher levels of accuracy by the SVM, achieving 62% accuracy as compared to 43.6% accuracy using SIFT features with PCA to reduce dimensionality [7].

Thumiki and Khandelwal use the TrashNet dataset to train a convolutional neural network on classifying waste materials for deployment on a mobile application. The authors trained the model using a convolutional neural network with 3 layers and running it through 100 epochs. They found that the accuracy of their model improved substantially after augmenting the input data. Their model was able to classify waste materials into the six classes of waste with 80% accuracy [8].

In addition to the four studies above that made use of the TrashNet dataset which our study also uses, we reviewed the literature regarding machine learning techniques for object detection.

Humeau-Heurtier discuss at a high level the various texture feature extraction methods for texture analysis [9]. We eventually implemented the RGB color histograms, local binary pattern, edge detection on the original dataset of images.

Torrente et al. propose using the Hough transform feature extraction technique to recognize feature curves on surfaces. They identify three steps for feature curve extraction: 1) potential feature points recognition, 2) projection of feature sets onto best fitting planes, and 3) feature curve approximation [10].

Alamri and Pugeault use contextual constraints to improve object detection of two state-of-the-art object detectors (faster RCNN and you only look once). They propose using three categories of contextual constraints: 1) semantic context which refers to the likelihood an object will co-occur with other objects, 2) spatial context which is the likelihood of finding an object in some position with respect to other objects in a scene, and 3) scale context which refers to the size of a reference object with respect to other objects in the scene [11].

Kang et al. propose using a contextual region-based convolutional neural network in order to detect ships from synthetic aperture radar. In the object detection network, the regions of interest are extracted as well as the contextual features around the region of interest. The framework uses deep semantic and shallow high-resolution features to improve detection performance [12].

III. DATA STRUCTURE

A. Class Distributions

The dataset that we utilize to train and test the models is comprised of 2527 total images separated into 6 waste classes. The breakdown for the data are given in the following table.

Class	Count
Cardboard	403
Glass	501
Metal	410
Paper	594
Plastic	482
Trash	137
Total	2527

TABLE I. Data Class Distribution

B. Image Size

Each of the 2527 images are 512 pixels wide and 384 pixels tall (512×384). This was verified manually and through code by using the `identify` utility in bash. The script is included below and executed in the top level of the dataset directory.

```
curdir=$( pwd )
echo $curdir
touch outfile
if [ -e ${curdir}/outfile ]; then
  rm ${curdir}/outfile
  touch ${curdir}/outfile
fi

for i in $( ls ${curdir} ); do

  if [ -d  ${i} ]; then
```

```
    echo ${curdir}/${i}
    echo $( ls -la ${curdir}/${i}| wc -l  )

      cd ${curdir}/${i}
      for j in $( ls -tr1 ); do
        echo $(identify -format '%wx%h'
        ↪  ${curdir}/${i}/${j} )\ >>
        ↪  ${curdir}/outfile
      done
      cd ..

  fi
done

echo $( sort ${curdir}/outfile | uniq )

rm $curdir/outfile
```

C. Other Considerations

Each of the images in the different classes is captured against a white background in well-lit conditions using either room lighting or sunlight. In most of the images, the object is fully in the frame. There are, however, many images in which the object is not completely in the frame.

Another consideration is that the object in the frame is not in any particular orientation. This has implications for our feature selection–we cannot choose features that encode the position or orientation of the object in the image. Below we include an example of an object which is not fully in the frame of the image and an example of an object which is fully within the frame.

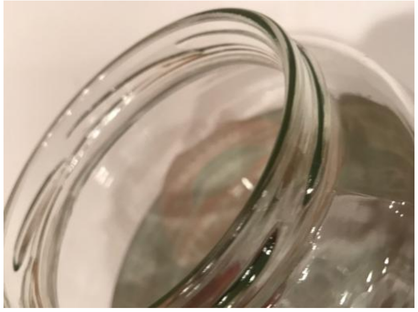

FIG. 1. Not Fully In-Frame

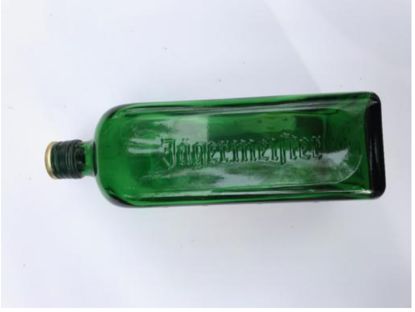

FIG. 2. Fully In-Frame

IV. EXPLORATORY DATA ANALYSIS

Starting the exploratory data analysis was a simple task since all the images are of the same size and the objects are all captured in generally well-lit conditions. We did not have to implement any additional preprocessing for the images to be standardized across all the classes. The first thing we did was extract the color histograms from the images and do a basic cosine similarity search to verify that the color histograms were providing adequate information to find images with similar color distributions. Here we walk through an example of this process. We start with the original image (glass235.jpg) in Figure 3.

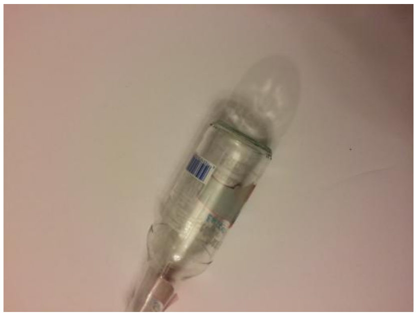

FIG. 3. glass235.jpg

First, we extract each of the color channels from the image and create histograms representing the counts of pixels across the different intensities from 0 to 255 of the color channel. This yields the result in Figure 4.

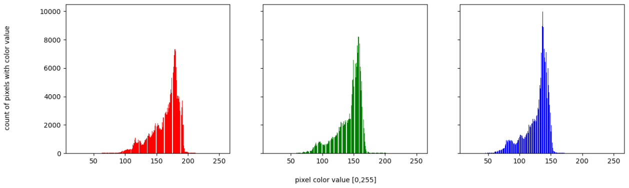

FIG. 4. glass235.jpg Color Histograms

We check to see whether the results obtained from the color histograms are as expected for a small sample of images. Since the results are as expected, we proceed to implement a cosine similarity search to find other images with similar color histograms. We do this by implementing three different functions: `get_vector` which gets the color histogram vector for an image (with a specified amount of bins), `cosine` which calculates the cosine similarity between two vectors, and `search` which takes in the histogram vectors and returns the top images with high cosine similarity to the input image. Snippets of code are included here.

```python
def get_vector(image, bins=32):
    '''
    Use cv2 to calculate color histograms from images using the given bins
    '''
    red = cv2.calcHist(
        [image], [0], None, [bins], [0, 256]
    )
    green = cv2.calcHist(
        [image], [1], None, [bins], [0, 256]
    )
    blue = cv2.calcHist(
        [image], [2], None, [bins], [0, 256]
    )
    vector = np.concatenate([red, green, blue], axis=0)
    vector = vector.reshape(-1)
    return vector

def cosine(a, b):
    '''
    calculate cosine similarity between two vectors
    '''
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

def search(img_vectors, idx, top_k=5):
    query_vector = img_vectors[idx]
    distances = []
    for _, vector in enumerate(img_vectors):
        distances.append(cosine(query_vector, vector))
    # get top k most similar images
    top_idx = np.argpartition(distances, -top_k)[-top_k:]
    return top_idx
```

As an example, the top two images which yield color histograms with the highest cosine similarity to our original image (glass235.jpg) are in Figure 5 and Figure 6.

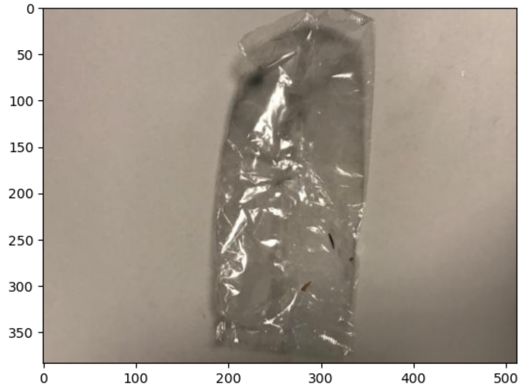

FIG. 5. Best Match

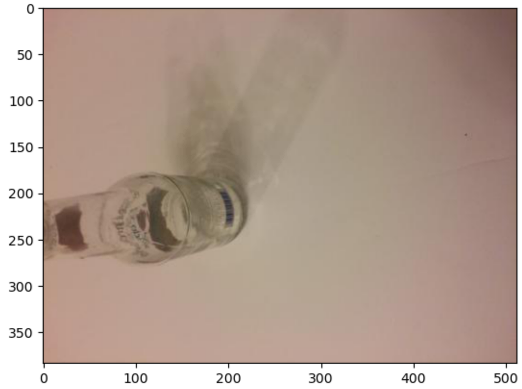

FIG. 6. Second Best Match

We can see that the images that are matched based on cosine similarity of the color vectors are similar, but there is one caveat. The image that is the best match for the example image is not of the same class. The example image is glass, and the best match is plastic. This suggests that only using color features is impractical for classification of transparent objects.

V. FEATURES OF INTEREST

We derived three datasets that were used for our training methods. Each training method required a different dataset in order to apply the features to their corresponding training approach.

1. Images with Edge detection and RGB Color histogram.

    - Dataset 1 ultimately rolled each pixel into its own feature. We also implemented Local Binary Pattern (LBP) here, but because of memory constraints, LBP was excluded from training approaches that used Dataset 1; adding LBP would have added an additional 384*512 parameters. This was primarily used for a XG Boosted Gradient Random Forest approach.

```python
def blur_image(self):

  # Do gaussian blurring if gaussian is
  ↪  passed into object
  if self.kernel == 'gaussian':

    blur_kernel = cv2.GaussianBlur(self.img,
    ↪  (5, 5), 0)
    edges = cv2.Canny(blur_kernel, 100, 200)
    ret, mask = cv2.threshold(edges, 50, 255,
    ↪  cv2.THRESH_BINARY)

  # Do median blurring if median is passed
  ↪  into object
  if self.kernel == 'median':

    blur_kernel = cv2.medianBlur(self.img, 5)
    edges = cv2.Canny(blur_kernel, 100, 100)
    ret, mask = cv2.threshold(edges, 50, 255,
    ↪  cv2.THRESH_BINARY)

  # Do mean blurring if mean is passed into
  ↪  object
  if self.kernel == 'mean':

    blur_kernel = cv2.blur(self.img, (5, 5))
    edges = cv2.Canny(blur_kernel, 100, 100)
    ret, mask = cv2.threshold(edges, 50, 255,
    ↪  cv2.THRESH_BINARY)

  # Save edges to the correct location with
  ↪  the correct filename
  cv2.imwrite(str(self.save_path.resolve()),
  ↪  edges)
```

2. Images with the Edge detection overlaid.

    - This was constructed by applying the Edge detection script across all the images as a pre-processing step before training. The dataset was primarily used for the ResNet-50 model, where we included edge detection from Gaussian, Mean and Median blurring filters.

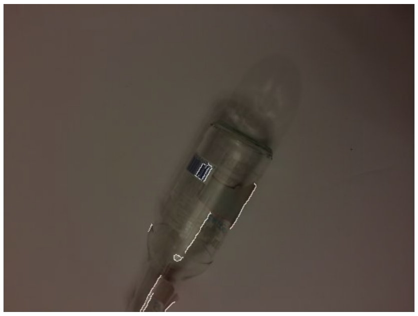

FIG. 7. glass235.jpg Edges overlaid on image

```python
def combiner(category, blur_type):

  '''
  Function to take the edges detected and
↪  overlay them onto the original
  images.
  '''

  combined_loc = Path(
    f'../combined/{category}/{blur_type}'
    )
  original_loc = DATA / category
  original_files = [str(original_loc / i) for
  ↪  i in os.listdir(original_loc)]
  blur_loc = Path(
    f'../outputs/{category}/{blur_type}'
    )
  blur_files = [str(blur_loc / i) for i in
  ↪  os.listdir(blur_loc)]

  for i, v in enumerate(original_files):
    original_img = cv2.imread(v)
    edges = cv2.imread(blur_files[i])
    filename = v.split('/')[-1]

    output = cv2.addWeighted(original_img,
    ↪  0.5, edges, 0.5, 0.0)

    cv2.imwrite(str(combined_loc / filename),
    ↪  output)
```

3. Raw Images from the original dataset with some data augmentation.

    - Because deep neural nets are largely unexplainable, we trusted that the architecture would allow the neural network to find the right patterns and connections that ultimately lead to accurate and correct predictions. As a preprocessing step to the neural networks, we resized images to 224x224 and added random augmentation such as ColorJitter (adjusts image brightness, contrast, saturation), RandomRotation (adjusts rotation), RandomAffine (adjusts radiation variation), and Normalize. This same preprocessing and augmentation was used for training the ConvNeXt, DenseNet, and RestNet-101 models as well. The raw images by themselves were used for training ResNet-18.

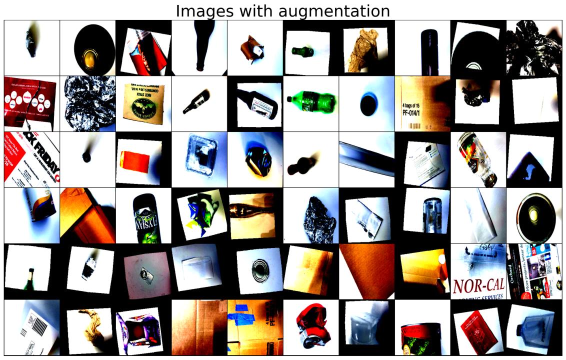

FIG. 8. Sample of image augmentations

```
 # Process data for CNNs and augment data
transformer = transform.Compose([
  transform.Resize((224, 224)),
  #Random image brightness, contrast, saturation
  transform.ColorJitter(brightness=0.2,
  ↪   contrast=0.2, saturation=0.2),
  #Random flip
  transform.RandomRotation(5),
  #Random radiation variation
  transform.RandomAffine(degrees=11,
  ↪   translate=(0.1,0.1), scale=(0.8,0.8)),
  transform.ToTensor(),
  transform.Normalize((0.485, 0.456,
  ↪   0.406),(0.229,0.224,0.225))
])
```

VI. MODEL SELECTION

A. Simple Models

Using Dataset 1, we trained a XG Boosted Gradient tree using 5-K Fold Cross Validation. XG Boosted Gradient was used because of its use of regularization in its loss function which penalizes the wrong predictions during training. This served as our true baseline model, outside of random guessing (expected accuracy if 1/6 or 16.67%).

We also attempted to train a KNN (K-Nearest Neighbors) Classifier with the idea that images with similar colors or pixel values should be a strong enough indicator to separate out the six trash classes and thus be separable into local clusters. However, each row in our training data when flattened had 589,824 features. Therefore, memory constraints caused the KNN implementation to crash so we determined this approach was not viable. We also developed our own “pseudo-KNN” manual implementation. After doing a 80-20 train-test split, we computed the cosine similarity score between each row in the entire dataset and each row in the training data. This alternative approach then uses an ensemble approach of using the top 100 similarities scores as “votes”. The class which receives the majority of the “votes” is chosen as the predicted class.

This “pseudo-KNN” took 13 hours to complete, and the accuracy achieved was 16%, which is about as good as randomly guessing the class. The execution of the model was very slow because every row needed to compute the cosine similarity with every other row to determine which groupings were most similar.

B. Complex Models

For Dataset 2, we trained ResNet-50. The focus of this training experiment was hyperparameter tuning and to explore how the different blur filters would affect the accuracy using the same model. Given that our training images were high quality with good lighting and white backgrounds, there was very little natural noise.

For Dataset 3, we trained ResNet-18, ConvNeXt, ResNet-101, and DenseNet. The focus of this training experiment was to treat the model as a black box and allow the deep architecture to learn the weights. For ResNet-18, we used the raw images alone and trained for 25 epochs. For the ConvNeXt, ResNet-101, and DenseNet, instead of adding additional features, we augmented images to introduce additional noise, which should force the more prominent, relevant features to be detected by the neural net and trained for 10 epochs each. The data augmentation boosted overall performance substantially.

```
def train(seed, epochs, model):
  model.to(device)
  criterion = nn.CrossEntropyLoss()
  if seed==2:
    optimizer =
    ↪   torch.optim.Adam(model.fc.parameters(),
    ↪   lr=0.001, betas=(0.9, 0.999), eps=1e-08,
    ↪   weight_decay = 1e-5)
  else:
    optimizer =
    ↪   torch.optim.Adam(model.classifier.parameters(),
    ↪   lr=0.001, betas=(0.9, 0.999), eps=1e-08,
    ↪   weight_decay=0)
  scheduler =
  ↪   torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, mode='max', factor=0.1,
        ↪   patience=3, verbose=True)
  #scheduler =
  ↪   torch.optim.lr_scheduler.OneCycleLR(optimizer,
  ↪   0.1, epochs=epochs,
  ↪   steps_per_epoch=len(loaders['train']),
  ↪   cycle_momentum=True)
  #scheduler =
  ↪   torch.optim.lr_scheduler.StepLR(optimizer,
  ↪   3, gamma=0.1)
  since = time.time()
  best_model = copy.deepcopy(model.state_dict())
  best_acc = 0.0
  print("Optimizer: ", optimizer)
  print("Scheduler: ", scheduler)
  print('Starting training...')
  for epoch in range(epochs):
```

```
  for phase in ['train', 'val']:
   if phase == 'train':
    model.train()
   else:
    model.eval()

   running_loss = 0.0
   running_corrects = 0.0

   for inputs, labels in loaders[phase]:
    inputs, labels = inputs.to(device),
    ↪  labels.to(device)
    optimizer.zero_grad()

    with
    ↪  torch.set_grad_enabled(phase=='train'):
     outp = model(inputs)
     _, pred = torch.max(outp, 1)
     loss = criterion(outp, labels)

     if phase == 'train':
      loss.backward()
      optimizer.step()
#      lr.append(scheduler.get_lr())
#      scheduler.step()

    running_loss += loss.item()*inputs.size(0)
    running_corrects += torch.sum(pred ==
    ↪  labels.data)

   if phase == 'train':
     acc = 100. * running_corrects.double() /
     ↪  dataset_sizes[phase]
     scheduler.step(acc)

   epoch_loss = running_loss /
   ↪  dataset_sizes[phase]
   epoch_acc =
   ↪  running_corrects.double()/dataset_sizes[phase]
   losses[phase].append(epoch_loss)
   accuracies[phase].append(epoch_acc)
   if phase == 'train':
    print('Epoch: {}/{}'.format(epoch+1,
    ↪  epochs))
   print('{} - loss:{},
   ↪  accuracy{}'.format(phase, epoch_loss,
   ↪  epoch_acc))
   lr.append(scheduler._last_lr)

   if phase == 'val':
    print('Time: {}m {}s'.format((time.time()-
    ↪  since)//60, (time.time()- since)%60))
    print('=='*31)
   if phase == 'val' and epoch_acc > best_acc:
    best_acc = epoch_acc
    best_model =
    ↪  copy.deepcopy(model.state_dict())
  #scheduler.step()
 time_elapsed = time.time() - since
 print('CLASSIFIER TRAINING TIME {}m
 ↪  {}s'.format(time_elapsed//60,
 ↪  time_elapsed%60))
 print('=='*31)
```

VII. RESULTS & COMPARISONS

We measured the test, train, and validation accuracy of each model. The XG Boost model does surprisingly well, considering it was not designed for handle scaling, rotation, and other transformations. However, it is clearly not as good as the neural networks in handling such variations. In our comparison of ResNet-18, ResNet-50, ConveNeXt, DenseNet-121, and ResNet-101, we find that the more parameters a model has, often the better the accuracy becomes. This might be because deeper networks simply have more layers and features that could make the finer grain connections to the details of the input image that are important features in the image.

Comparing the three that were eventually used as an ensemble, we see all three have near perfect accuracy. Augmenting the data and deepening the network architecture helped with detecting the right class for our use case. As shown in Figure 9 below, there does not seem to be any significant overfitting.

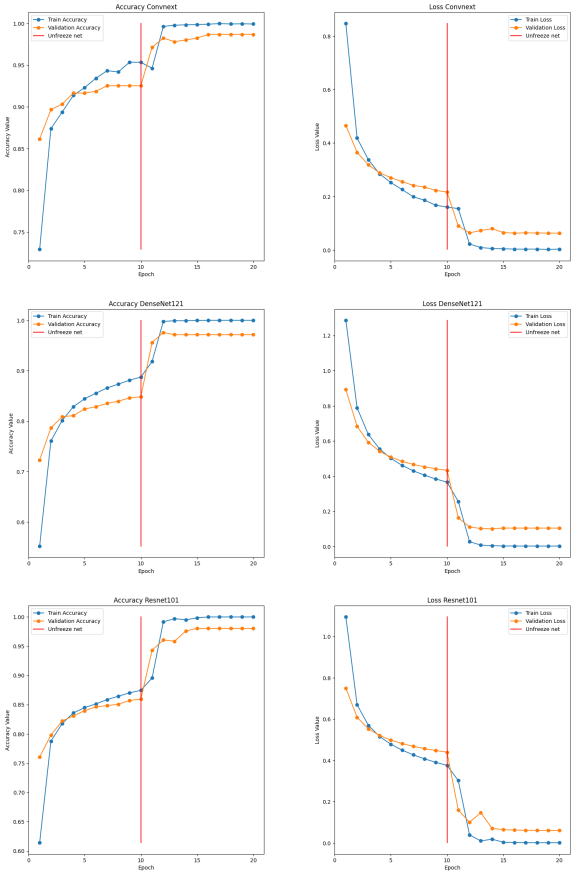

FIG. 9. ConveNeXt vs DenseNet vs ResNet-101

From the results below, there does not seem to be any strong evidence of overfitting as the test and train accuracy of each model do not differ significantly.

Because we were also interested in how easy it would be for recycling professionals to train and deploy such a

Version	Model	Test	Train	Val
V1	XG Boost	53%	52%	53%
Original	ResNet18	88%	83.3%	89.5%
Original	ResNet50	90.7%	89.1%	89%
Gaussian Edge	ResNet50	95%	91.5%	93%
Median Edge	ResNet50	93%	90%	92%
Mean Edge	ResNet50	94%	93%	92%
V1- Ensemble	ConvNeXt	N/A	93%	92%
V1- Ensemble	**DenseNet-121**	**99.6%**	**99.9%**	**98.6%**
V1- Ensemble	ResNet-101	N/A	88.7%	84.8

classification model, we also kept track of the time it took to train the models discussed. The table below shows the model complexity and their training time in minutes.

Version	Model	N. Params	Time (min)
V1	XG Boost	2049	26
Original	ResNet18	11.7M	20
Original	ResNet50	25.5M	35
Gaussian Edge	ResNet50	25.5M	35
Median Edge	ResNet50	25.5M	35
Mean Edge	ResNet50	25.5M	35
V1- Ensemble	ConvNeXt	27.8M	274
V1- Ensemble	**DenseNet-121**	**7.6M**	**83**
V1- Ensemble	ResNet-101	42.5M	143

Looking at the test predictions of the ensemble method, we see only 2 out of the 506 predictions were incorrect, resulting in an accuracy 99.6%.

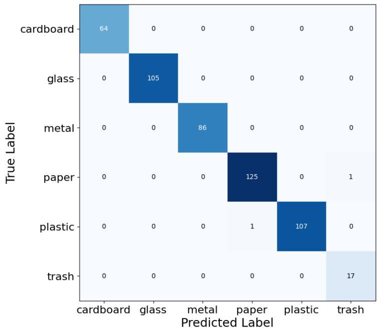

FIG. 10. ConveNeXt vs DenseNet vs ResNet-101 Confusion Matrix Results for Waste Classification

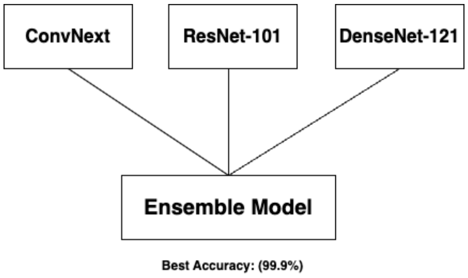

FIG. 11. Ensemble model

VIII. DISCUSSION

As seen through the results above, DenseNet provided the highest accuracy on all test, train and validation dataset with an accuracy of at least 98.5%.

The XG-Boost Random Forest Classifier only yielded an accuracy of 50% and was our worst performing algorithm. This may be because the random forest only creates branches on the features we created and treats each value we provide (colour/edge histogram value) as a feature. The RFC seems much less flexible and adept at handling translations, scaling, and rotation problems that HOG, SIFT, and deep neural networks seem to be able to better process.

Nevertheless, the XG Boosted trained relatively quickly. Training time was under 30 minutes; the best XGBoost tree had 53% training accuracy and 52% test accuracy. The training speed is expected because Xtreme Gradient Boosting takes advantage of parallel computing since the approach has both a linear model solver and tree learning algorithms, both of which can be optimized and run in parallel.

The results from ResNet-18, 50, and 101 were much more accurate, with ResNet-50 Gaussian edge combined model performing the best with a test accuracy of 95% and validation accuracy 93%.

ResNet-18 gave an test accuracy of 89% but did not perform as well on training data with 83% accuracy. This shows that the model may be underfitting, which usually occurs when the model is not complex enough and is unable to generalize well to the test data. The much higher accuracy on all train, test, and validation sets for ResNet-50 further supports this.

Although ResNet-50 seemed to perform significantly better than ResNet-18, ResNet-101 did not seem to identify waste more accurately. For simpler datasets where features are easier to identify and classify, ResNet-50 may be a better fit as the reduced amount of layers allows the model to be more easily trained and deployed. ResNet-101 is more computationally complex and may overfit the training data. However, we should not simply discard the ResNet-101 model as waste identification in real-life scenarios may not be as straightforward as the current images in our dataset, and the increased layer complexity in ResNet-101 may provide better accuracy as compared to ResNet-50.

ResNet-50 was trained using all variations of Dataset 2, in which we used different blurring filters. We were surprised initially that the Gaussian filter had the worst performance, compared to the Median and Mean filters, which were about the same. We tuned the blurring filter and found that the mean or median work better for our images.

DenseNet produced the highest accuracy across the spectrum with minimal differences between test and train accuracies. This shows that the model is generalizable and all else equal is the most suitable model for waste classification among the models that we tried. However, despite the accuracy, DenseNet took the longest time to run. Even with GPU computing, the model took 10 hours to complete running. This is considerably longer than ResNet-50, which only took 35 minutes to complete 10 epochs.

Even though the over 99% test accuracy of the ensemble method is nearly perfect, the dataset used was fairly small with under 2600 images in total. The limitation of the work done here is that the images were taken in relatively ideal conditions, while the real world is noisy. We tried data augmentation in Dataset 3, but still there are not large datasets of labelled images of waste readily and publicly available. Future next steps to expand this would be to gather real world examples at municipalities and waste management facilities, where the background makes a difference and could allow for contextual features as well. Further expansion would include supporting additional other types of trash, such as styrofoam, precious metals, rubber, and other commodities that can be recycled. This not only would allow for a better, more diverse dataset, but would also allow the model to become more practical in the real world.

IX. CONCLUSION

We managed to significantly increase the accuracy of waste classification compared to the literature using the TrashNet dataset presented above. We consistently reached accuracies of 90% and above for our algorithms. Among the different algorithms, the most successful was DenseNet, where we achieved an accuracy of 99% and above.

An issue we will need to consider in real life implementation is a trade-off between accuracy and complexity. Although DenseNet provided the highest accuracy, the runtime may be relatively less efficient due to high memory usage. We spent over 10 hours running DenseNet on GPU, whereas it took about 35 minutes to run 10 epochs of ResNet-50. DenseNet has a high memory footprint in each layer as all feature maps from previous layers are concatenated together, which can lead to a large number of feature maps. However, if we have more computing resources and a requirement for the most accurate algorithm, DenseNet is definitely the most appropriate algorithm for this purpose.

In the future, we need to test the generalizability of our algorithms with external images. Additionally, we need to ensure our model performs well in typical settings for waste sorting, as the types of images processed during real-time classification will likely be more complex. As mentioned above, more complex models such as ResNet-101 or DenseNet may be more adept at capturing key features necessary for waste classification.

[1] R. Partnership, "Contamination impact: What really happens when the wrong things get in the bin," `https://recyclingpartnership.org/wp-content/uploads/2019/07/Recycling-Partnership-Contamination-Impact-Report.pdf` (2019).

[2] E. Binion, J. Gutberlet, and C. Satterfield, Journal of Public Affairs **e1896** (2018), 10.1002/pa.1896.

[3] M. Yang and G. Thung, (2017).

[4] Vaswani, Shazeer, Parmar, Uszkoreit, Jones, Gomez, Kaiser, and Polosukhin, (2017).

[5] F. Shamshad, (2022).

[6] R. A. Aral, Şeref Recep Keskin, M. Kaya, and M. Hacıömeroğlu (Institute of Electrical and Electronics Engineers, Seattle, Washington, USA, 2018).

[7] A. P. Puspaningrum, S. N. Endah, P. S. Sasongko, R. Kusumaningrum, Khadijah, Rismiyati, and F. Ernawan (Institute of Electrical and Electronics Engineers, Semarang, Indonesia, 2020).

[8] M. Thumiki and A. Khandelwal (Institute of Electrical and Electronics Engineers, Bhopal, India, 2022).

[9] A. Humeau-Heurtier, (2015).

[10] M.-L. Torrente, S. Biasotti, and B. Falcidieno, Pattern Recognition **73**, 111 (2018).

[11] F. Alamri and N. Pugeault, IEEE Transactions on Cognitive and Developmental Systems **14**, 1320 (2022).

[12] M. Kang and et al., Remote Sensing **9** (2017).