Region Proposal Object Detection Comparative Performance Analysis

By
Karthik Datta Yerram (ky2482)
Vasanth Margabandhu (vm2656)
Aiman Shariff (as6476)

Goal:

Our main goal is to train an object detection model without the use of a dataset labelled with bounding boxes, and to analyze which classification architecture works best for any given scenario.

Solution: We propose a pipeline that uses selection search algorithm on the image and it proposes the regions that could potentially contain an object. Each region is then fed to a classifier which is fine tuned to a target dataset (food dataset in our case) that classifies the object in that region. This whole pipeline is can be used as an object detection pipeline.

Value: Our solution circumvents the tedious process of labelling bounding boxes on the dataset for the purpose of training an object detection model. Additionally, it can be used for generating a dataset for object detection models.

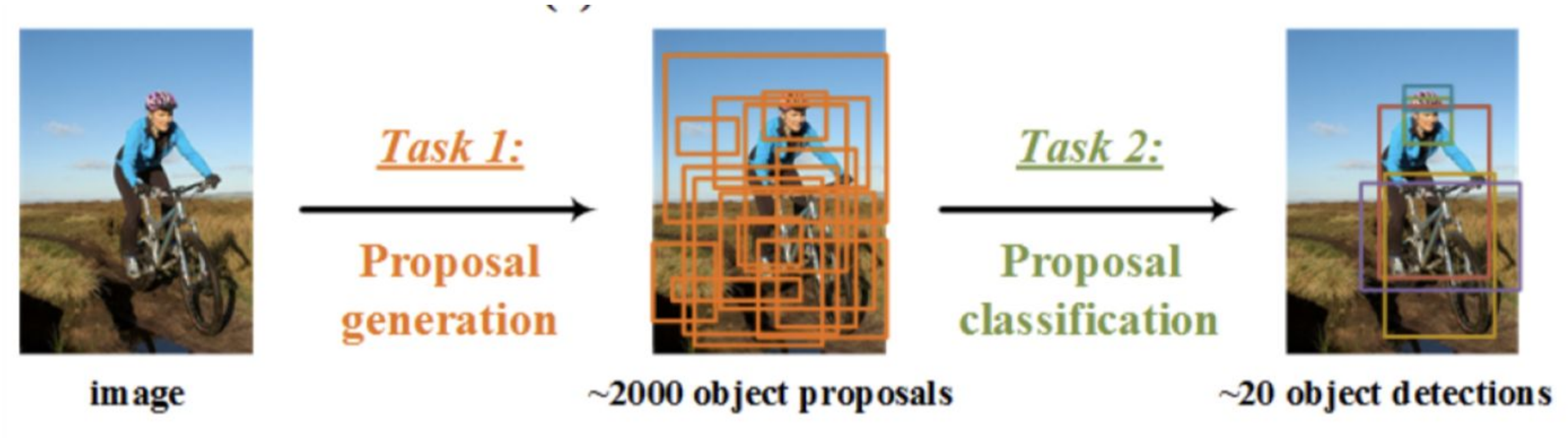


Fig 0: Broad Outline.

Problem Motivation

- If we need to build an object detector on a particular dataset, oftentimes we do not find a dataset that is labelled with bounding boxes.
- In such cases, one needs to invest a significant amount of time manually labelling bounding boxes on every object in every image.
- This motivated the idea of a region proposal object detector.
- Selective search is preferred to traditional object detection methods that involve sliding windows and image pyramids, as it isn’t as slow and isn’t sensitive to parameter choices like window size and pyramid scale.

Background Work

Selective Search, first introduced by Uijlings et al. in their 2012 paper, ***Selective Search for Object Recognition***, is a critical piece of computer vision, deep learning, and object detection research. In their work, Uijlings et al. demonstrated:

1. How images can be *over-segmented* to automatically identify locations in an image that *could* contain an object
2. That Selective Search is *far* more computationally efficient than exhaustively computing image pyramids and sliding windows (and without loss of accuracy)
3. And that Selective Search can be swapped in for *any* object detection framework that utilizes image pyramids and sliding windows

- Yosinski et al. in their 2014 paper, ***How transferable are features in deep neural networks?***
- https://medium.com/@manasnarkar/transfer-learning-getting-started-9cebf5855a08

The above two resources gave us sufficient guidance in implementing transfer learning and adapting it to our particular use case.

Technical Challenges

- The first technical challenge that we faced was, during the selective search, the algorithm's default hyperparameters like the minimum and maximum size of the region proposal window produced inaccurate bounding boxes.
- Once we got the bounding boxes + probabilities, we wanted to filter out the weak classifications, but the confidence levels across each class was not uniform, i.e, high confidence for a particular class may correspond to moderate confidence for another class (in absolute terms). So after multiple iterations, we understood the right threshold for setting the probability in order to efficiently eliminate weak predictions across all classes.

Technical Challenges (Contd.)

- While fine tuning the classifiers, we had a hard time in finding the right optimizer. The models took lot of time in training and reaching a certain accuracy due to the size of the dataset.
- When we change the optimizer we had to run the whole model from scratch. After multiple trials, we decided, SGD worked the best for our models.
- “Imagenet_utils.decode predictions “ is the function to output, class name + probabilities. This function was only suited for datasets having the same labels and number of classes as ImageNet. As such, we had to write our own decode function specific to our dataset. (Details provided in the approach).
- We ran our experiments on Google Colab, which didn’t support the use of the imshow function in cv2. As a workaround, we had to use cv2_imshow from the google.colab.patches module.

Approach

- We used ResNet50, Inceptionetv3, MobileNet, VGG16, EfficientnetB0 and EfficientnetB1 pre-trained on ImageNet (transfer learning) and then fine tuned each classifier with SGD as the optimizer, lr=0.001 and momentum=0.9 on the Food101 dataset.
- These models were trained till 93% accuracy was achieved.
- EfficientnetB0 and B1 were discarded at this stage due to significant training time and poor training and validation accuracy.
- We used callbacks to save the best model based on validation loss.

Table 1: Analysis on models to reach 93% accuracy

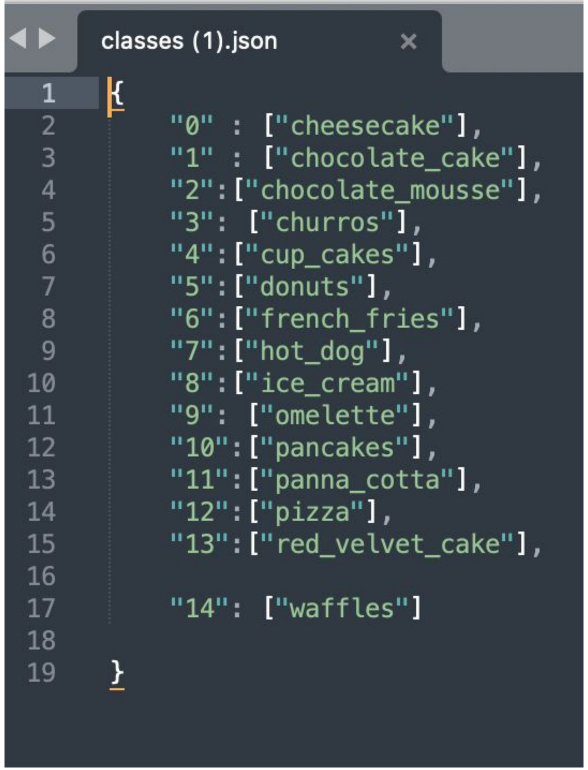
MODEL	NUMBER OF PARAMETERS	TIME (seconds)	NUMBER OF EPOCHS
VGG16	138M	23040	36
Resnet50	23M	11660	22
InceptionetV3	24M	10350	23
Mobilenet	6.9M	12150	30

After our training and gathering the results, the model that took the least time to train was InceptionetV3. The highest time was taken by VGG16. Talking in terms of memory, Mobilenet is a lighter model than Inceptionv3, if there is a memory constraint we can use Mobilenet and not Inceptionet.

Table 2: Analysis on models with respect to Test Time and Test Accuracy.

MODEL	TIME IT TOOK TO TEST ON 3750 IMAGES (seconds)	ACCURACY
VGG16	72.63	84.71%
Resnet50	49.89	87.38%
InceptionetV3	38.35	88.21%
Mobilenet	29.25	83.54%

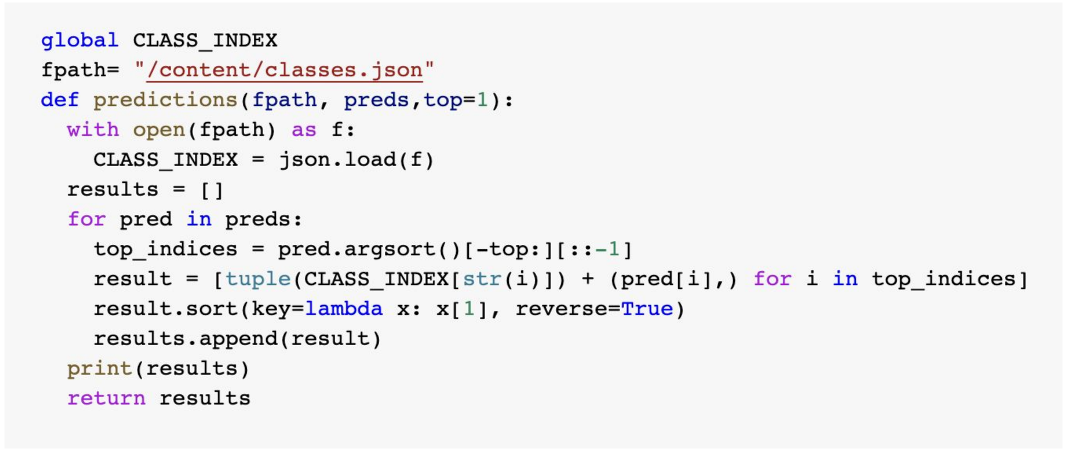
From the above table it can be inferred that inceptionetV3 had the highest Test Accuracy but it is not the fastest. Where as the Mobilenet model was the fastest yet not the most Accurate. So here comes a Trade-Off.
With the help of this table one can decide which model to choose on the basis of his requirement(speed or accuracy)



```
{
      "0" : ["cheesecake"],
      "1" : ["chocolate_cake"],
      "2":["chocolate_mousse"],
      "3": ["churros"],
      "4":["cup_cakes"],
      "5":["donuts"],
      "6":["french_fries"],
      "7":["hot_dog"],
      "8":["ice_cream"],
      "9": ["omelette"],
      "10":["pancakes"],
      "11":["panna_cotta"],
      "12":["pizza"],
      "13":["red_velvet_cake"],

      "14": ["waffles"]

}
```



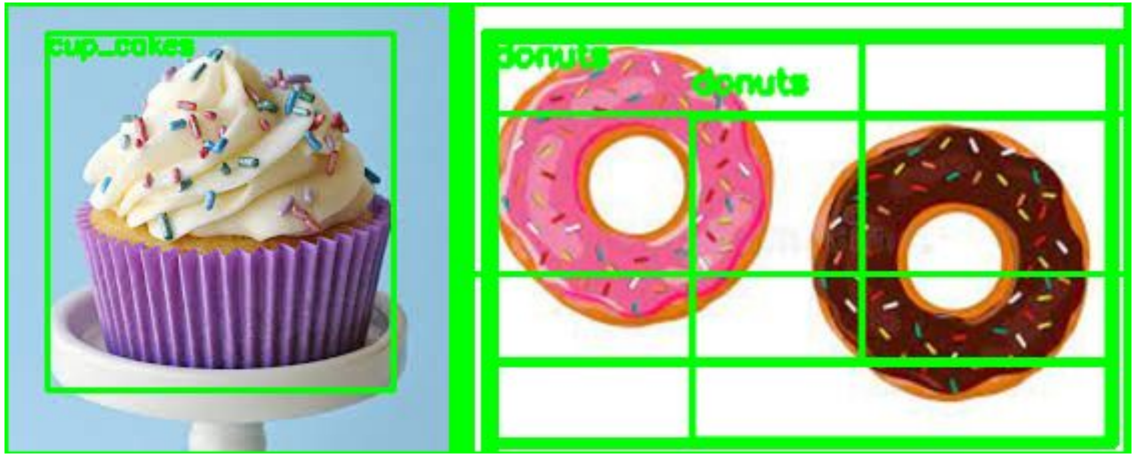
```
global CLASS_INDEX
fpath= "/content/classes.json"
def predictions(fpath, preds,top=1):
  with open(fpath) as f:
    CLASS_INDEX = json.load(f)
  results = []
  for pred in preds:
    top_indices = pred.argsort()[-top:][::-1]
    result = [tuple(CLASS_INDEX[str(i)]) + (pred[i],) for i in top_indices]
    result.sort(key=lambda x: x[1], reverse=True)
    results.append(result)
  print(results)
  return results
```

Fig 2- Custom decoder function.

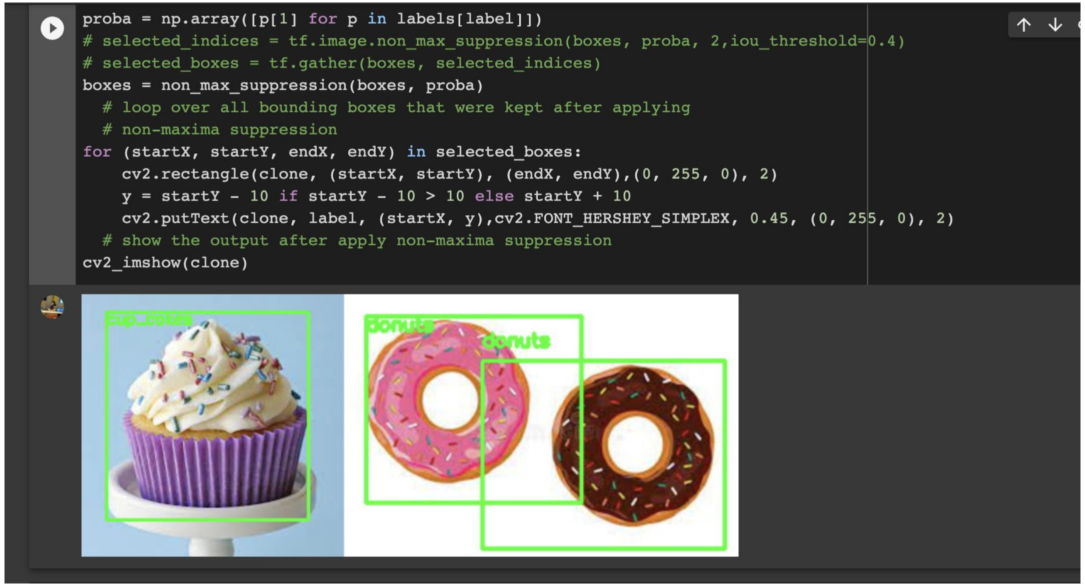
Fig 1- JSON file with 15 classes and their class IDs.

Since training on 1000 images each of 101 classes of the Food101 dataset was a tedious task, we picked 15 classes of our choice, and wrote a custom decoder function rather than using the imagenet_utils function for the same.

- The best classifier architecture is decided based on our time and memory constraints, and is forwarded to the object detection portion of the pipeline.
- After finding the best hyperparameters through trial and error, we use our object detection script to output the regions that could have an object of interest using selective search.

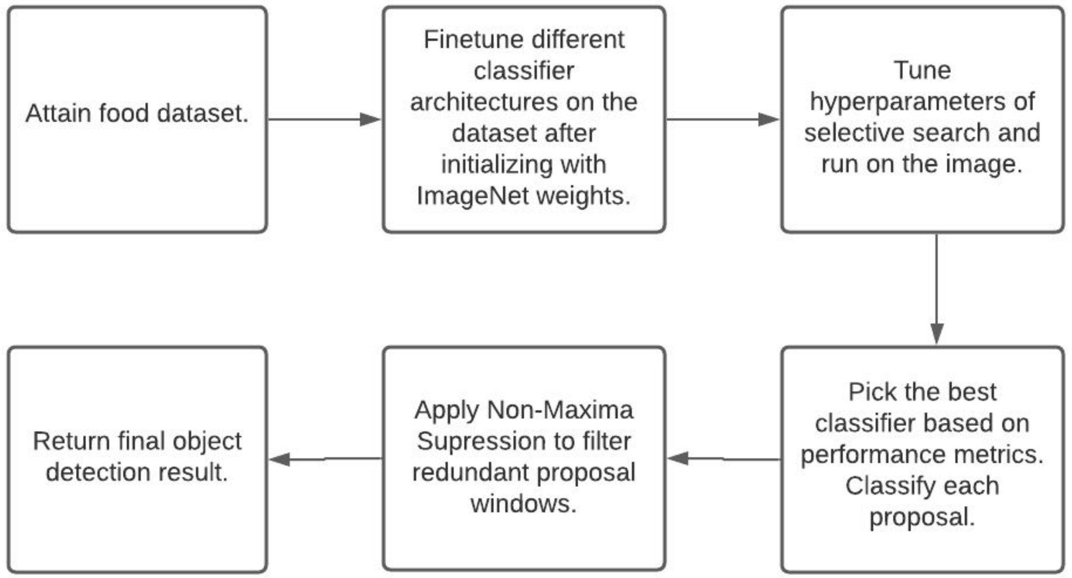


- We then apply non-maxima suppression to suppress weak, overlapping bounding boxes.
- Then, each region is passed to the classifier and then predicted.
- Finally, we return the classified objects.



```
proba = np.array([p[1] for p in labels[label]])
# selected_indices = tf.image.non_max_suppression(boxes, proba, 2,iou_threshold=0.4)
# selected_boxes = tf.gather(boxes, selected_indices)
boxes = non_max_suppression(boxes, proba)
  # loop over all bounding boxes that were kept after applying
  # non-maxima suppression
for (startX, startY, endX, endY) in selected_boxes:
    cv2.rectangle(clone, (startX, startY), (endX, endY),(0, 255, 0), 2)
    y = startY - 10 if startY - 10 > 10 else startY + 10
    cv2.putText(clone, label, (startX, y),cv2.FONT_HERSHEY_SIMPLEX, 0.45, (0, 255, 0), 2)
  # show the output after apply non-maxima suppression
cv2_imshow(clone)
```

Solution Diagram


Attain food dataset.
Finetune different classifier architectures on the dataset after initializing with ImageNet weights.
Tune hyperparameters of selective search and run on the image.
Pick the best classifier based on performance metrics. Classify each proposal.
Apply Non-Maxima Supression to filter redundant proposal windows.
Return final object detection result.

Implementation Details

Hardware: NVIDIA K80 GPU on Google Colab platform.

Software: Keras, Tensorflow, OpenCV.

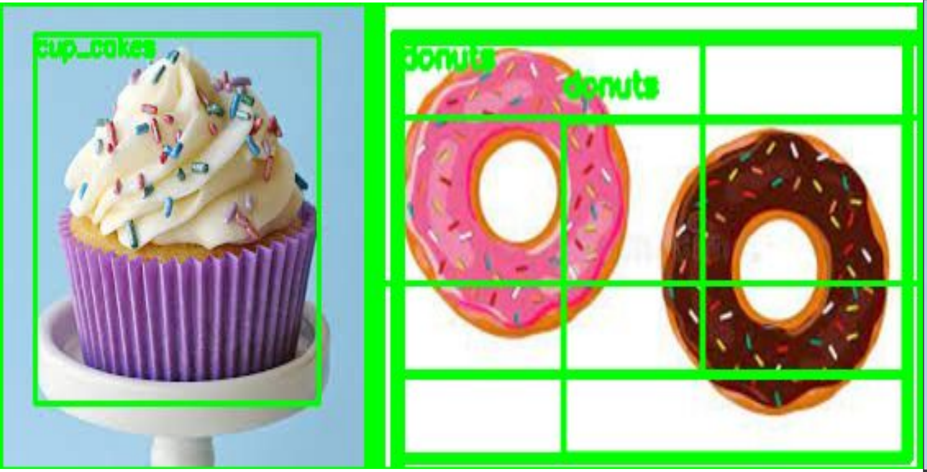
Dataset: Food101.

Experiment Design Flow

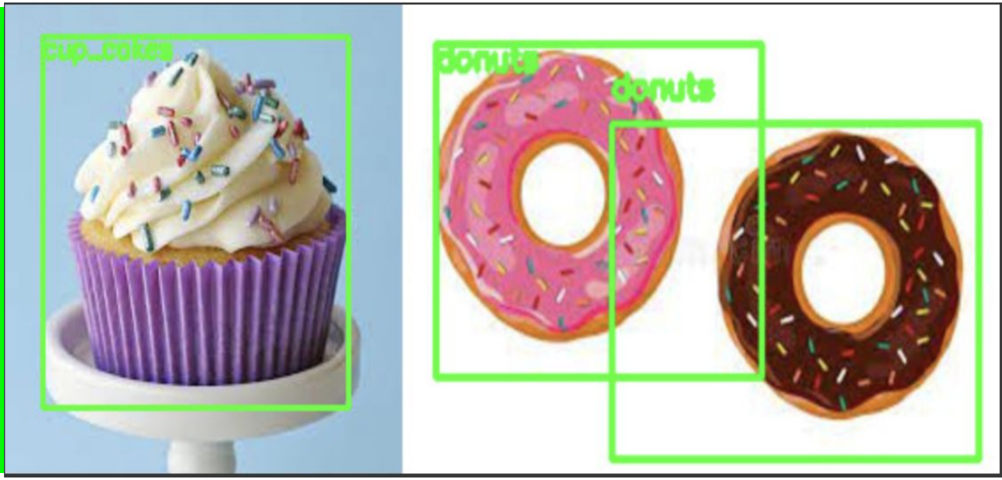
We will be showing object detection results on images having a single object and images having multiple objects of the same or different class.

- Results before Non Maximum Suppression are first displayed.
- Then results after applying Non Maximum Suppression are shown.
- In cases where we could not find images having two different objects of specific classes, we merged two images having different classes into one single image.

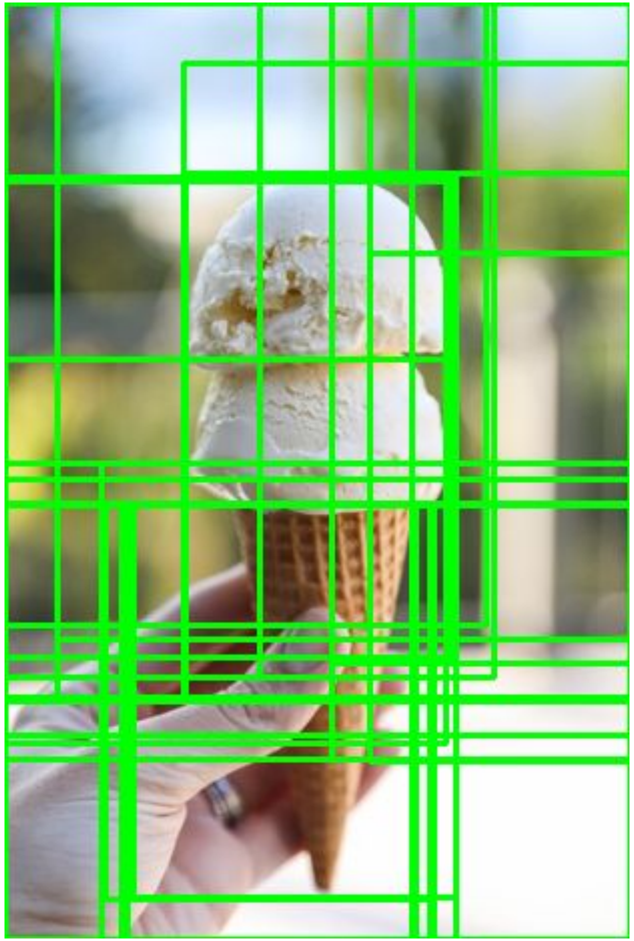
Experimental Evaluation



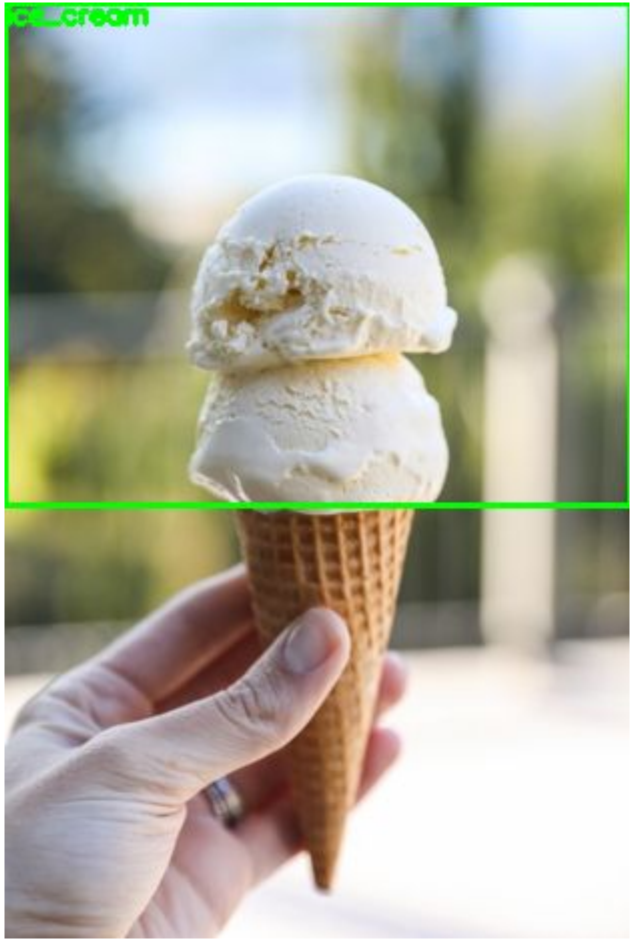
Before NMS



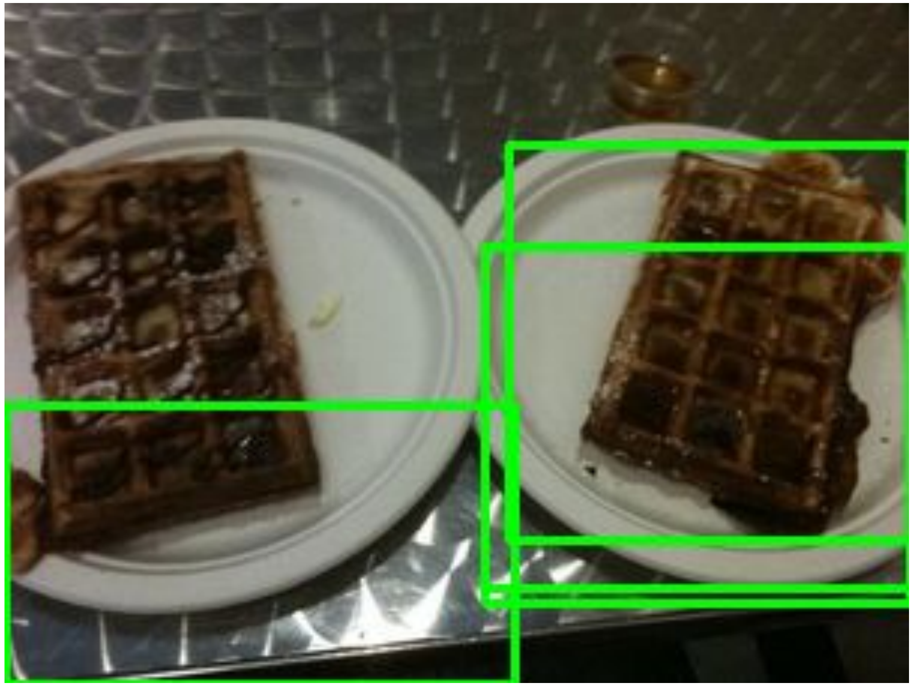
After NMS



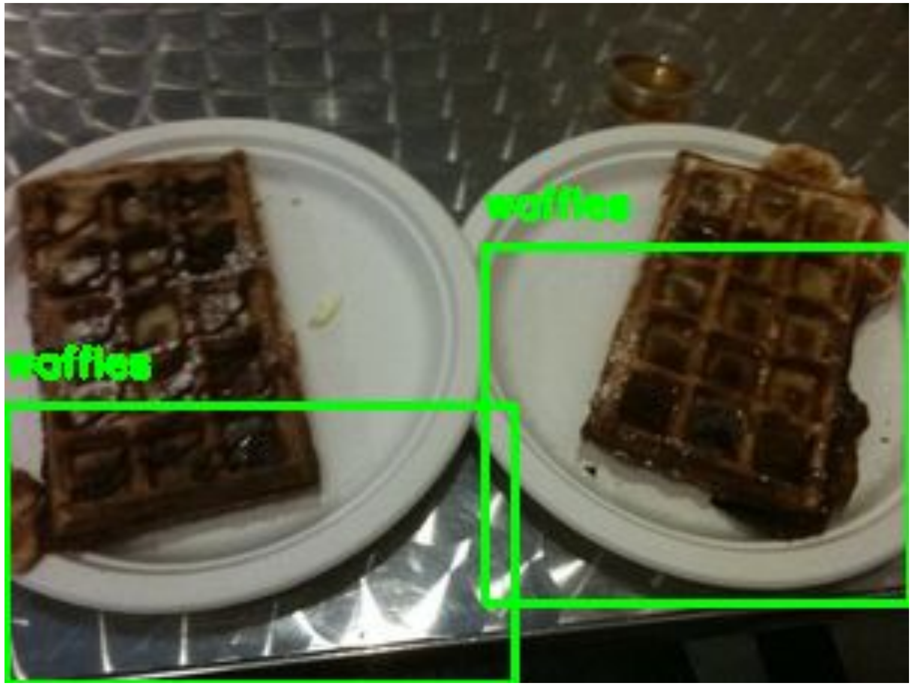
Before NMS



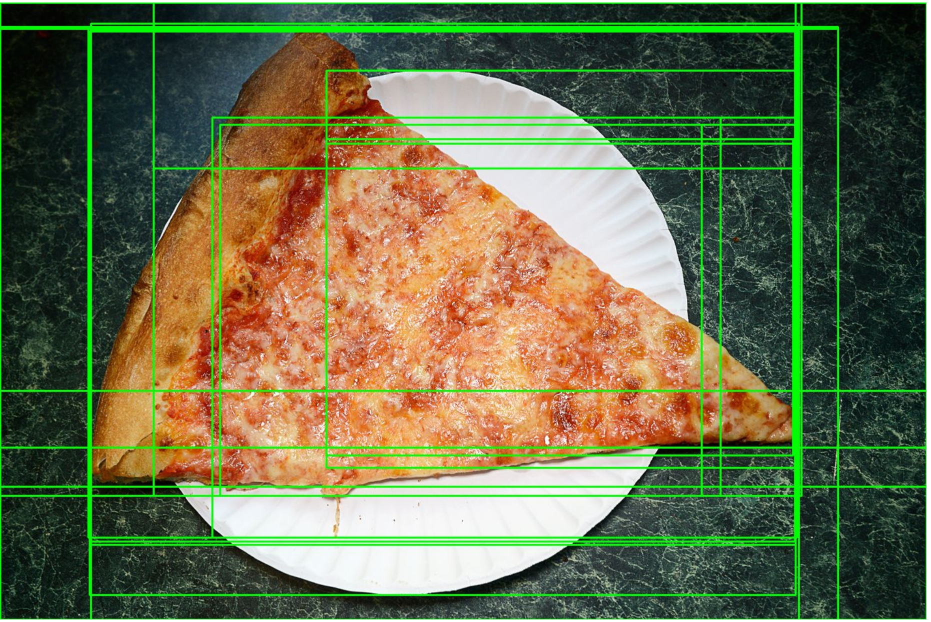
After NMS



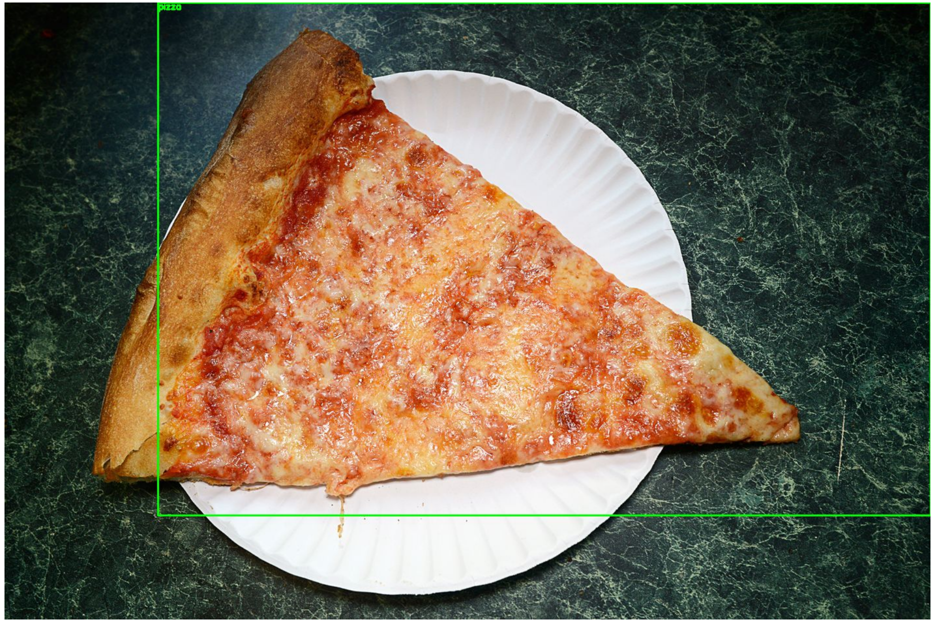
Before NMS



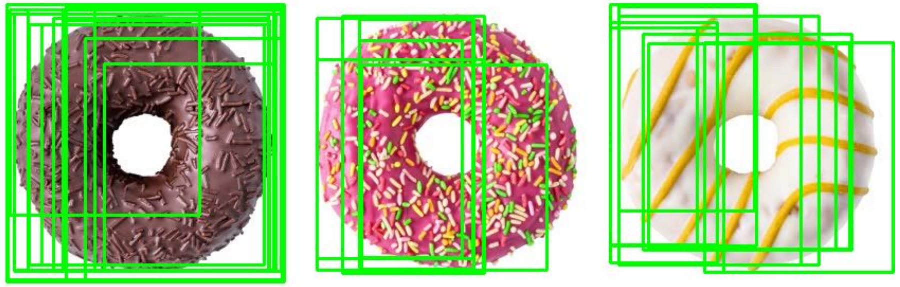
After NMS



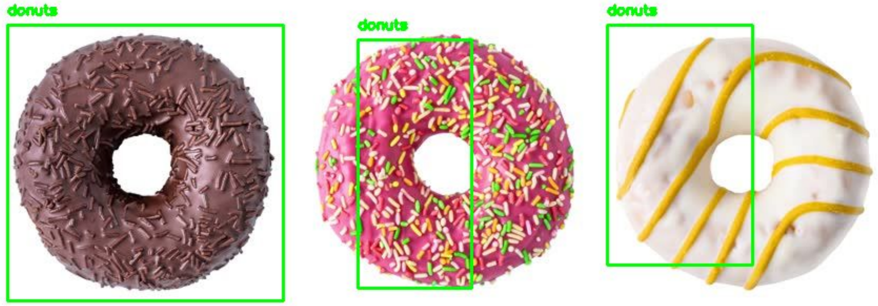
Before NMS



After NMS



Before NMS



After NMS

Conclusion

- We could successfully develop an end to end pipeline that does not require the tedious process of labeling the bounding boxes over every image.
- Selection search proved faster and more effective than other image processing algorithms like image pyramid and sliding window.
- From our comparative performance analysis, we can conclude that InceptionetV3 was the best model without memory constraints and Efficientnet B0&B1 could not reach an accuracy more than 93%. Also, if one needs to create an application that is time sensitive, one can use mobilenet architecture.
- Our encouraging results show that our pipeline can be used to generate datasets for training object detection models.