CSCI 677 Homework 2

Jingyun Yang

Student ID: 8357011225

1 Mean Shift Segmentation

1.1 Directory Structure

In this section, we present the code and results of the mean shift segmentor. First we explain the directory structure of our program. Under the program directory, `mean_shift.py` is the main python program that implements the mean shift algorithm, runs an experiment with a single setting on all images in the input directory, and saves image files to the output directory `out_ms`; `run_ms.sh` is a script file that runs `mean_shift.py` multiple times to carry out multiple experiments. `utils.py` is a python file containing useful helper functions for either mean shift segmentor, or selective search algorithm, or both. `HW2_ImageData` is the provided folder for input data and ground truth files. The directory tree, with `mean_shift` as the base directory name, is shown below.

```
mean_shift/
|-- mean_shift.py
|-- run_ms.sh
|-- utils.py
|-- out_ms/
|-- HW2_ImageData/
```

1.2 Source Listing

Now we list the source files that are used for the mean shift segmentor program.

```python
''' mean_shift.py '''
import argparse
import numpy as np
import cv2
import os
from utils import mkdir, show_img

''' Apply meanshift on a given image.

Args:
    img_path: path to the image.
    sp: spatial window radius for meanshift.
    sr: color window radius for meanshift.

Returns:
    image after meanshift, in RGB color space
'''
def apply_meanshift(img_path, sp, sr):
    # read image
    img = cv2.imread(img_path)

    # convert image to LAB color space
    img_lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
    img_filtered = img_lab

    img_filtered = cv2.pyrMeanShiftFiltering(img_lab, sp, sr, img_filtered, maxLevel=1)
```

```
    filtered_bgr = cv2.cvtColor(img_filtered, cv2.COLOR_LAB2BGR)

    return filtered_bgr

''' Performs a single mean shift experiment and outputs images to specified
    output directory.
'''
def main():
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--imgdir', type=str, default=None, help='directory to source images.')
    parser.add_argument('--outdir', type=str, default=None, help='directory to save output data.'
                                                                 )
    parser.add_argument('--sp', type=int, default=50, help='spatial window radius for meanshift.'
                                                           )
    parser.add_argument('--sr', type=int, default=50, help='color window radius for meanshift.')
    parser.add_argument("--show_img", help='show image results', action="store_true")
    args = parser.parse_args()

    exp_name = 'sp{}_sr{}'.format(args.sp, args.sr)
    mkdir(args.outdir)

    files = os.listdir(args.imgdir)
    for filename in files:
        if filename.endswith('jpg'):
            full_path = os.path.join(args.imgdir, filename)

            # apply meanshift on image
            res = apply_meanshift(full_path, args.sp, args.sr)

            cv2.imwrite(os.path.join(args.outdir, exp_name + '-' + filename[:-4] + '_ms.jpg'),
                                                                   res)

            if args.show_img:
                window_name = 'Meanshift Result for {}'.format(filename)
```

```
''' utils.py '''
import os
import cv2
import re

''' Safe mkdir that checks directory before creation. '''
def mkdir(dir):
    if not os.path.exists(dir):
        os.makedirs(dir)

''' Display an image in a window with given name. '''
def show_img(window_name, img):
    cv2.imshow(window_name, img)
    cv2.waitKey(0)
    cv2.destroyWindow(window_name)

''' Extract bounding boxes from a specified XML file. '''
def extract_bboxes_from_xml(filename):
    bboxes = []

    with open(filename, 'r') as f:
        content = ''.join(f.readlines())
        content = re.sub(r'\s+', '', content)
        bbox_re = re.compile('<bndbox>'
            + '<xmin>[0-9]*</xmin>'
            + '<ymin>[0-9]*</ymin>'
            + '<xmax>[0-9]*</xmax>'
            + '<ymax>[0-9]*</ymax>'
            + '</bndbox>')
        bbox_strings = bbox_re.findall(content)

        for bbox_str in bbox_strings:
```

```
            number_re = re.compile('>([0-9]+)<')
            bbox = [int(x) for x in number_re.findall(bbox_str)]
            bboxes.append(bbox)

    return bboxes

''' Converts bounding box format from xywh to x1y1x2y2 format. '''
def bbox_xywh_to_xyxy(bbox):
    x, y, w, h = bbox
    return [x, y, x + w, y + h]

''' Get area of a bounding box in xyxy format. '''
def get_area(bbox):
    return (bbox[2] - bbox[0]) * (bbox[3] - bbox[1])

''' Get overlap area of two bounding boxes, returns 0 if no overlap. '''
def get_overlap_area(bbox1, bbox2):
    x1 = max(bbox1[0], bbox2[0])
    x2 = min(bbox1[2], bbox2[2])
    y1 = max(bbox1[1], bbox2[1])
    y2 = min(bbox1[3], bbox2[3])
    if not (x1 < x2 and y1 < y2):
        return 0 # no overlap
    return (x2 - x1) * (y2 - y1)

''' Evaluate predicted bounding boxes using IOU.

Args:
    pred_bboxes: bounding boxes to be evaluated.
    gt_bboxes: ground truth bounding boxes.

Returns:
    precision: precision of the predicted bboxes.
    recall: recall value of the predicted bboxes.
    correct_bboxes: a list of correct bboxes (with iou > 0.5).
'''
def eval_iou(pred_bboxes, gt_bboxes):
    tp = 0
    correct_bboxes = []
    for gt_bbox in gt_bboxes:
        for pred_bbox in pred_bboxes:
            area_pred_bbox = get_area(pred_bbox)
            area_gt_bbox = get_area(gt_bbox)
            intersect_area = get_overlap_area(pred_bbox, gt_bbox)
            union_area = area_pred_bbox + area_gt_bbox - intersect_area
            iou = float(intersect_area) / union_area
            if iou > 0.5:
                tp += 1
                correct_bboxes.append(pred_bbox)
                break
    precision = tp / len(pred_bboxes)
    recall = tp / len(gt_bboxes)
    return precision, recall, correct_bboxes

''' Draw a list of bounding boxes in a given image with a color.

Args:
    bboxes: a list of bounding boxes.
    img: image to be annotated on.
    color: optional parameter specifying bounding box color.

Returns:
    img_with_bboxes: an image with annotated bounding boxes.
'''
def draw_bboxes_on_image(bboxes, img, color='g'):
    if color == 'b':
        color_tuple = (255, 0, 0)
    elif color == 'g':
```

```
        color_tuple = (0, 255, 0)
    elif color == 'r':
        color_tuple = (0, 0, 255)
    elif color == 'k':
        color_tuple = (0, 0, 0)

    img_with_bboxes = img.copy()
    for bbox in bboxes:
        x1, y1, x2, y2 = bbox
        img_with_bboxes = cv2.rectangle(img_with_bboxes, (x1, y1), (x2, y2), color_tuple, 1)
    return img_with_bboxes
```

```
: 'run_ms.sh'
IMGDIR='./HW2_ImageData/Images'
OUTDIR='./out_ms'

for sp in 2 8 32 128
do
  for sr in 2 8 32 128
  do
    python3 mean_shift.py --sp=$sp --sr=$sr --imgdir=$IMGDIR --outdir=$OUTDIR
  done
done
```

1.3 Results

Now we present the results of our program. We varied two parameters of the mean shift algorithm: spatial window radius (denoted in the rest of the section by `sp`) and color window radius (denoted in the rest of the section by `sr`). In our experiments, both `sp` and `sr` can take any of the following values: 2, 8, 32, and 128. We tested all combinations of the values listed above, and studied changes in program outputs. We hereby show two subsets of our results and state our findings.

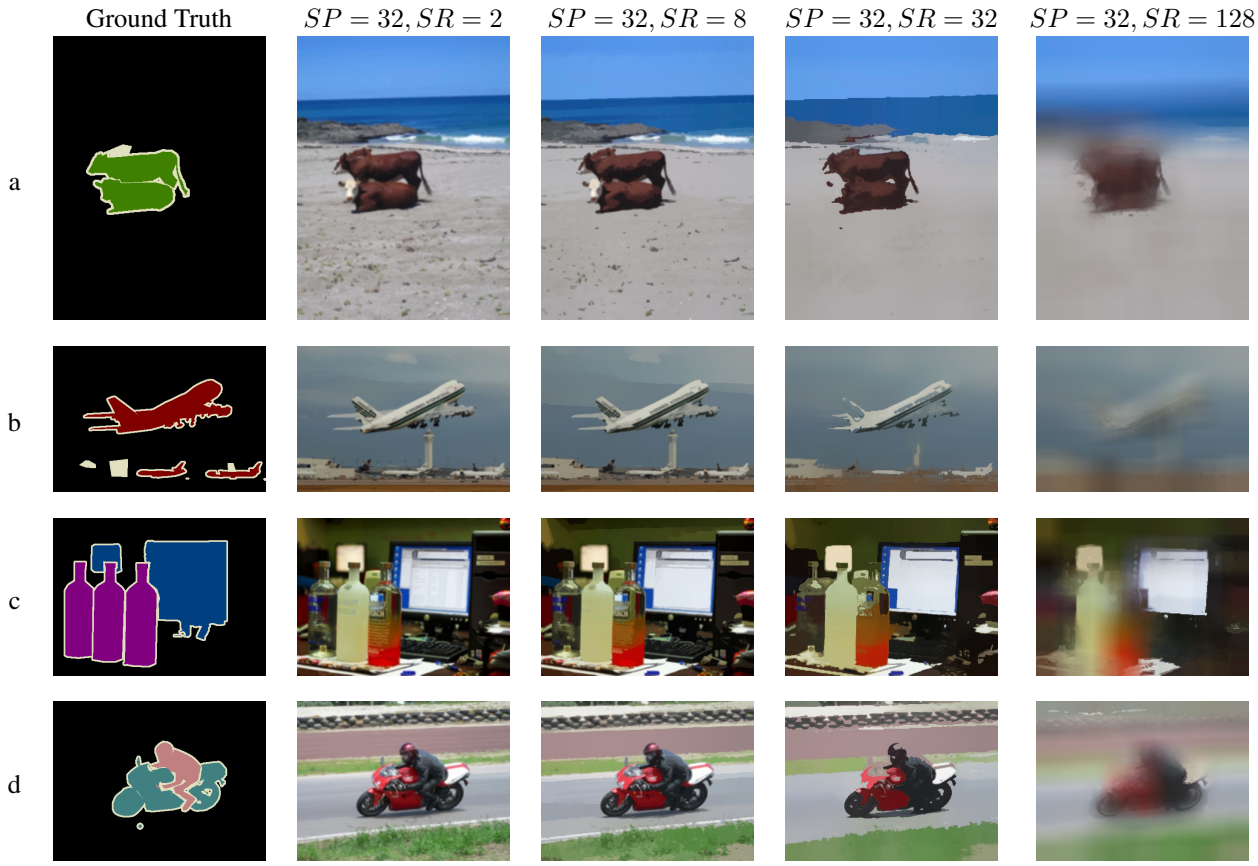

Figure 1: Results of Mean Shift Segmentor by Fixing $SP = 32$ and Varying SR

In Figure 1, we fix `sp` parameter to 32 and vary `sr` parameter to values 2, 8, 32, and 128. By comparing outputs of varying `sr` parameters on the same input image (aka. images in the same row), we can see that experiments using small `sr` values resemble the clear, original image a lot. On the other hand, experiments using very large `sr` values assigns near-average values in nearby regions of any given pixel, making the resulting image blurry. Comparing with the ground truth images, we see that we get result that most resembles the ground truth when using 32 as the `sr` parameter value.

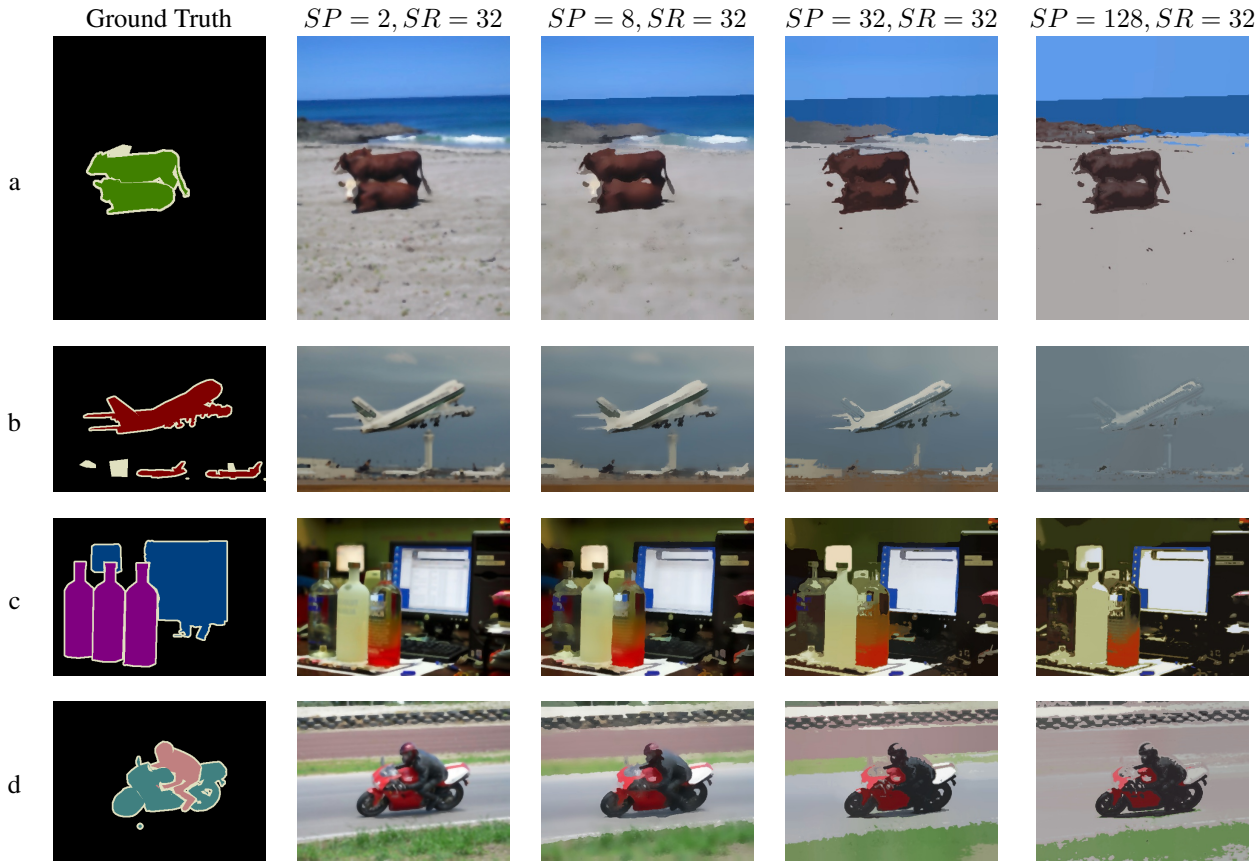

Figure 2: Results of Mean Shift Segmentor by Fixing $SR = 32$ and Varying SP

In Figure 2, we fix `sr` parameter to 8 and vary `sp` parameter to values 2, 8, 32, and 128. By comparing outputs of varying `sp` parameters on the same input image, we see that when the parameter is small, the output image looks very like the original image, which is not a good result for the purpose of segmentation. When we increase the `sp` parameter, the mean shift filter tends to set more pixels in the same ground truth segment to similar pixel values. When the parameter reaches the value of 32, the program can group pixels in the same ground truth segment to similar pixel values quite well. If we continue to increase the parameter, though, we can see that when `sp` reaches 128, the program starts to segment across ground truth segments, assigning pixels in nearby ground truth segments values close to the pixels in specific segments.

In summary, setting proper `sp` and `sr` values are important to achieve good segmentation result in the case of mean shift segmentor. When the parameters are too small, the output images resemble the original image and fails to perform the segmentation task; when the parameters are too large, the program will do "too much," blurring or making incorrect judgments on the image.

2 Selective Search

2.1 Directory Structure

For selective search, we run two experiments using two different similarity measures: color and multiple. We first describe the directory structure we used in the experiments. Under the program directory, `selective_search.py` is the main python program that implements the selective search algorithm, runs an experiment with a single setting on all images in the input directory, and saves image files with bounding box annotations to the output directory `out_ss`; `run_ss.sh` is a script file that runs `selective_search.py` multiple times to carry out multiple experiments. `utils.py`, `HW2_ImageData` are the same files as corresponding files described in the previous section.

The directory tree, with `selective_search` as the base directory name, is shown below.

```
selective_search/
|-- selective_search.py
|-- run_ss.sh
|-- utils.py
|-- out_ss/
|-- HW2_ImageData/
```

2.2 Source Listing

Now we list the source files that are used for the selective search program.

```
''' selective_search.py '''
import cv2
import numpy as np
import argparse
import re
import os
from utils import *

''' Applies selective search algorithm on a given image with specified strategy.

Args:
    img: provided input image.
    strategy: similarity strategy used by the selective search algorithm.

Returns:
    bboxes: a list of bounding boxes generated by selective search.
'''
def apply_ss(img, strategy):
    # define color strategies
    stra_color = cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyColor()
    stra_texture = cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyTexture()
    stra_size = cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategySize()
    stra_fill = cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyFill()
    stra_multi = cv2.ximgproc.segmentation.createSelectiveSearchSegmentationStrategyMultiple(
                                                    stra_texture, stra_color, stra_size,
                                                    stra_fill)

    ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()

    ss.clearStrategies()
    if strategy == 'color':
        ss.addStrategy(stra_color)
    elif strategy == 'texture':
        ss.addStrategy(stra_texture)
    else:
        ss.addStrategy(stra_multi)

    ss.setBaseImage(img)

    ss.switchToSelectiveSearchFast()

    # perform selective search and convert bboxes to xyxy format
    bboxes = [bbox_xywh_to_xyxy(bbox) for bbox in ss.process()]

    return bboxes

''' Main program that runs a single experiment and saves output files. '''
def main():
    # declare command line args
    parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
    parser.add_argument('--imgdir', type=str, default=None, help='directory to source images.')
    parser.add_argument('--gtdir', type=str, default=None, help='directory to ground truth xml
                                                    files.')
```

```
    parser.add_argument('--outdir', type=str, default=None, help='directory to save output data.'
                                                                )
    parser.add_argument('--max_rects', type=int, default=100, help='max number of boxes to show.'
                                                                )
    parser.add_argument('--strategy', type=str, default='multi', help='color strategy for ss.')
    parser.add_argument("--show_img", help='show image results', action="store_true")
    args = parser.parse_args()

    # name experiment and create output directory
    exp_name = '{}_{}'.format(args.strategy, args.max_rects)
    mkdir(args.outdir)

    # loop through all files in input directory and apply selective search algorithm
    files = os.listdir(args.imgdir)
    for filename in files:
        if filename.endswith('jpg'):
            # get ground truth boxes
            gt_path = os.path.join(args.gtdir, filename[:-3] + 'xml')
            gt_bboxes = extract_bboxes_from_xml(gt_path)

            # read image
            img_path = os.path.join(args.imgdir, filename)
            img = cv2.imread(img_path)

            # apply meanshift on image
            pred_bboxes = apply_ss(img, args.strategy)
            display_pred_boxes = pred_bboxes[:min(len(pred_bboxes), args.max_rects)]
            precision, recall, correct_bboxes = eval_iou(pred_bboxes, gt_bboxes)

            # add bboxes to image
            img_bbox = draw_bboxes_on_image(display_pred_boxes, img, color='g')
            img_correct_bbox = draw_bboxes_on_image(display_pred_boxes, img, color='g')
            img_correct_bbox = draw_bboxes_on_image(correct_bboxes, img_correct_bbox, color='r')

            # output annotated images
            cv2.imwrite(os.path.join(args.outdir, exp_name + '-' + filename[:-4] + '_bbox.jpg'),
                                                                img_bbox)
            cv2.imwrite(os.path.join(args.outdir, exp_name + '-' + filename[:-4] + '_correct_bbox
                                                                .jpg'), img_correct_bbox)

            # show image if requested
            if args.show_img:
                window_name = 'SS Result for {}'.format(filename)
                show_img(window_name, img_bbox)

            # print experiment result
            print('Experiment {}, file {}: p={}; r={}'.format(exp_name, filename, precision,
                                                                recall))

if __name__ == '__main__':
    main()
```

```
: 'run_ss.sh'
IMGDIR='./HW2_ImageData/Images'
GTDIR='./HW2_ImageData/boundingbox_groudntruths'
OUTDIR='./out_ss'

python3 selective_search.py --max_rects=100 --strategy=color --imgdir=$IMGDIR --gtdir=$GTDIR --
                                                outdir=$OUTDIR
python3 selective_search.py --max_rects=100 --strategy=multi --imgdir=$IMGDIR --gtdir=$GTDIR --
                                                outdir=$OUTDIR
```

2.3 Results

We now present the result of our program.

Figure 3: Bounding Box Outputs by Selective Search Algorithm

Experiment	Precision	Recall
Color, Image 1	0.0015	1.0
Color, Image 2	0.0030	1.0
Color, Image 3	0.0022	1.0
Color, Image 4	0.0015	1.0
Multi, Image 1	0.0015	1.0
Multi, Image 2	0.0030	1.0
Multi, Image 3	0.0022	1.0
Multi, Image 4	0.0015	1.0

Table 1: Adapted Behaviors Distance (ABD), the lower the better

Figure 3 shows the result of our experiments, in which we apply two similarity measures on the four input images. We display 100 generated bounding boxes in each output annotated image in green. For each ground truth bounding box that has a corresponding output bounding box with IOU of at least 0.5, we draw one corresponding output bounding box in the image in red. The precision and recall values of the experiments are shown in Table 1.

By inspecting the results, we can see that the recall values of the program is extremely high. Actually, all the experiments end up finding all the target bounding boxes of interest. On the other hand, the precision of the program is very low. This happens because the selective search algorithm proposes a lot of bounding boxes, only a small portion of which actually match the ground truth bounding boxes with IOU values of at least 0.5. Also, we can see that using color strategy and all strategies don't induce different bounding box outputs, and their precision and recall values are the same.