Package of Evaluation Framework for Range Image Segmentation Algorithms

Jaesik Min
Department of Computer Science and Engineering
University of South Florida
jmin@csee.usf.edu

1 Introduction

Performance evaluation of computer vision algorithms has received increasing attention in recent years. We have developed an automated framework for objective performance evaluation of region segmentation algorithms. The framework should be applicable to other types of imagery for which proper ground truth can be reliably specified.

We distribute a package of the automated evaluation framework. The package has several features. It minimizes human intervention by dispensing with manual operations in the process of training. Users just need to provide minimum information in specified configuration files. Its text-based I/O enables the framework to run under any version of UNIX system.

By using our evaluation framework, authors of a segmentation algorithm can compare their algorithms' performance result to the baseline algorithm to show their performance over that of the baseline algorithm. Other than the evaluation purpose, users can just train their segmentation algorithms on different environments by using the automated parameter training function. Our further research will enable the algorithm developers to analyze their algorithms and eventually design improved algorithms.

This document is composed of several sections. Section 2 illustrates how the framework works and explains the basic concept of each element of the framework. Section 3 shows the contents of the package, system requirements, what the users need to know before using the framework, and how to compile and run the program, etc. Section 4 provides some tips and precautions to help users run the program. Finally, Section A explains every element of the package in detail.

2 Evaluation Framework

2.1 Overview

With a set of train images along with proper ground truth specifications, and clear definition of a scoring function, we are ready to train a segmentation algorithm. The training function consists of two parts; parameter tuning and validation. The training repeats the segmenter parameters until it

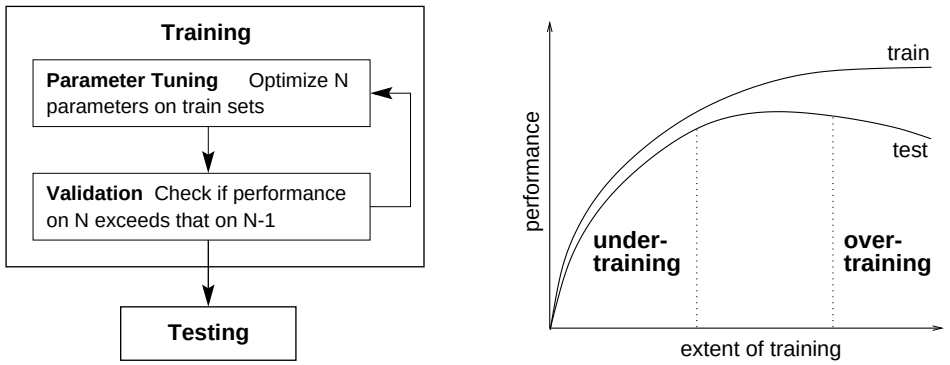


Figure 1: (a) Overview of evaluation framework. Parameter tuning repeats until no significant improvement is found on validation tests. (b) Hazard of over-training. Beyond some extent of training, the performance of testing gets worse. One key insight here is that the process of selecting the appropriate number of parameters is part of the overall training process.

gets no more significant improvement on validation tests. The final training results are ready for testing. (Figure 1 (a))

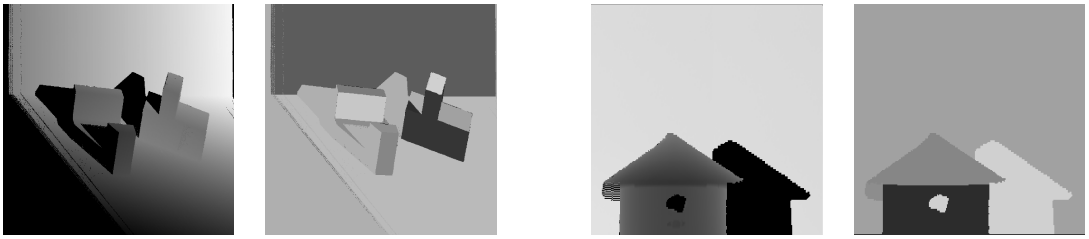
The reason that we have employed a validation step is to avoid an undesirable result known as over-training. Since the training performance always gets better — at least not worse — as the extent of training increases, the training result might be too dedicated to the train sets far over the proper training level. The effect often shows up as degraded performance results on test sets as shown in Figure 1 (b). So when to stop the training is an issue. We try to avoid the hazard of falling into over-training by testing the tuned parameter set on our validation data sets, which can be thought as pseudo test data sets, every time the extent of training is at decision.

The final training results in the form of parameter settings — one for each train set — that have undergone the validation tests are to be fed into the testing process, which does exactly the same function as the validation test but this time with test sets instead of validation sets.

2.2 Image Data Set

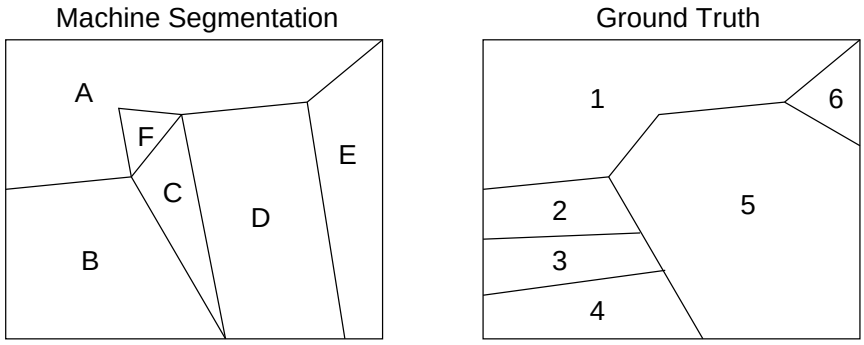
We provide two types of range image data: ABW for planar surface scenes and Cyberware for curved surface scenes. We assume that five different types of curved surfaces can exist in the curved surface scenes: plane, cylinder, cone, sphere, and torus. We prepared 40 ABW images and 40 Cyberware images along with their ground truth images constructed by pixel-level manual specification. (Figure 2)

We divided those 40 images into three groups; 14 images for train pool, 13 for validation pool, and the other 13 for test pool. From each pool, we have drawn 10 sets of images, each set having six images. The 10 train sets, 10 validation sets, and 10 test sets are provided in this package for training and testing.



(a) ABW range image and GT (b) Cyberware range image and GT

Figure 2: Pairs of a range image and its ground truth specification of planar surface (ABW) and curved surface (Cyberware) scenes



MS region A corresponds to GT region 1 as an instance of correct segmentation.
GT region 5 corresponds to MS regions C, D, and E as an instance of over-segmentation.
MS region B corresponds to GT regions 2, 3, and 4 as an instance of under-segmentation.
GT region 6 is an instance of a missed region.
MS region F is an instance of a noise region.

Figure 3: Illustration of definitions for scoring region segmentation results. There exist several overlap thresholds in determining the correct segmentation, which are not shown in this figure.

2.3 Scoring of Performance Metrics

A machine segmentation (MS) of an image can be compared to the ground truth (GT) specification for that image to count instances of correct segmentation, under-segmentation, over-segmentation, missed region, and noise region. (Figure 3) These definitions are based on the degree of mutual overlap between a region in the MS and a corresponding region in the GT. Currently, when we mention a performance we refer to the percentage of correct segmentation instances only.

A comparison tool has been made to compare MS regions and GT regions at a specified overlap threshold and automatically score the results. We preset ten overlap thresholds as $\{0.51, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95\}$. Comparisons with these thresholds yield 10 scores in terms of percentages of correct segmentation instances. Then we are able to draw a performance curve from which a quantitative performance value — so-called area under the curve(AUC) — is scored.

2.4 Tuning of Segmenter Parameters

The automated training procedure currently used is a form of adaptive search. It operates as follows. Assume that the number of parameters to be tuned, and the plausible range of each parameter, are

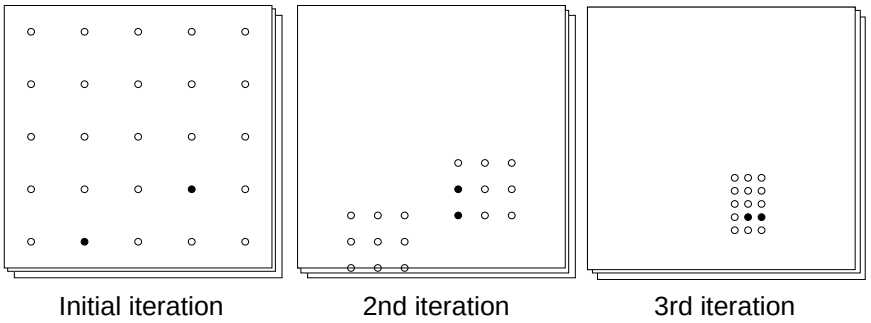


Figure 4: Illustration of the adaptive sampling of 2-parameter space in parameter tuning process. Filled-in points represent highest performing parameter settings to be carried forward.

specified. The range of each parameter is sampled by 5 evenly-spaced points. If D parameters are tuned, then there are 5^D initial parameter settings to be considered. The segmenter is run on each of the training images with each of these 5^D parameter settings. The segmentation results are evaluated against the ground truth using the automated scoring function described in Section 2.3. The highest performing one percent of the 5^D initial parameter settings, as ranked by area under the curve(AUC) are selected for refinement in the next iteration. The refinement in the next iteration creates $3 \times 3 \times \cdots \times 3$ sampling around each of the parameter settings carried forward. (Figure 4)

In this way, the resolution of the parameter settings becomes finer with each iteration, even as the total number of parameter settings considered is reduced on each iteration. The expanded set of points is then evaluated on the training set, and AUCs again computed. The top-performing points are again selected to be carried forward to the next iteration. Iteration continues until the improvement in the AUC drops below 5% between iterations. Then the current top-performing point is selected as the final parameter setting with the current number of parameters. The execution format of the parameter tuning is as follows.

```
segtrain -c configfile -d dimension
example) segtrain -c eval.config -d 3
```

2.5 Validation of Trained Parameters

The validation test is performed by testing the train results on the validation sets and finding any statistically significant improvement in performance. The testing command, `segtest`, has following execution format, which is similar to that of `segtrain`.

```
segtest -c configfile -d dimension
example) segtest -c eval.config -d 3
```

On completion of the parameter tuning on m train sets, users will have m parameter settings, which will be used as part of the inputs to the validation test. Other inputs to the validation test are the validation image sets. That is, m parameter settings that came from parameter tunings on m train sets are to be applied to n validation sets. The original form of each individual validation test result is a performance curve along different threshold values of region overlap tolerance. In

determining the improvement of performance, we compare the quantitative form of the curve. After the validation test has been done the users will have a total of $m \cdot n$ AUC values.

At every termination of parameter tuning with a given number of parameters, the training program has to determine if any performance improvement has been obtained from the latest parameter tuning. A statistical sign test is carried out with significance level at $\alpha = 0.05$ in normal approximation of binomial distribution. If current performance turns out to be significantly better than that of with fewer parameters, the parameter tuning will resume with additional parameters. Otherwise, the whole training will be finished and the result is final. The execution format of the sign test is as follows.

```
signtest -c configfile -d1 dimension -d2 dimension
example) signtest -c eval.config -d1 2 -d2 3
```

2.6 Segmenter Requirements

We recommend that the users give a minor change to their segmentation algorithms so that it accepts a range image and a parameter file as its input arguments and, at termination, creates a machine segmented image as output result. The recommended execution format of a segmenter is as follows.

```
segmenter -r rangeimage -p paramfile -m MSimage
example) segmenter -r ABW/house -p paramfile -m ABW/house
```

The `rangeimage` is the file prefix of the input range image. The image file is supposed to have `.range` as its suffix. Currently, our framework deals with image files of PGM format only. There should exist a corresponding ground truth image with the same file prefix but with different suffix (`.gt-seg`) just like those enclosed in this package.

As we mentioned above, we recommend that users structure their algorithms' I/O format so that the parameter values are given to the segmenterin the form of a file. Let us call it `paramfile`. Each line of this file represents the value of a parameter. An example of a `paramfile` for a 7-parameter segmenter is given below.

```
3.0
180
4.5
0.12
800
7
12.8
```

Every range segmentation algorithm has a different number of parameters and the number of parameters to be tuned can be predefined. We put no limit on the number of parameters to be tuned. However, since tuning over 5 or 6 parameters of a typical algorithm implies that a huge number of parameter settings are to be tested, it is practical to train the first several significant parameters only. Therefore, it is better to list the parameters in the order of significance in the `paramfile`. Users can set the order of parameters in the `param-config-file`, which is described later in Section A. Users need not worry about creating the `paramfile`. When the segmentation algorithm is being trained the `paramfile` will be created by the training algorithm automatically based

on the specifications of `param-config-file` and its contents will be modified accordingly every time the segmenter runs with a new parameter setting. What users need to do is to create the `param-config-file`. Our training program also names the `paramfile` with a 20-character random string starting with "_". Therefore, please do not remove it in the middle of training process.

When an execution of the segmenter on an input image with a parameter setting is finished, the segmentation result (MS image) will be created with the same file prefix as the `rangeimage` but with different suffix (`.ms-seg`). This image file also is in PGM format.

2.7 Baseline Segmentation Algorithm

In this package you will find two segmenters that were originally developed in University of Bern, Switzerland, and slightly changed to fit to our evaluation framework. We enclose their source codes so that users can use them as the baseline algorithms. The first one, `UBC`, deals with curved surface scenes and will be the baseline segmenter on Cyberware images. The other one, `UBP`, was designed to handle planar surface scenes only and will be the baseline segmenter on ABW images.

We also enclose these baseline segmenters' training results obtained at our site. This is to pretest the user's training environment. Some users' system environment might be different from ours and ignoring this may cause unexpected problems. Therefore we recommend that, as a prerequisite to the training of the user's own segmenter, the user train these baseline segmenter by using our automated training program and check if the train results are the same as those enclosed in the package.

2.8 How long will it take?

It depends on various factors such as the speed of segmenter, the number of images in the train set, the number of train sets, the number of parameters being tuned, and so on, but usually the automated training itself is a time-consuming procedure. For example, if we are about to tune 3 parameters of a segmenter on 10 train images, $5^3 \times 10 (= 1250)$ executions of the segmenter are needed just in the initial step. In case of 4 parameters the number of initial segmenter executions goes up to 6250.

In the latest version, we have made it possible to run the training program on multiple machines that are under DFS(Distributed File System) environment. The heavy job load will be shared among those machines without conflict or redundancy. When an additional machine becomes available later, users can add it to the working machine group even in the middle of the training process. The synchronization between processes in the working group is accomplished by creation and deletion of a number of semaphore files under the working directory. Therefore, make sure that the processes are launched under the same directory even though they are on different machines.

3 How to Use Package

In developing the framework, one of our goals was to make it as generic as possible so that it is easy to run by any user who uses the UNIX environment. Intermediate results are displayed in text mode. Following are how to unpack, verify, and use our package.

1. Unpack.

```
gzip -d evalpack.tar.gz
tar -xf evalpack.tar
```

The following directories and files will be created.

```
CW/     Cyberware range, GT images, calibration files
        lists of train, validation, and test sets
ABW/    ABW range, GT images, calibration files
        lists of train, validation, and test sets
UBC/    UB curved surface segmenter source codes
        eval.config, param.config
UBP/    UB planar surface segmenter source codes
        eval.config, param.config
train/  training source codes
```

Uncompress all `*.Z` files under `CW/` and `ABW/` directories.

```
[under ABW/ or CW/] uncompress *.Z
```

2. Compile the baseline segmenter.

```
[under UBC/ or UBP/] make
```

Compile the training source code.

```
[under train/]
 gcc -o compare compare.c
 make scrpt
 make segtrain
 make segtest
 make signtest
```

3. Make a working directory and train the baseline segmenter. (Edit `eval.config` if necessary)

```
scrpt -t ../../train/ -c eval.config -d 4
```

4. Compare your results of baseline segmenter to those performed at our site (stored under `UBC/usf/` and `UBP/usf/`). If you see any difference between them, contact us.

5. Train your segmenter. That is, create or edit `eval.config` and then repeat step 3 here.

6. Compare results to baseline training results from step 4.

7. If train results do not improve on baseline, re-design algorithm.

8. Repeat step 5 and 7 until done.

9. Run test process and report the result.

4 Miscellaneous

When the training is under way, the training program creates several temporary files with random names (e.g., `_Vg93KGop0I3ef7Dt62M`) under the working directory and several directories with similar names under `ms-images-path`. Some files with `.semfile` suffix will also be created to act as semaphores. On completion of the whole training process, these temporary files and directories will be deleted automatically.

Once the training has been launched, there is no way to hold or cancel it without ruining the intermediate results. If the training ends without completion due to some erroneous reasons, restart the process. Currently our framework does not support *resume* function so users have to restart the whole process even if intermediate results are present. When restarting the process, make sure that the previous results are completely removed or displaced. The presence of intermediate or final results will affect the flow of training process.

A Appendix

A.1 Training

The training command, `scrpt`, simply repeats the execution of `segtrain`, `segtest`, and `signtest` up to designated times as long as the performance result gets improvement. It has the following execution format:

```
scrpt -t traindir -c configfile -d maxdim
example) scrpt -t ../../train/ -c eval.config -d 4
```

where `traindir` is the directory where `segtrain`, `segtest`, and `signtest` reside, `maxdim` is the maximum number of parameters to be tuned, and `configfile` is the configuration file that contains a total of 15 definitions that are required during the execution of `segtrain`, `segtest`, and `signtest`. An example contents of `configfile` is given in Figure 5. In the following subsections, each line of the file is explained in detail. Users may create a new file or change the contents of enclosed sample file (e.g. `eval.config` under the baseline segmenter directories).

Note that the training program simply looks for colons (:) to locate reading pointers. So you can omit strings on the first column, but do not remove colons. Also note that the ordering of lines should not be changed. The whole training is a time-consuming process. So it is recommended to launch multiple trainings on different machines simultaneously, if possible.

A.1.1 `range-images-path`

Path name where the input range images reside.

A.1.2 `gt-images-path`

Path name where the input GT images reside.

```
range-images-path :       ../../ABW/
gt-images-path :          ../../ABW/
ms-images-path :          ../../ABW/
segmenter :               ../segn
compare-tool :            ../../train/compare
param-config-file :       ../param.config
number-of-train-sets :    10
number-of-val-sets :      10
train-image-set :         ../../ABW/trnset
validation-image-set :    ../../ABW/vldset
train-result-file :       trainresult
train-log-file :          trainlog
validation-result-file :  vldresult
stat-result-file :        statresult
eval-result-file :        evalresult
```

Figure 5: Sample contents of an evaluation configuration file. It should be created or modified by users before launching the training process.

A.1.3 `ms-images-path`

Path name where the output MS images will be saved after every execution of the segmenter. To avoid any possible conflict in case of distributed execution of training, the training program creates a temporary directory under this path and uses this directory to store MS images.

A.1.4 `segmenter`

Executable path/file name of the segmenter.

A.1.5 `compare-tool`

Executable path/file name of the automated segmentation scoring program.

A.1.6 `param-config-file`

This file should be built by users before training. It should have domain information of all parameters of the segmenter. Each line should have the following configuration.

```
comment :  type  default  minimum  maximum  interval
```

A sample contents of this file is given in Figure 6. Again, note that our trainer simply looks for colons (:) to locate reading pointers. So you can omit the optional comments before the colons but colons themselves. The `minimum` and `maximum` values indicate the range of a parameter value. The `interval` column is an integer value that indicates the displacement of integer parameters. It is introduced to handle *window size* parameters and void for floating-point parameters.

It is recommended that these parameters are listed in the order of significance. Among these parameters, only the first D ones (as selected by option `-d` of the `segtrain`) are selected as variable parameters; other parameters will have their `default` values.

```
1ST PARAMETER :  float  0.45  0.1   1.0   0
2ND PARAMETER :  int    200   30    800   1
3RD PARAMETER :  float  8.0   4.0   12.0  0
4TH PARAMETER :  float  0.1   0.01  1.0   0
5TH PARAMETER :  int    3     3     13    2
6TH PARAMETER :  float  0.3   0.1   1.0   0
```

Figure 6: Sample contents of a parameter configuration file. Users should specify the contents before launching the training process.

A.1.7 number-of-train-sets

The number of training image sets. This package has 10 six-image training sets. Users need not worry about the number of images in each train set.

A.1.8 number-of-val-sets

The number of validation image sets. This package has 10 six-image validation image sets. Again, users need not worry about the number of images in each train set.

A.1.9 train-image-set

This file set is provided in our package. This package has 10 train image sets whose names are referred by adding extensions to this file set name (e.g., `trnset-0, trnset-1, ... , trnset-9`). It consists of a list of train image file names. (Prefixes only. Paths and suffixes will be attached by the trainer.) An example contents is shown below.

```
tjunction
conecyls
pipebend2
buoy
cone2
funnel
```

A.1.10 validation-image-set

This file set has the same form as the `train-image-set` does, except that the images come from a different image pool.

A.1.11 train-result-file

At completion of parameter tuning, the `segtrain` will create one train result file for each train set that contains training results, such as the best parameter setting, number of segmenter executions, etc. (Figure 7) Their names are specified by adding extensions to this file name. For example, train result of 2-parameter tuning on train set #4 will be named as `trainresult-d2-s4`.

```
NUMBER OF TRAIN IMAGES: 6

TRAIN IMAGE LIST:
tjunction
conecyls
pipebend2
buoy
cone2
funnel

NUMBER OF PARAMETERS TRAINED: 2

PARAMETER VALUE RANGES GIVEN:
0.100000 1.000000
0.010000 1.000000

PARAMETER SETTING TUNED: 0.550000 0.505000
TOTAL NUMBER OF SEGMENTER EXECUTIONS: 204

Begining Time: Fri Oct  5 16:19:40 2001
Ending   Time: Fri Oct  5 16:31:36 2001
```

Figure 7: Contents of a train result file. This file is created by the training process and also is used as an input to the validation function.

A.1.12 `train-log-file`

The history of the whole training process will be given through to a log file. The naming method of log files are same as `train-result file`. It lists all the parameter settings tried, along with their performance values. (Figure 8)

A.1.13 `validation-result-file`

After each validation ends, a file will be created. For example, a file `vldresult-d2` will be created right after a validation procedure on training result with 2-parameter tuning ends. The contents contain the validation results in the form of $m \cdot n$ AUC values. These values are to be fed as one of the inputs to the sign test.

A.1.14 `stat-result-file`

Once a sign test is completed, the result of statistical significance in performance will affect the flow of evaluation framework. As long as any improvement is reported, the training continues with an extended number of parameters. On completion of each sign test, the result will be recorded in the `stat-result-file` with an appropriate suffix. For example, `statresult-d2-d3` will be created right after a sign test on `vldresult-d2` and `vldresult-3` ends.

A.1.15 `eval-result-file`

When no more statistically significant performance improvement has been found, the training stops and the final training results — m parameter settings — will be stored in this file.

```
LOG FILE
junction (  0.5500 ) 285.7143
junction (  0.1900 ) 285.7143
junction (  0.3700 ) 285.7143
junction (  0.7300 ) 285.7143
junction (  0.9100 ) 285.7143

cyls1 (  0.5500 ) 485.7143
cyls1 (  0.1900 ) 525.7143
cyls1 (  0.3700 ) 485.7143
cyls1 (  0.7300 ) 485.7143
cyls1 (  0.9100 ) 485.7143

torus1 (  0.5500 ) 719.0475
torus1 (  0.1900 ) 759.0475
torus1 (  0.3700 ) 552.3809
torus1 (  0.7300 ) 719.0475
torus1 (  0.9100 ) 685.7142

...
```

Figure 8: Contents of a train log file. This file is created by the training process automatically. The first column is the name of image, the second column (inside the parentheses) shows the value of parameter(s) being tuned, and the third one is the performance result.