

Computer Vision, A.Y. 2020-2021, Spring semester
Report on Final Project (Detection of Boats in an Image)

Luca Badin 1242060

Project outline

The boat detector I developed is based on a Bag of Words approach with Support Vector Machine classification. Three classes performing three distinct functions are present:

- **Trainer**, which has methods to build a dictionary and train a SVM on the supplied training set;
- **Segment**, which performs segmentation of an input image in preparation for inference;
- **Detector**, which performs inference on each extracted segment using the SVM, and can additionally evaluate the performance of a detection attempt given the ground truth file for the test image.

A brief outline of the three different functions/phases follows, followed by results on test images, a few comments on performance and improvements and instructions for running the program itself.

Training the detector

Training is formed by two distinct phases: building a Bag of Words dictionary that the detector uses to describe the content of an image, and training a Support Vector Machine that separates features found in boats from features not found in boats, enabling the actual inference process.

Building a dictionary. The dictionary building process collects a list of descriptors from every sample from the training set. This does not discriminate between boats and other objects, since we're aiming to tell the program what a whole image will look like. Then, all these descriptors are clustered using K-Means into a certain number of clusters (1000 was picked empirically, from lower counts of 100 and 500 that were attempted earlier, to try to balance between too many types of features and too few.)

The clustering process is rather time consuming, and on my machine the whole process took about half an hour for a training set of about 3000 samples. The storage of descriptors also takes a considerable amount of memory.

Training a SVM. The SVM training process similarly needs to collect a list of descriptors from every sample from the training set, but this time positive samples (those belonging to boats) are separated from negative samples (those belonging to every other object).

Positive samples are defined by the ground truth files, and therefore for each boat contained in each training sample, SIFT descriptors are extracted, cast into dictionary words based on the dictionary we built earlier, and collected. Negative samples are similarly extracted from the parts of the images that do not belong to boats, made into words and collected.

A SVM is finally trained on these samples, in a two-class classification setting (positive samples are labeled "1", negatives "0"). The SVM uses an RBF kernel.

Segmentation

The segmentation process aims to extract candidate objects from the input image; each segment is then to be fed into the detector.

Preprocessing. The preprocessing that is made before actual segmentation begins consists of bilateral filtering, meanshift and a Laplacian filter. These attempt to, respectively:

1. smooth out noise while preserving borders;
2. quantize the color levels and perform some sort of color pre-segmentation;
3. enhance the borders.

Meanshift is by far the step in the whole segmentation chain that takes the most time, due to high parameters of drift physical space and color space radii; but not using meanshift (and instead employing eg. stronger bilateral filtering) is quite ineffective, and lower parameter values have also caused poor results.

Floodfill centroid generation. The actual segmentation is performed via thresholding and the `floodFill` function. This function requires a set of starting centroids from which to begin filling. Thus, on a normalized grayscale version of the preprocessed input:

1. the image is quantized in 16 intensity levels, where each level will have a number of connected components;
2. for each level, connected component contours are extracted and their centroids are computed (but only if the area of the component is above a certain threshold).

All centroids are accumulated together regardless of the intensity level at which they were computed.

Large segment extraction. For each centroid, the floodfill algorithm is performed on the normalized grayscale input to extract a binary map, which ideally represents a connected blob of similar intensity. Such map is only retained if its area is above a certain threshold (1% of the image).

Then, since some centroids will end up in the same area and produce identical maps, they are checked for and discarded.

At the end of this step, most of the image will be segmented in large chunks (segments), each defined by a binary mask. Testing shows this usually results in very good segmentations for water, trees, sky and uniform parts of a boat's hull.

Contiguous small segment extraction. At this point, thresholding has left some small areas unclustered, and the large segment extraction has also potentially resulted in relatively large areas, made up of contiguous small segments that were discarded earlier, entirely surrounded by large well segmented portions that were retained. Such contiguous uneven areas might be detailed parts of boats, which are likely bounded by well-segmented sky and water.

We can exploit this to extract parts of the image which would be hard to group together earlier than this step (because they belong to the same object, and therefore they are contiguous, but have very different intensities).

1. A "mask of small parts" is built by adding together all binary maps, and then taking the negative;
2. This mask is eroded, to make sure that loosely connected components are eliminated;
3. Connected components are extracted via contour computation, and those higher than a threshold (5% of the image) are retained and made into new binary masks defining new segments. Note that the threshold is higher than the previous one, since we are fine with ignoring very small uneven components that did not make the more lenient cut earlier.

At this point, the segmentation is terminated.

Detection/Inference

Given the output of the segmentation, a list of binary masks defining areas of the image, we then proceed to use the SVM we trained to hopefully decide whether or not a boat is present in the segment or not.

For each segment, descriptors belonging to the area of the image defined by the segment's binary mask are computed, cast into dictionary words and evaluated by the SVM. Then, if the detection is successful, the bounding box of the binary mask is computed and drawn on the result image.

If a ground truth file for the input image is supplied, the detection is scored using the Intersection over Union metric: this is done by building a binary mask for the real boats defined by the ground truth file, another mask for the boats detected by the algorithm, and then computing the area of their intersection (bitwise and) over the area of their union (bitwise or).

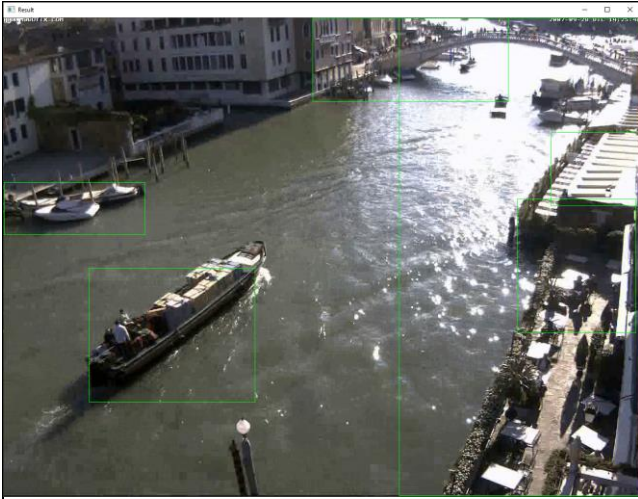
Detection is pretty fast by itself; it is strongly hampered by the slow meanshift algorithm.

Results and potential improvements

The results for the Venice and Kaggle datasets follow, with a commented sample of images.

Sample	IoU
venice/00.png	0.1592
venice/01.png	0.0374
venice/02.png	0.0217
venice/03.png	0.1548
venice/04.png	0.2025
venice/05.png	0.2042
venice/06.png	0.4946
venice/07.png	0.0737
venice/08.png	0.0995
venice/09.png	0.3204
venice/10.png	0.2066
venice/11.png	0.1291

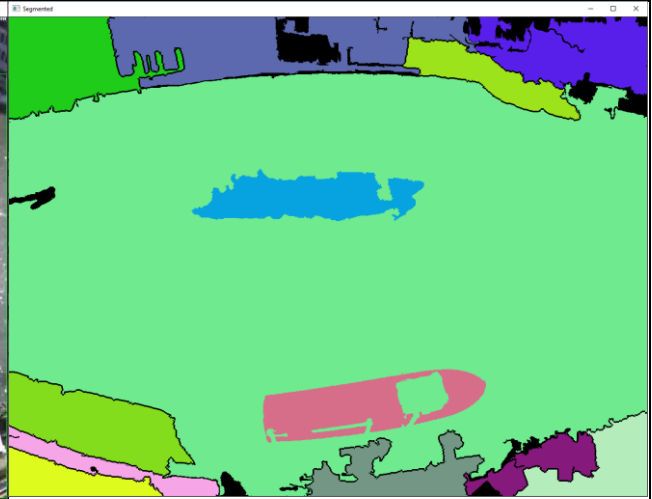
Sample	IoU
kaggle/01.jpg	0.6773
kaggle/02.jpg	0.0062
kaggle/03.jpg	0
kaggle/04.jpg	0.1454
kaggle/05.jpg	0.0021
kaggle/06.jpg	0
kaggle/07.jpg	0
kaggle/08.jpg	0.0084
kaggle/09.jpg	0.0741
kaggle/10.jpg	0.8247



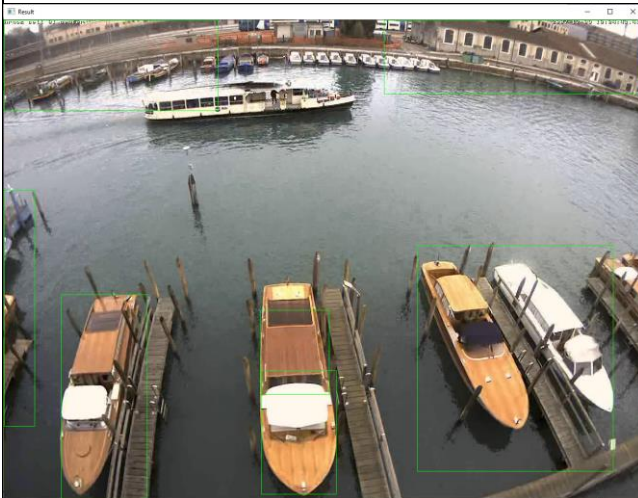
venice/00.png: Result was penalised by a large bounding box spanning the whole right side of the image. Boats in the foreground correctly detected.



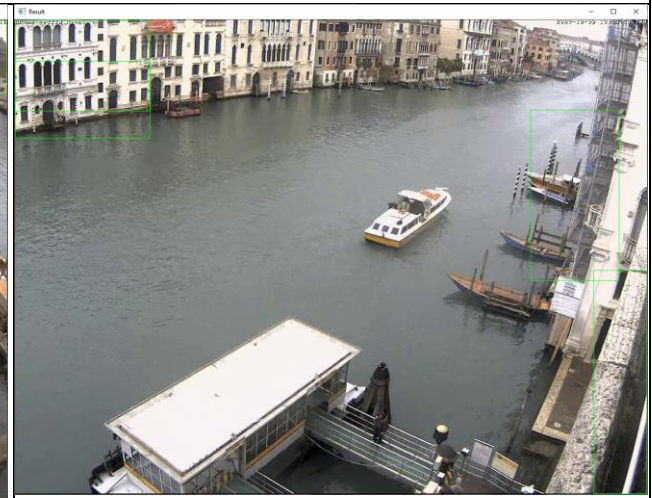
venice/03.png: Façade and window fooling the detector. Low contrast boat was neither segmented by itself, nor detected as a boat in another segment.



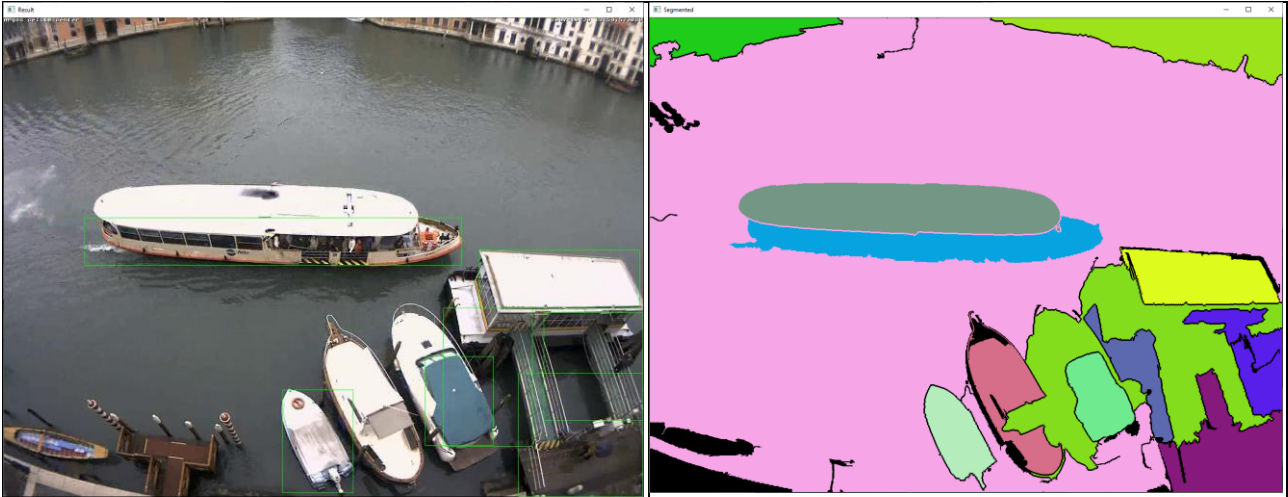
venice/04.png result and segmentation: Again many clusters not containing boats are marked as containing boats, and the fact the water segmentation picks up the white boat makes the whole water area marked as a positive hit. The boat on the bottom left is very well segmented but not recognised as a boat.



venice/07.png. Very bad result: boat is actually very well segmented, but not correctly detected. More instances of buildings fooling the detector.

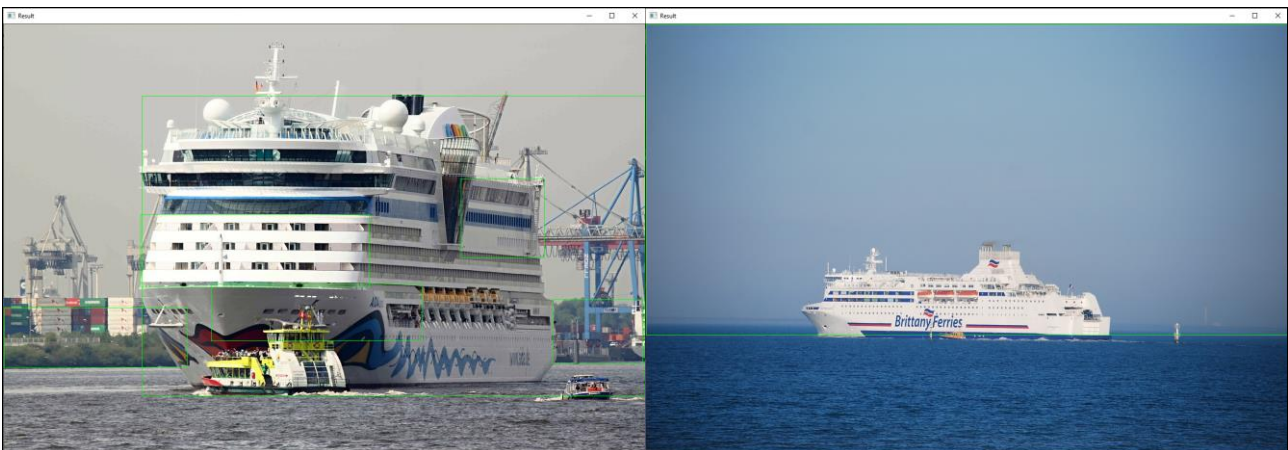


venice/07.png. Very bad result: boat is actually very well segmented, but not correctly detected. More instances of buildings fooling the detector.






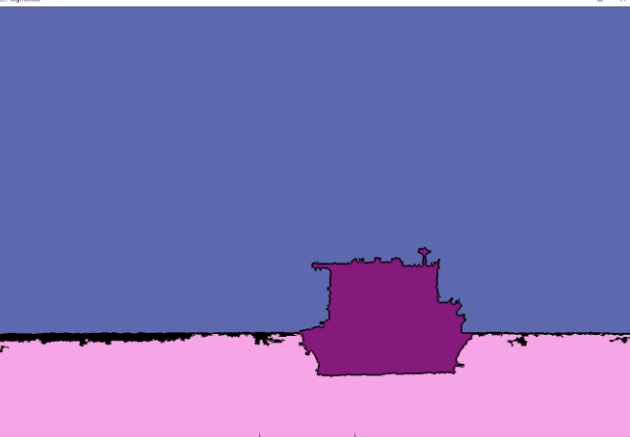
venice/09.png result and segmentation. Segmentation not so bad, good enough result on water, buildings and boats, and for once detector not fooled by buildings; but it doesn't detect the low middle boat and is completely thrown off by the ACTV stop.

The algorithm seems to struggle a lot with images that are very blurry and grayscaled, such as `venice/01.png` and `venice/02.png`. In both cases the segmentation was subpar and, as a results, very large segments included lots of water containing boats, and the IoU score took a big hit. In general, almost all cases in which the segmentation caused a boat to fuse with water resulted in the whole water picking up boat keypoints and being classified as a boat. Other low scores not related to segmentation issues were probably caused by uncommon points of view (such as `venice/07.png`) which despite a good segmentation prevented correct detection.



kaggle/01.jpg. Segmentation split the big ship in 5 segments, most of which were correctly detected, but detection overall is not very precise.

kaggle/04.jpg. Even though the image was segmented well enough, the boat segment was not detected as a boat (perhaps the segment was too limited to the interior) and the sky must have picked up some boat descriptors that were enough to trigger incorrect detection.

	
<p>kaggle/05.jpg. First image to be not segmented at all and recognised as just one cluster, but the detector did correctly predict a boat was present.</p>	<p>kaggle/10.jpg. Best result across all datasets, even though both the dinghy and the water are segmented in three portions each.</p>
	
<p>kaggle/09.jpg. Another case of a good segmentation but the sky segment being the one to be recognised as a boat instead of the ship segment.</p>	

The Kaggle dataset exposed lots of weaknesses in the pipeline that could be directly traced to segmentation. For example, some images contained lone ships that were too small and were thresholded out, or even swallowed by another cluster.

Another problem was that even correct segmentations failed detection: it may be that boat segments are too tight to pick up enough of the boat's keypoint-rich border for the cluster to register as a boat. A possible solution may be to dilate each segment mask, but this will certainly also lead to less accurate results.

In general, it seems extremely difficult to strike the balance and achieve a tradeoff that works across very different datasets and types of images.

Some efforts were poorly focused, in hindsight; in my mind, building a strong segmentation algorithm would have allowed to place nice, tight bounding boxes, but very good segmentations did not necessarily lead to good results, and imperfect segmentations (where a boat was split in more components) were sometimes picked up better than rather good segmentations.

It seems that it could have been better to accept smaller segments, run inference on them and then merge hits according to some similarity metric to build the full boat. Segmentation was also pretty slow because of meanshift (in no way could my approach have worked for tracking!); again, the higher quality segmentation might not have even mattered in the end, since other parts of the process were flawed themselves.

As for the training and inference process, it could chiefly be improved in three respects:

1. Use a richer training set for the purpose of negative sample detection, seeing as the Venice dataset struggled a lot with buildings;
2. Perform better validation on the tunable parameters of the dictionary (chiefly, class count) and SVM (the ν parameter).

While the segment-and-detect strategy seems like an option that could work for a system that does not rely on end-to-end NN usage, clearly the whole detection element could have been replaced by a NN, and it's unclear how good the performance of a non-NN detector could ultimately be. This doesn't mean that this approach cannot be improved without changing major parts of the pipeline though: the results were generally poor, but areas of improvements have been identified.

Instructions

Command line options

The executable can be used in four modes using the following options.

Inference/detection. Use the following command line options:

-i	Sets inference/detection mode
-I="path_to_input_file" --input="path_to_input_file"	Sets path to input file

The program also supports:

-G="path_to_GT_file" --groundtruth="path_to_GT_file"	Sets path to ground truth file (optional, for result evaluation via IoU)
-O="path_to_output_file" --output="path_to_output_file"	Sets path to output file (optional, to save result)
--dictionary="path_to_dictionary"	Sets path to dictionary to use in inference (optional, default: ./dictionary.yml)
--svm="path_to_svm"	Sets path to SVM to use in inference (default: ./svm.yml)
--show_segments	Shows the final result of segmentation
--show_seg_steps	Shows all steps of segmentation process

Dictionary training. Use the following command line options:

-d	Sets dictionary training mode
-T="path_to_training_set" --training_set="path_to_training_set"	Sets path to training set root folder

The program also supports:

--dictionary="path_to_dictionary"	Sets path for output dictionary to be saved to disk (optional, default: ./dictionary.yml)
--	---

SVM training. Use the following command line options:

-s	Sets SVM training mode
-T="path_to_training_set" --training_set="path_to_training_set"	Sets path to training set root folder

The program also supports:

--dictionary="path_to_dictionary"	Sets path for dictionary to use in training (optional, default: ./dictionary.yml)
--svm="path_to_svm"	Sets path for output SVM to be saved to disk (default: ./svm.yml)

Full (dictionary and SVM) training. Use the following command line options:

-t	Sets full training mode
-T="path_to_training_set" --training_set="path_to_training_set"	Sets path to training set root folder

The program also supports:

--dictionary="path_to_dictionary"	Sets path for output dictionary to be saved to disk (optional, default: ./dictionary.yml)
--svm="path_to_svm"	Sets path for output SVM to be saved to disk (default: ./svm.yml)

Training set and ground truth format

The training set supplied for training must have the structure:

<training set root folder>/	IMAGES/	image0001.png
		...
		imageXYZ.png
	LABELS_TXT/	image0001.txt
		...
		imageXYZ.txt

It is not required that every image in the IMAGES folder has an associated ground truth file in the LABELS_TXT folder, but images without a ground truth file will not be used for SVM training.

The format for ground truth files is that agreed upon by students of the course. The ground truth file **imageXYZ.txt** contains a row for each boat displayed in **imageXYZ.png**:

class:X1;X2;Y1;Y2;

where **class** is currently ignored, and **X1,X2,Y1,Y2** are integer coordinates to the boat bounding box.