Searching for Semantic Segmentations in Urban LIDAR Data

David Dohan, Brian Matejek, and Thomas Funkhouser
Princeton University
Princeton, NJ USA

In this supplementary material, we elaborate implementation details of our algorithm, along with providing a few additional illustrative examples of successes and failures of the merging process.

1. Implementation Details

1.1. Segment features

We use pointwise features (section 1.3) as the basis for computing segment descriptors. Given a supersegment S, which may be composed of multiple segments from the initial oversegmentation, we compute statistics (percentiles, average, variance, maximum difference) based on each of the input feature types (e.g., height, linearness, curb depth...). In addition to features based on statistics of pointwise features, we extract features related to the extent of supersegment. Specifically, we include the number of member segments, length and depth extent relative to the curb, along with the total height of the object. Based on these values, we also compute bounding box volume, the bounding box area when projected to ground, and the ratio of the longer horizontal edge to the shorter one. We concatenate these features together to form a feature descriptor for a segment. This results in a 96 dimensional feature vector to describe a segment. It is straightforward to substitute other features in place of both the segment and merge features which we used.

1.2. Merge features

Merge features are based on the feature vectors describing each of the merge candidates. Given a merge between A and B, the core of the merge descriptor is based on concatenating the descriptor for each, the differences between the two descriptors, and the descriptor for segment if the merge were completed, giving the descriptor a dimensionality 4 times the segment descriptor. We also compute other merge specific features related to the relative alignment of the two merge candidates, including: (1) Amount of overlap along each dimension. If A's bounding box is contained inside B's bounding box along one dimension, then the overlap is the A's size along that dimension, and (2) Distance between the two candidates centroids along each dimension and in 3D.

Together, this descriptor allows the merge affinity function to consider the shape of each candidate segment, the relative differences between them, and how object like the combined object would be. Additionally, it captures information about the pose of the two segments, such as whether they are stacked vertically or one projects from the other (as is often the case in objects such as street lights).

1.3. Feature types

Along with 3D position, each point in a run has a number of features associated with it based on the region around it in 3D space and projections from images onto the point cloud. We use a number of these precomputed features for each point, which we now briefly explain.

RGB color: Based on projections from images, each point is assigned a red, green, and blue color value. Images are not necessarily aligned, so colors are not generally accurate around the edges of objects, but this feature captures the overall color of large objects well, such as the color of trees.

Depth from curb: This specifies the distance from the curb for each point. This is an application specific feature (urban point clouds), and is based on an estimate of curb location found during preprocessing.

Viewpoint depth: Measures the distance from the scanner to the point. This is the raw feature produced by the scanner, along with scan angle, as it records data.

Height: Height of each point relative to the estimated ground plan

$\lambda_2 - \lambda_1$ and $\lambda_3 - \lambda_2$: These two features are based on applying principal component analysis to the neighborhood around each point. If the eigenvalues are $\lambda_1 > \lambda_2 > \lambda_3$, then $\lambda 2 - \lambda 1$ is a measure of "linearness" , while $\lambda 3 - \lambda 2$ measures "surfaceness" of the region.

$\mathbf{n} \cdot \mathbf{v}$: Dot product between the estimated normal to each point (obtained by fitting a plane plane to its neighborhood) and the vector from the LIDAR scanner to the point.

Segmentwise normals: The oversegmentation process includes fitting a plane to each segment individually. Each point includes features representing the Z component of the vector representing this plane for its segment.

1.4. Implementation

The project is implemented in C++. Because of the large dataset size, each stage (training example extraction, classifier training, and testing) is implemented to run separately. The programs share a common backing library which is split into a classification project specific code.

The project specific backing library focuses on the implementing IO, representing the depth images for ease of use, feature extraction, and the core of the training and testing algorithms such as graph construction and the actual affinity based merging methods. An important abstraction is the Supersegment type, which we use throughout all segment based methods to handle computing and caching descriptors (the majority of each extraction and prediction step is spent computing descriptors, so this is important for performance).

1.4.1 Classifiers

We use the random forest implementation in the Shark Machine Learning Library [3] (based on [1]) for all classification in our project. We found that it consistently provided good classification and probability (in the form of ensemble confidence) results using default parameters.

1.4.2 Merging Algorithm

There are a number of hierarchical merging implementation details to consider. The minimum information we use to represent each supersegment is (a) a list of segment ids that compose it, (b) a record of the supersegment which it was merged into (its parent) if it has been merged into another supersegment, and (c) an update id which is used to determine if a merge is out of date. When the merging process completes, any supersegments without a parent are used as the final segmentation (having a parent indicates that the segment has been merged into another segment). When a supersegment B is merged into A, we add all member segments from B into A, update B to have A as a parent, and set each update id to the current merge number. The parent information provides a fast way of mapping from a segment in the original oversegmentation to the supersegment which it is a part of when adding new merge candidates. The final supersegment that a segment is part of can be determined by walking up the parents until the root is found. An additional optimization is to do path compression by updating the parent reference to the root whenever a query is made. This loses information about the order in which segments were merged together, but we do not use this information in our algorithm.

The update id is necessary because, unlike in pure hierarchical clustering in which every merge above a certain score is completed, merges can go out of date if one of the supersegments in the merge has changed since the merge was added to the queue, and it is slow to clean out-of-date merges immediately after each merge. The update id marks how up to date each is so we can easily discard old merges as they reach the top of the queue. We found that this works well in practice and is practical on even our largest point sets. We use a counter of the number of merges executed so far, and each supersegment is updated with this counter when it is involved in a merge. Merges store this value for both supersegments in the merge when constructed, and to check if a merge is out of date, we check if this update id matches between the merge and supersegments.

2. Examples

In this section, we show several additional examples of successes and failures of the algorithm, along with the illustrating case for why merging segments is desireable and how the initial oversegmentation can fail in figure 2. In each figure,

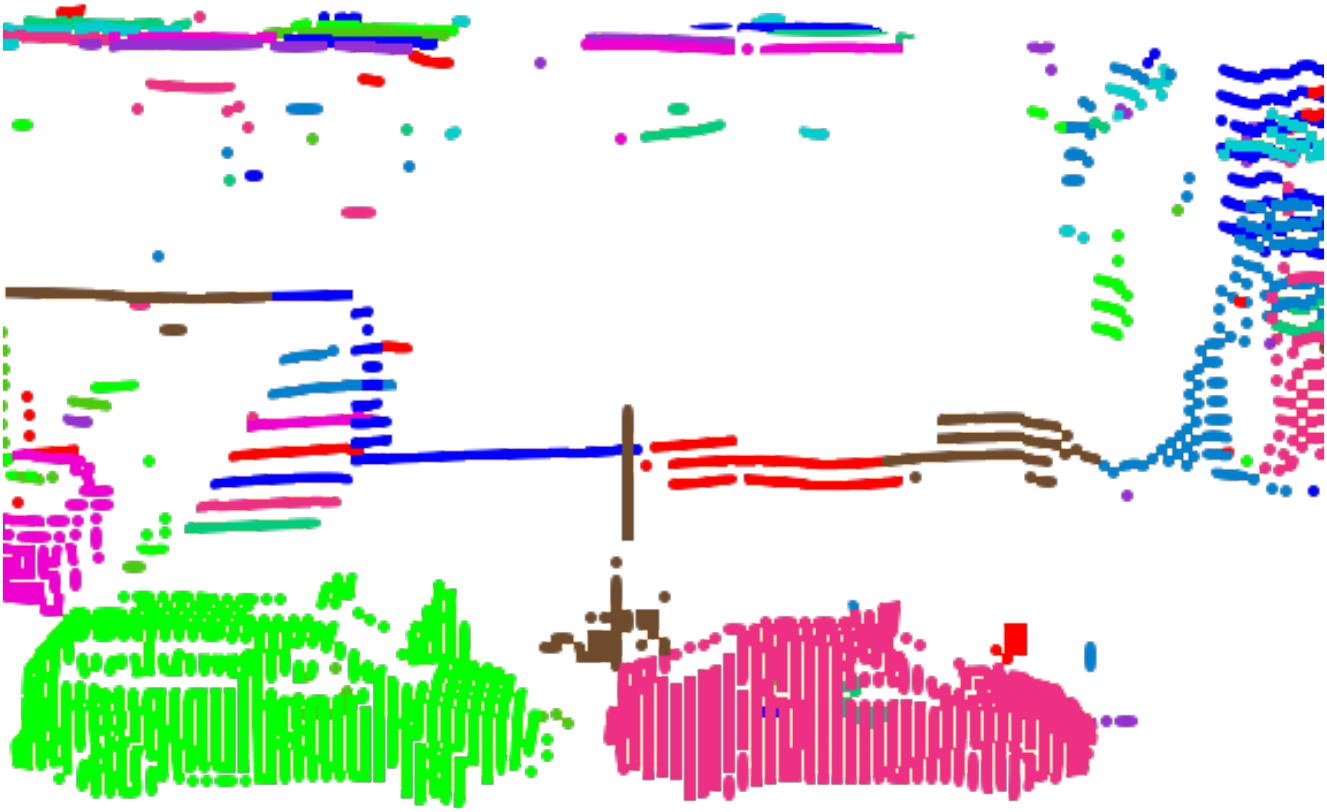


Figure 1. The oversegmentation removes most large objects, but some remnants remain, as indicated by the lines of building points in the background. The oversegmentation may also combine some small objects, such as including a nearby person in the green car segment or a bicycle with a sidewalk pole. Future work includes determining how a finer oversegmentation performs as a starting point.

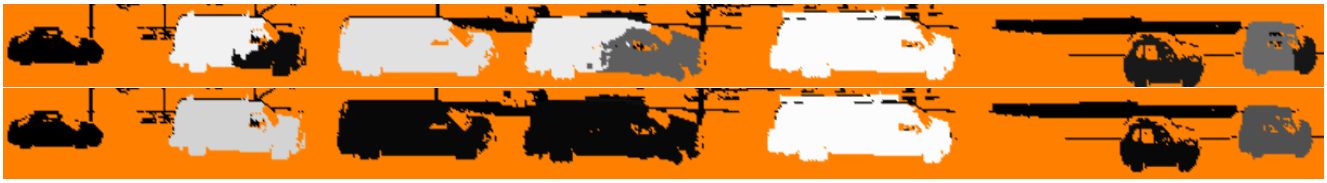


Figure 2. Example illustrating classifying vans in an oversegmentation (top) and a merged segmentation with bad merges (bottom). The front part of a van appears similar to a part of a car before merging, but merging with the rest of the van makes the class clear, improving classification quality (leftmost and rightmost vans). This case also shows how merging can be detrimental, as the merging process merged two vans into a single segment, which dropped the predicted probability of both to just above the segment classifier threshold.

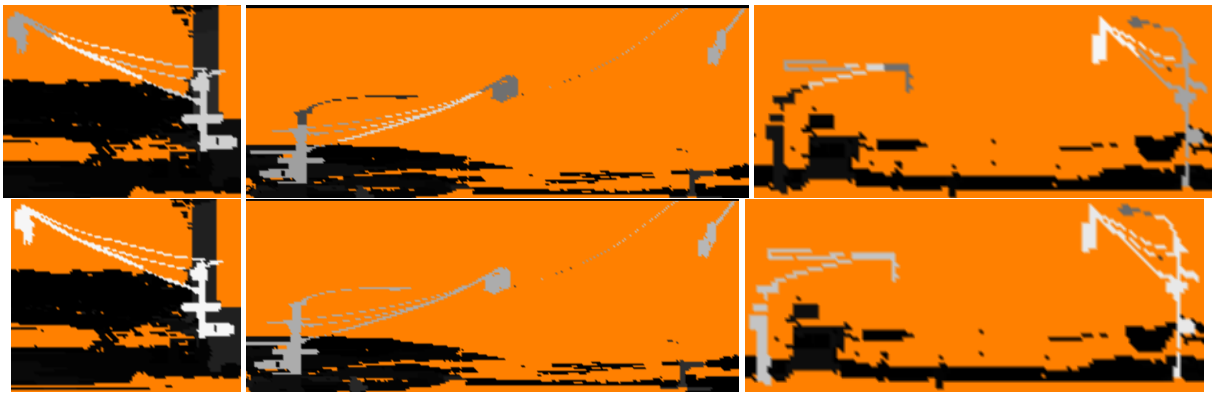


Figure 3. Examples of merging helping the traffic light class

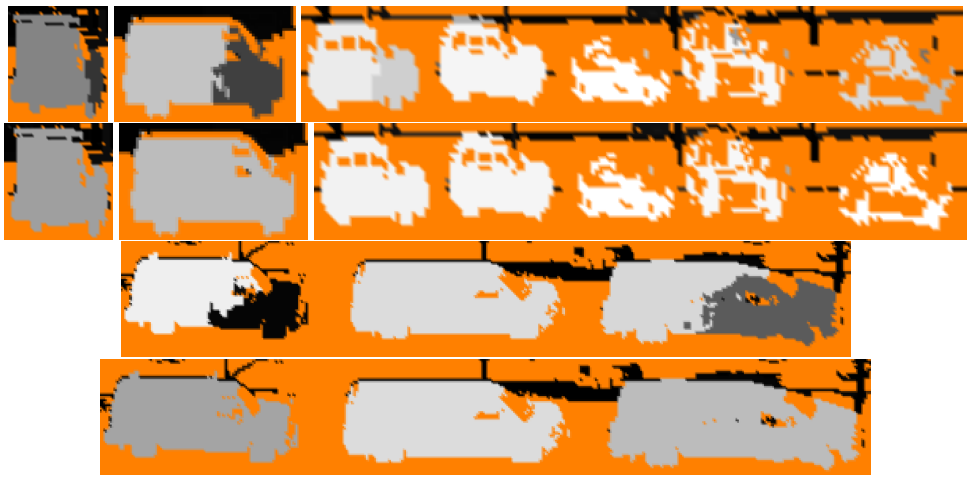


Figure 4. Examples of merging helping the vehicles. Many vehicle merges improve the segmentation quality, but do not change the recognition precision-recall metrics at all because the constituent part scores are already fairly high

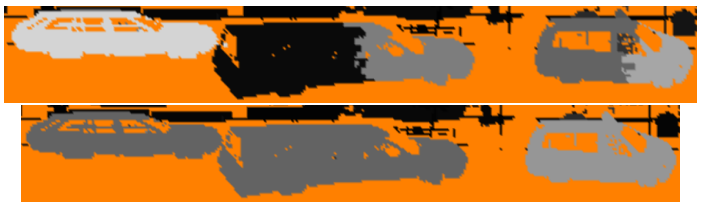


Figure 5. Example of merging helping accuracy but producing an incorrect segmentation. It incorrectly merges the left and right cars together, which has the side effect of increasing overall confidence.

3. Parameter exploration

3.1. Effects of subsampling pointwise training data

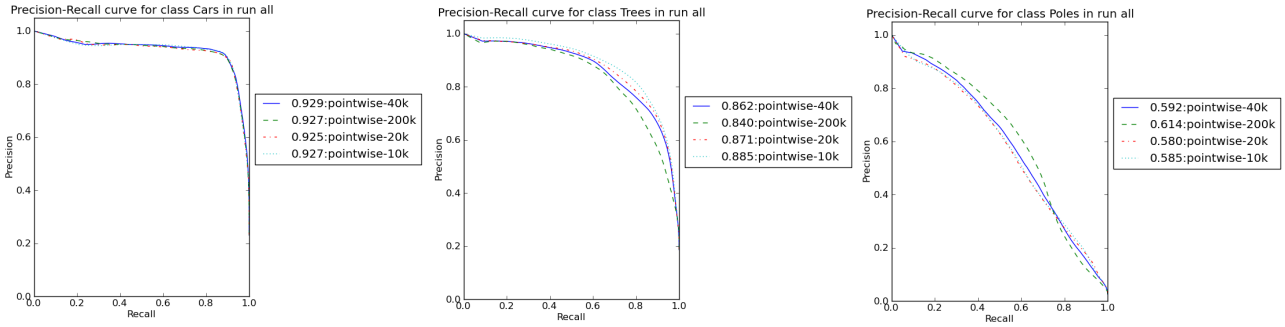


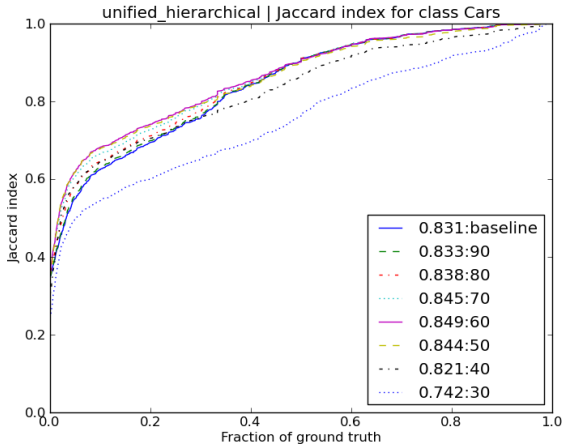
Figure 6. Varying the number of training examples has negligible effects on performance across the three broad categories

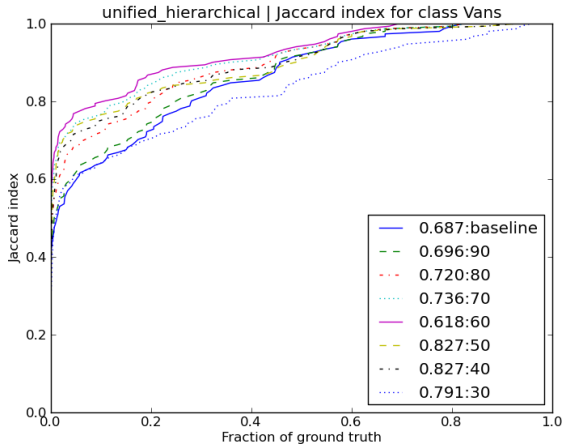
While training on every segment from the oversegmentation is on the order of a few tens of thousands of training examples, pointwise training can draw on a much larger number of training examples. We evaluated the effect of subsampling the pointwise training examples before training on classification accuracy for each class.

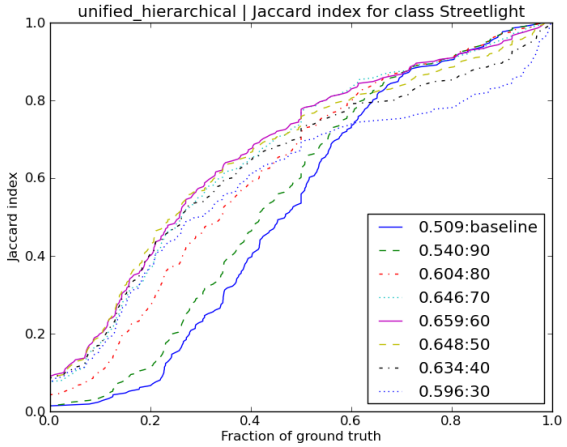
Figure 6 shows the results of training pointwise predictors on 10k, 20k, 40k, and 200k point training examples. We conclude that using more training examples does not necessarily help pointwise classification accuracy. For the three grouped classes, it helps a marginal amount for poles, has minimal effect for cars, and hurts for trees as much as it helps for poles. One explanation for this is that each point descriptor is low dimensional (only 8 in our experiments). There are only a few point characteristics that explain most of the examples, and these are mostly captured by few training examples. Car predictions, for example, are mostly explained by the depth from curb feature (as verified using Info Gain in the Weka toolbox [2]). Examining the outputs reveals that the improvement for poles is actually at the detriment of trees, as tree trunks are commonly confused between the two.

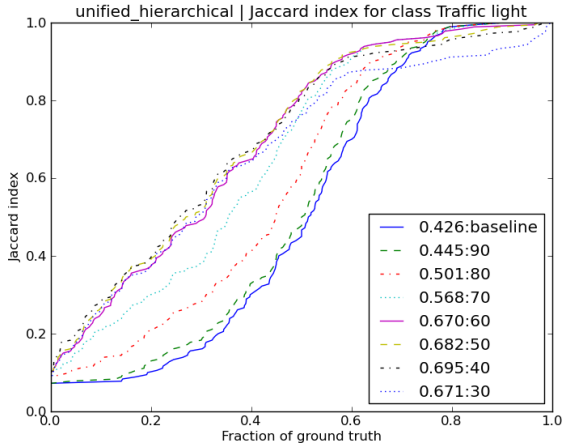
3.2. Effects of threshold on jaccard indices

In this section, we illustrate the effects of allowing merging to proceed too far on segmentation quality. Each graph shows the jaccard index plots without any merging, along with at various thresholds. A threshold of 50 indicates that only merges with a confidence above .5 were considered. "unified_hierarchical" refers to using the same non-class-specific merge classifier for all classes. In general, we see that segmentation quality increases steadily until about .5, after which it decreases in quality until even worse than baseline.









References

[1] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. 2

[2] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009. 5

[3] C. Igel, V. Heidrich-Meisner, and T. Glasmachers. Shark. *Journal of Machine Learning Research*, 9:993–996, 2008. 2