Learning to Merge: A New Tool for Interactive Mapping

Reid B. Porter[*a], Sheng Lundquist[a], Christy Ruggiero[b]
[a]Intelligence and Space Research; [b]Nuclear Engineering and Nonproliferation,
Los Alamos National Laboratory, Los Alamos, New Mexico, USA 87545

ABSTRACT

The task of turning raw imagery into semantically meaningful maps and overlays is a key area of remote sensing activity. Image analysts, in applications ranging from environmental monitoring to intelligence, use imagery to generate and update maps of terrain, vegetation, road networks, buildings and other relevant features. Often these tasks can be cast as a pixel labeling problem, and several interactive pixel labeling tools have been developed. These tools exploit training data, which is generated by analysts using simple and intuitive paint-program annotation tools, in order to tailor the labeling algorithm for the particular dataset and task. In other cases, the task is best cast as a pixel segmentation problem. Interactive pixel segmentation tools have also been developed, but these tools typically do not learn from training data like the pixel labeling tools do. In this paper we investigate tools for interactive pixel segmentation that also learn from user input. The input has the form of segment *merging* (or grouping). Merging examples are 1) easily obtained from analysts using vector annotation tools, and 2) more challenging to exploit than traditional labels. We outline the key issues in developing these interactive merging tools, and describe their application to remote sensing.

Keywords: interactive segmentation, structured output prediction, image analysis tools, geospatial mapping

1. INTRODUCTION

An important activity in remote sensing is the generation of maps and overlays from overhead imagery. In some cases the key quantities of interest are well known and specific to the particular type of imagery used. In this case remote sensing scientists often develop sophisticated physics-based algorithms to estimate the quantities, and once developed, these algorithms typically run in an automated fashion. In other cases, the specific quantities of interest are not known ahead of time, or, there are too many combinations of quantities and image types to develop algorithms up-front, or, there is simply too much variability between images for automated algorithms to perform well. In these cases interactive tools, which help end-users develop maps and overlays on-the-fly, can provide a key capability.

Interactive tools are used in a wide variety of remote sensing applications. Some of these applications can be formulated as pixel classification, where each pixel is assigned a semantically meaningful label that is associated with a particular land cover or terrain type [1]. Another class of applications can be formulated as object delineation, or pixel segmentation, where each pixel is assigned to a cluster (or partition) that is associated with real-world objects [2], or changes [3]. Interactive tools have also been developed to learn sequences of user interaction during a shape production task for cartography [4].

In this paper we develop tools for a new class of application related to segmentation. Most interactive segmentation tools have been developed to help analysts delineate a small number of complex objects. The analyst provides a partial solution, and the segmentation algorithm completes the work. In this paper we develop interactive segmentation tools that help analysts delineate a large number of similar objects. The analyst provides a number of complete objects as examples, and the segmentation algorithm learns to complete other examples.

In Section 2 we provide an overview of the technical methods that underlie classification and segmentation tools. In Section 3 we describe how these methods can be applied to the new class of interactive segmentation tools that "*learn to merge*". In Section 4 we present a number of satellite image mapping applications that we use to evaluate our work. These applications illustrate the wide utility of tools that can "*learn to merge*", and highlight some of the key technical challenges.

[*] rporter@lanl.gov

2. STRUCTURED OUTPUT PREDICTION

Pixel classification and segmentation problems have received extensive study and numerous methods have been developed. In recent years there has been a growing body of work in a sub-field of machine learning, known as structured output prediction. Structured output prediction provides a unified framework for classification and segmentation problems. In this section we describe classification and segmentation problems in this context and place particular emphasis on the aspects of these problems most relevant to interactive mapping applications.

2.1 Classification Tools

Pixel classifiers are now a standard tool in the geospatial image analysis toolbox. The typical work-flow for using these tools is illustrated in Figure 1 and involves three main steps.

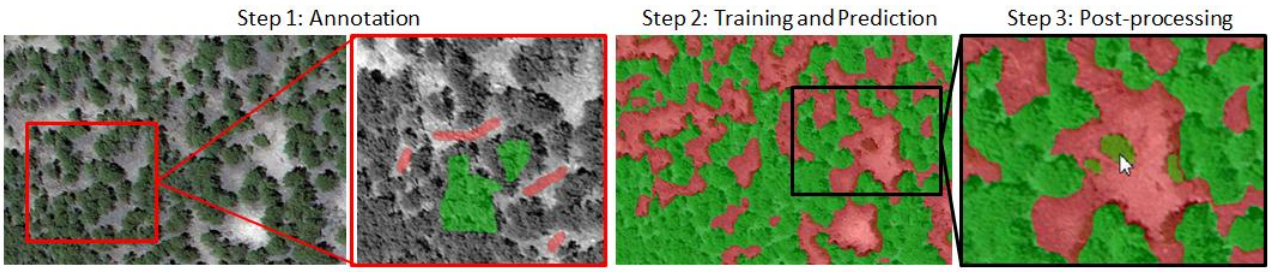

Figure 1: A typical workflow for pixel classification involves 3 main steps: 1) annotation, 2) training and 3) post-processing.

Step 1: The first step in the workflow requires the user to provide training data through basic annotation tools, such as a paint-brush, line or polygon drawing tools. These annotation tools are used to label pixels as belonging to different classes, or categories. Typically, the color of the annotation is associated with a particular class. For example, in Figure 1, the user has provided training data for two classes: pixels annotated with the red brush-strokes correspond to the background class (-1) and pixels annotated with the green brush-strokes correspond to the class of interest (+1), which in this case is vegetation.

Step 2: We introduce some notation to explain how these annotations are used by pixel classifiers. Our presentation is a little more general (and therefore more confusing!) than the typical presentation, but this will help explain segmentation in subsequent sections. An image X is a collection of pixels $\{\dots, x_i, x_j, x_k, x_l, \dots\}$ arranged on a regular grid as shown in Figure 2. Pixels are assumed to be real-valued, $x \in \mathbb{R}^D$, but could have one (grayscale), three (color) or any number of dimensions (multi-, hyper-spectral). We define Y as a set of discrete random variables for the pixel labels, where each variable is associated with a pixel location $\{\dots, y_i, y_j, y_k, y_l, \dots\}$. The number of distinct labels can be quite large but we will typically discuss the binary case: $y \in \{+1, -1\}$.

The pixel classifier is defined with an energy function[†]:

$$E(Y) = \sum_i f(y_i, X) \tag{1}$$

which is simply a sum of energies calculated at each location. In some applications the pixel classifier focuses on spectral information, and in this case, the energy at each location only depends on the pixel at that location, $f(y_i, x_i)$. However in the more general-case pixel classifiers may exploit spatial information and this introduces neighborhood dependencies as illustrated in Figure 2.

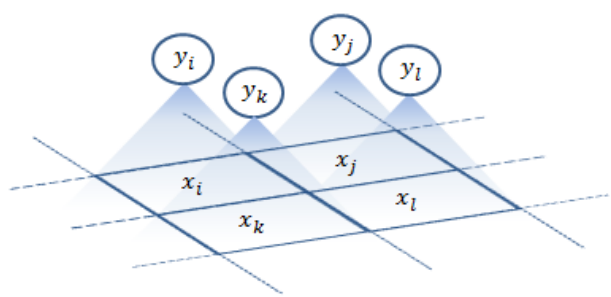

Figure 2: Pixel classifiers predict a set of random variables, where each variable is associated with a pixel location.

[†] Energy functions are often associated with a probability density function and given a probabilistic interpretation.

Once the energy function is defined, the classifier predicts labels for each location through a minimization known as inference:

$$\hat{Y} = argmin_{\,Y}\, E(Y) \tag{2}$$

For pixel classification, this minimization problem is often simplified by minimizing the energy at each location independently, which means, $\hat{Y} = \{\hat{y}_1, \hat{y}_2, \dots, \hat{y}_P\}$, where $\hat{y} = argmin_{\,y}\, f(y,\, X)$. In the binary case, we can take this a step further and define $f(y,\, X) = yf'(X)$. With these simplifications, the standard pixel classifier:

$$\hat{y} = sign\big(f'(X)\big) \tag{3}$$

is a minimizer of Eq. 2. So far, we have assumed that the energy function is provided. But in practice, we would like to tailor the energy function based on the training data provided in Step 1. This is accomplished through learning. The annotation tools produce a training sample, which is a set of (pixel, label) pairs:

$$S = \{(x(1), y(1)), (x(2), y(2)), \dots, (x(N), y(N))\}$$

We parameterize the energy function, for example we may choose a linear combination of features:

$$f'_{\,w}(X) = w^T . \begin{bmatrix} F_1(X) \\ F_2(X) \\ \vdots \\ F_m(X) \end{bmatrix}$$

For a particular choice of parameter values in the energy function, we have an inference result:

$$\hat{Y}_w = argmin_{\,Y}\, E_w(Y)$$

The objective of learning is to find the parameter values that minimize a loss function based on the training set:

$$\hat{w} = argmin_{\,w}\, L\big(\hat{Y}_w, S\big) \tag{4}$$

The loss function for pixel classification is usually related to the misclassification rate and also often includes a regularization term:

$$L\big(\hat{Y}_w, S\big) = \sum_{n=1}^{N} I(\hat{y}_w(n) \neq y(n)) \;\; + \;\; \lambda \|w\| \tag{5}$$

where $I(\cdot)$ is the indicator function and λ is a meta-parameter.

Step 3: Once trained, the pixel classifier can be used to predict the label for unlabeled image pixels. Of course, in most applications the classifier is not perfect and there will be mistakes. At this point, a user interacts with annotation tools again, but this time it is to clean up and correct the predicted results. The revised predictions can be fed back into the workflow to improve the accuracy of the classifier in an iterative fashion. In some applications the iterative approach can produce more accurate classifiers than training a classifier with all the data up-front (see page 19 in [5]).

2.2 Segmentation Tools

Segmentation (or pixel clustering) is another important tool in the geospatial image analysis toolbox. In this case, the user is less interested in specific labels, and more interested in grouping similar pixels into some (often unknown) number of classes, or clusters. Segmentation can be formalized in a similar way to pixel classification. The significant difference is that segmentation models statistical dependencies between elements of Y. These dependencies are illustrated with edges in Figure 3 and are represented by pair-wise terms in the segmentation energy function:

$$E(Y) = \sum_{i,j} g\big(y_i, y_j, X\big) \tag{6}$$

The energy associated with each edge often has the general form:

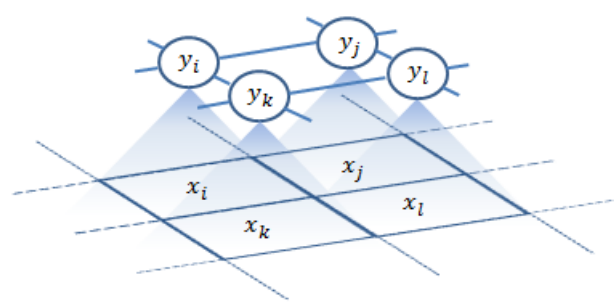

Figure 3: Pixel segmentation predicts a set of dependent random variables. Dependencies often form a lattice in the image plane.

$$g(y_i, y_j, X) = I(y_i \neq y_j)\, A_{i,j} \tag{7}$$

Where $A_{i,j} \epsilon \mathbb{R}$ is an affinity, or similarity measure between pixels i and j. Minimizing this energy function leads to a labeling $\hat{Y}$ that assigns the same label to similar pixels and different labels to dis-similar pixels. This energy function is used for clustering as well as image segmentation. In panchromatic imagery the affinity is typically related to the image gradient.

Many different interactive segmentation tools have been developed and are used in a wide range of applications. In bio-medical image analysis interactive segmentation has been used to delineate complex 3-dimensional objects such as organs and bones [6], identify synaptic pathways [7], and count different cell types [8]. Similar applications are also found in material science [9], geology and remote sensing [2], [3].

Step 1: In all of these applications, the user input is similar to pixel classification, but it is used by the segmentation algorithm in a completely different way. Unlike pixel classification, which incorporated user input via training (step 2), interactive segmentation has traditionally incorporated user input directly into the energy function. Figure 4 illustrates the general workflow. User input (top right) is generated by paint-program annotation tools. This input is often called markers (or seeds) and it is used to define segmentation constraints. Each marker (each rectangle in Figure 4) identifies a set of pixels that should belong to the same segment. It is assumed that pixels under different markers should belong to different segments. These constraints are included in the energy function by introducing additional terms, $h(y, m)$, that penalize segmentations that violate the marker constraints:

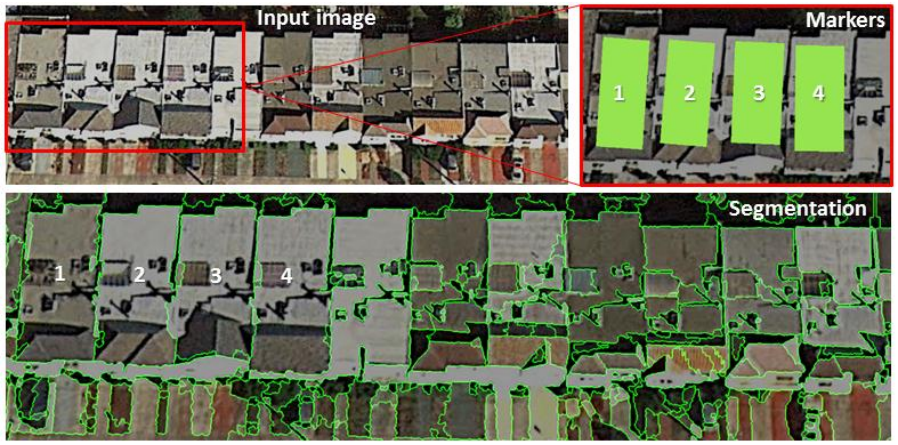

Figure 4: Most interactive segmentation algorithms incorporate user input (markers) directly into the energy function. In this example the four markers help produce the larger (more complete) segmentations for the four houses of interest.

$$E(Y) = \sum_{i,j} g(y_i, y_j, X) + \sum_i h(y_i, m_i) \tag{8}$$

Couprie et. al. [10] provide specific details of Eq. 8 and show how variations of the energy function lead to a large number of interactive machine learning methods including graph-cuts [11], random walkers [12], watershed [13], and geodesic segmentations [14]. Note, that these interactive tools finish after Eq. 8 is minimized. In Figure 4 we see that 4 markers generate reasonable segmentations for the 4 houses, but the remaining houses are over-segmented. In traditional interactive segmentation tools, the user must place a marker on each object of interest, and there is not attempt to learn from markers and adjust the energy function globally. In many mapping applications, where the objects of interest reoccur throughout the image and larger archive, this can become a repetitive and tedious process.

Step 2: More recently, researchers have developed training methods for segmentation algorithms. This is partly due to the development of training sets that include *ground-truth* segmentations [15], and partly due to developments in structured output prediction ,which provides training methods for more complex energy functions. Learning is much like learning in pixel classification but with a more complex training set:

$$S = \{(X(1), Y(1)), (X(2), Y(2)), \dots, (X(N), Y(N))\} \tag{9}$$

Each training instance includes a complete image with its corresponding ground-truth segmentation. We use the energy function defined by Equations 6 and 7 and we parameterize an affinity function:

$$A_{i,j} = f_w(x_i, x_j) \tag{10}$$

Similar to pixel classification, the objective is to estimate the parameters by minimizing a loss function. However, unlike pixel classification the solution to Eq. 2 (inference) does not simplify and, in fact, in many cases it is intractable

(exponential in the number of variables). The dependencies between the variables Y in the energy function help to capture the desired global characteristics of the segmentation problem. These must be taken into account when learning the parameters of the local affinity function, which means we must solve inference as part of learning. The relationship between learning (Eq. 4) and inference (Eq. 2) is the topic of a recent ICML workshop [16] and a growing body of research.

When the affinity function in Eq. 10 is real-valued, the segmentation method is closely related to correlation clustering, and inference is NP-hard [17]. A number of general purpose methods have been developed for finding approximate solutions. Most related to this paper is the application of structured support vector machines to the correlation clustering problem [18]. Another approach is to restrict the affinity function in a way that makes the inference problem tractable. Turaga et. al. [19] suggest using a binary affinity function and an inference procedure based on connected component labeling (a depth-first search). We will discuss this more in section 3.

Step 3: Once a segmentation has been produced, users often need to correct and clean-up the result. There are a number of editing tools that can be used to split, merge and fine-tune segments. In this paper we focus on merging. We assume that the initial segmentation produces an over-segmented result and the user selects groups of segments to merge. This is illustrated in Figure 5. We also assume that there are a large number of merges that the user must perform to complete the task, and our objective is to learn from as few examples as possible to predict (automate) additional merges.

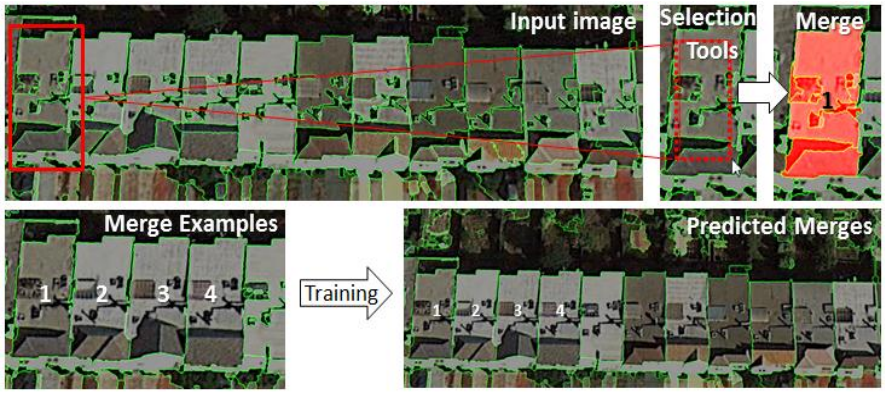

Figure 5: In learning to merge, the user edits and corrects an initial segmentation (from Step 1 or 2) using merging and splitting tools. In this example, the user selects a number of segments (top-right) and merges them into larger segments (bottom-left). Additional merges are then predicted by the tool (bottom-right).

Learning to merge has recently been called *learning to agglomerate* by Jain et. al. [20]. They investigate a reinforcement learning approach to the problem. In this paper we follow the approach developed by Turaga et. al. [19] very closely. One difference in our approach is that instead of learning an initial segmentation, we learn a second, higher-level segmentation that merges the first-level segmentation graph (super-pixels), much like Jain et. al. [20].

We note that most segmentation tools typically focus on one of the three steps. However, in practice these steps could be used in any combination and, in fact, the three different steps solve complementary problems. We suggest steps 1 and 3 are currently the most important to interactive mapping applications because the user input in these stages is often easier for end-users to provide. In step 1, the user identifies a subset of pixels for each segment through markers. In step 3, the user selects a subset of the segments that should be merged. Step 2 typically requires a complete segmentation, which can be difficult (and expensive) for end-users to provide. Some recent work has developed a structured output prediction approach that uses latent variables to relax this requirement [21] which makes step 2 more conducive to interactive settings. Future interactive segmentation tools might well exploit aspects of all three types of user input.

3. LEARNING TO MERGE

In this section we define learning to merge (Step 3 in Section 2.2) in more detail. Our approach assumes an initial segmentation which partitions the image into a number of segments and defines a planar graph $G = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$. A vertex $v_i \epsilon \mathcal{V}$ is associated with a segment x_i (previously a pixel) and edges $e_{ij} \epsilon \mathcal{E}$ are associated with pairs of segments $\{x_i, x_j\}$ that are connected in the image plane.

3.1 Energy Function

Similar to Turaga et. al. [19] we restrict the affinity function to be binary:

$$C_{i,j} = I\left(f_w\left(x_i, x_j\right) > \theta\right) \tag{11}$$

And we will use connected components as an inference procedure. We refer readers to [22] for the energy function associated with this procedure. As pointed out in in [19], connected components is simpler than most segmentation algorithms, but it is interesting for two main reasons:

1. We can develop tractable modifications of standard learning algorithms to train a classifier. This means we can put the complexity into the affinity function.
2. Connected components guarantees transitivity, which is often a key global property of interest in segmentation (and clustering). More complex methods (such as correlation clustering) often benefit from transitivity-like properties, but it may not be guaranteed.

3.2 Training Data

Our training set has the following form:

$$S = \{m(1), m(2), \dots, m(N)\} \tag{12}$$

where each example specifies a subset of vertices $m(n) = \{v_i, v_j, \dots, v_k\}$. The number of vertices in each example can vary from one sample to the next. We also assume that vertices within each set form a connected component. This type of training data is easily collected from users by various segment selection tools, such as the rectangle selection tool shown in Figure 5. Formally, we represent the merge examples by assigning unique identifiers to vertices in the image graph. The vertex identifiers are illustrated with color on the right in Figure 6. They are used to generate a new training set for the edges which is illustrated on the right in Figure 6. Every edge that joins two vertices with the same identifier (part of the same merge) is assigned a +1 label. Note, in our experiments the user does not provide any explicit negative examples, i.e. *do-not merge* examples. Instead, we assume negative examples implicitly: every edge that has one vertex associated with a merge identifier is assigned a label of -1. All remaining edges connect vertices that are not part of any merge, and these are left unlabeled. In fact, with our approach we assume there are (many) other positive examples in the graph that are not provided, and our objective is to predict these components.

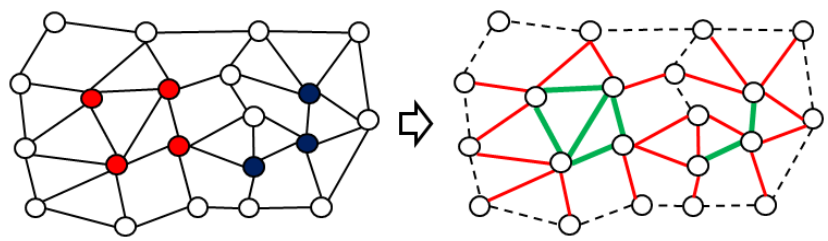

Figure 6: Left) Training data collected from user. Right) Edge labels generated from training data. Green edges indicate +1 and red edges indicate -1.

3.3 Loss Functions

A number of different loss functions have been suggested for segmentation and clustering problems. The per-sample loss for segmentation, $\Delta(\hat{Y}, Y)$, is based on comparing the predicted segmentation $\hat{Y}$ for a particular training image X to the desired segmentation Y. A popular choice is based on the Rand index which makes a comparison over all pairs of variables [19]:

$$\Delta(\hat{Y}, Y) = \binom{N}{2}^{-1} \sum_{i<j} \left| I(y_i = y_j) - I(\hat{y}_i = \hat{y}_j) \right| \tag{13}$$

In this paper we investigate a variation of the Rand Index that only sums over edges within the graph:

$$\Delta(\hat{Y}, Y) = |\mathcal{E}|^{-1} \sum_{e_{ij} \epsilon \mathcal{E}} \left| I(y_i = y_j) - I(\hat{y}_i = \hat{y}_j) \right| \tag{14}$$

This is similar to the MITRE loss [23] which measures the number of splits and/or merges required to correct the prediction. The MITRE loss was investigated in combination with the Rand index in [18]. Unlike the Rand index, Eq. 14 does not place weight on particular splits and merges based on the component size. With the Rand index an edge that splits one segment from a large component would be preferred to an edge that splits a large component into two equal sized sub-components. In some applications this weighting can be useful since it produces visually similar segmentations. However in other (more interactive) applications, a user might be trying to perfect a segmentation and, in this case, errors in the larger segments might be preferred since they could be more easily identified and corrected.

3.4 Learning Algorithms

Our objective is to find parameters for an edge classifier (Eq. 11) that minimizes the loss function in Eq. 14. The general challenge is that the classifier output is only indirectly involved in the loss function (through $\hat{Y}$). One approach is to rewrite the loss function so that the classifier output is directly involved [24]:

$$\Delta(\hat{C}, Y) = |\mathcal{E}|^{-1} \sum_{e_{ij} \epsilon \mathcal{E}} \left| I(y_i = y_j) - \hat{C}_{i,j} \right| \tag{15}$$

However, this does not capture the connected components inference step. If $C_{i,j} = 1$ then vertices i and j are guaranteed to share the same label after inference. But, when $C_{i,j} = 0$, the lowest energy labels may be the same, or they may be different, and this is not known until after inference. We now make the connected components procedure more explicit, with the $MaxMin$ procedure introduced in [19].

The edge classifier produces a binary output $C_{k,l}$ for each edge in the graph through Eq. 11. Let $P_{i,j}$ be a path through the graph G that connects vertices i and j. A path is made up of several edges (each connecting two vertices) which we denote $e_{k,l} \epsilon\, P_{i,j}$. If we consider all possible paths between vertices i and j, which we denote $\mathcal{P}_{i,j}$, then the $MaxMin$ procedure is:

$$C^*_{i,j} = \max_{P_{i,j} \epsilon\, \mathcal{P}_{i,j}} \; \min_{e_{k,l} \epsilon\, P_{i,j}} C_{k,l} \tag{16}$$

This procedure implements connected component analysis on edges: if $C^*_{i,j} = 1$ then vertex i and j are guaranteed to have the same label after inference. And if $C^*_{i,j} = 0$ then the labels are guaranteed to be different. This links the post inference output to the loss function:

$$\Delta(\hat{C}^*, Y) = |\mathcal{E}|^{-1} \sum_{e_{ij} \epsilon \mathcal{E}} \left| I(y_i = y_j) - \hat{C}^*_{i,j} \right| \tag{17}$$

A key property of the $MaxMin$ procedure is that it commutes with thresholding (theorem 1 in [19]):

$$MaxMin\left(I\left(f_w(x_i, x_j) > \theta\right)\right) = I\left(MaxMin\left(f_w(x_i, x_j)\right) > \theta\right) \tag{18}$$

This means that if we threshold the $MaxMin$ affinity graph, we are guaranteed that all components will be completely connected. This enables [19] to develop gradient descent learning algorithms for Eq. 17. They also show how the real valued $MaxMin$ affinity scores can be efficiently computed with a Minimum Spanning Tree calculation as illustrated in Figure 7.

In the experiments in this paper we use very simple classifiers related to decision stumps [25]. We generate a number of real-valued features. We transform each feature with the $MaxMin$ procedure. We then find the threshold that minimizes Eq. 17 through exhaustive search. We pick the feature and threshold that achieve the lowest error. Figure 8 illustrates the difference between simple classifiers designed with Equations 15 and 17. On the left, there is little difference in ROC-curves because the merge examples are based on local information. But on the right, where the merge examples are large connected components, designing with the global loss function (Eq. 17) becomes much more important.

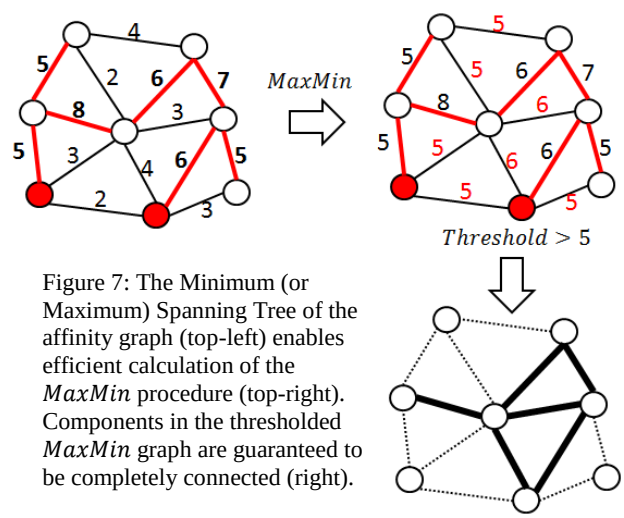

Figure 7: The Minimum (or Maximum) Spanning Tree of the affinity graph (top-left) enables efficient calculation of the $MaxMin$ procedure (top-right). Components in the thresholded $MaxMin$ graph are guaranteed to be completely connected (right).

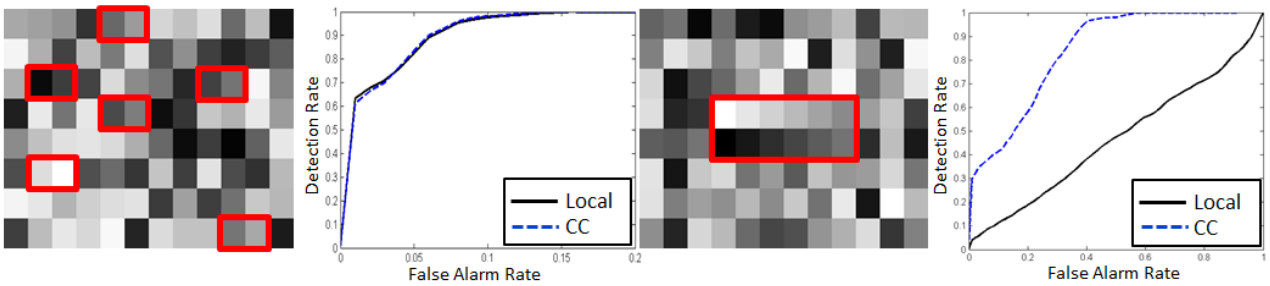

Figure 8: Synthetic problems that illustrate how the problem "connectedness" relates to the performance of classifiers designed with local (Eq. 15) and global (Eq. 17) loss functions. Red boxes indicate the merge examples used for training data. On the left the merges are based on a local difference in gradient. On the right, the merge is defined by a path of local differences.

3.5 Combining Segmentations

Ensemble methods, such as Bagging and Boosting, are an important tool in machine learning. They provide general purpose methods for increasing the accuracy of base-level, often application specific, classifiers. Applying these methods to the base-level segmentations described in the previous section is non-trivial. The base-level segmentations are guaranteed to produce a connected output. We would like the ensemble to have this property as well. A number of ensemble methods combine base-level classifiers using a weighted average, but this does not guarantee the connectedness of the ensemble. Research in consensus clustering, or clustering ensembles, may be applicable to this task [26], however these methods are often complex optimization problems in their own right, and it is not clear if they would provide advantages over training a more complex classifier to minimize Eq. 17 directly.

One ensemble method that can preserve the connectedness of base-level segmentations is a cascade. Cascade classifiers have been successfully applied to difficult object recognition problems [27]. The intuition in that application is that objects are very rare. This motivates a *one-sided* cascade, where at each stage a classifier predicts -1: the cascade exists and the example is predicted -1; or +1: the example is passed to the next stage. Any examples that are passed on by the final stage are predicted +1. The one-sided nature of the cascade can also be used to propagate the connectedness in our segmentations (illustrated in Figure 9).

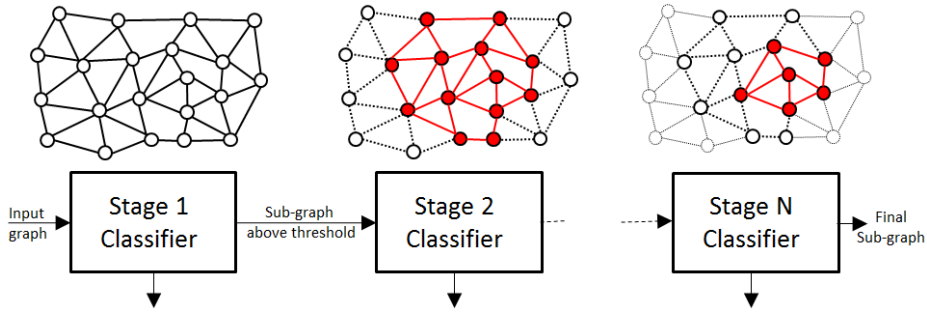

Figure 9: A cascade classifier can be used to improve the accuracy of base-level segmentations and guarantees the connectedness of the ensemble.

4. EXPERIMENTS

In this section we apply a number of the techniques presented in the last section to real-world mapping applications using three-color commercial satellite imagery. Our initial segmentation is generated with a watershed algorithm. It is tuned to over-segment the objects of interest in the different datasets. We then produce the graph representation described at the start of Section 3, where each vertex corresponds to a segment, and edges correspond to connected segments in the image plane. We calculate a number of attributes for the edges, and these are summarized in Table 1. Our edge classifier is a simple classifier using spheres on training samples as a hypothesis class [25]. This means, for each attribute we generate a set of features, where each feature is the distance to a particular training sample. Figure 10 illustrates the four different problems that we used to investigate our methods, and Figure 11 provides details of the merge statistics found in the four problems.

Table 1: Summary of attributes calculated for edges in the image graph.

Edge Attributes	
Edge Length	Pixel length of shared perimeter
Edge Linearity	Line length of shared edge / Edge Length
Average and Difference	For the two segments connected by edge
Color	Mean pixel value
Area	Number of pixels
Circularity	Ratio of area and perimeter relative to circle
Aspect Ratio	Ratio of major and minor axis of fitted ellipse
Aspect Angle	Angle of major axis of fitted ellipse

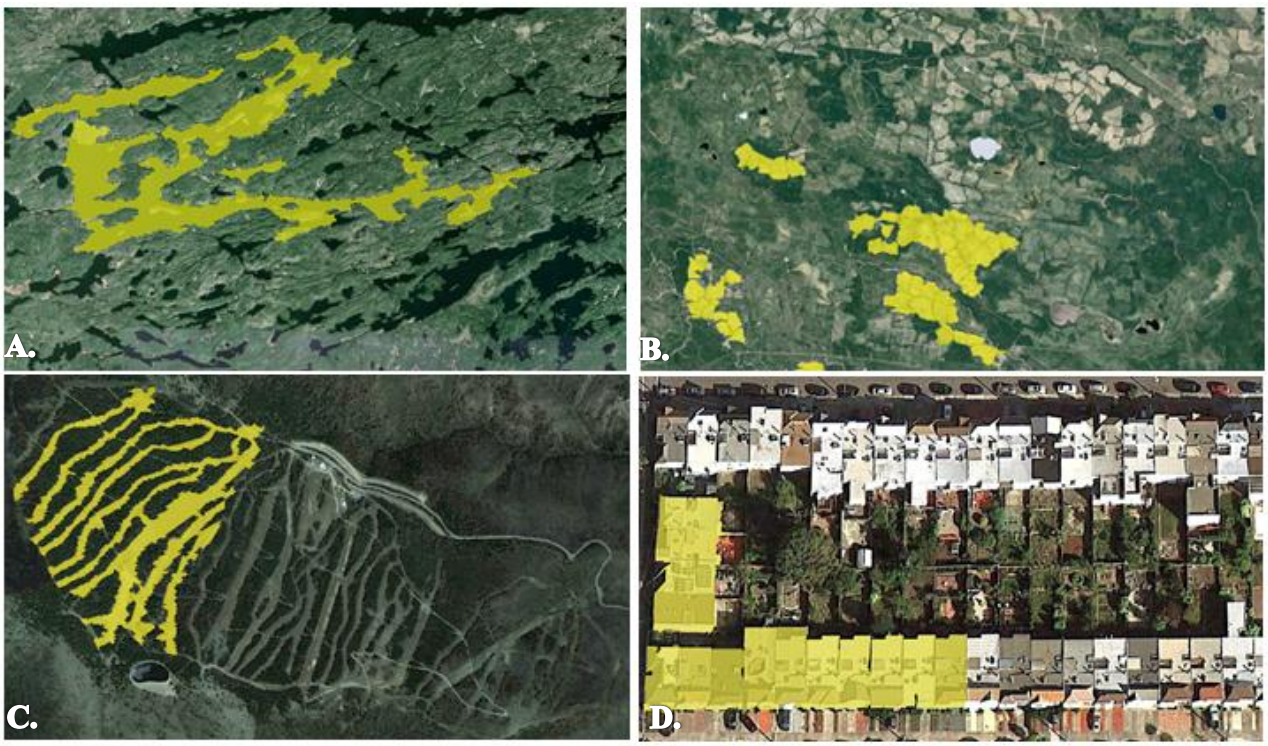

Figure 10: Images used in experiments with example merges overlaid in yellow. A) Lakes, B) Clear-cut forest, C) Ski-runs and D) Houses. Note that edges between merges are not visible in these images, e.g. between ski-runs and between houses.

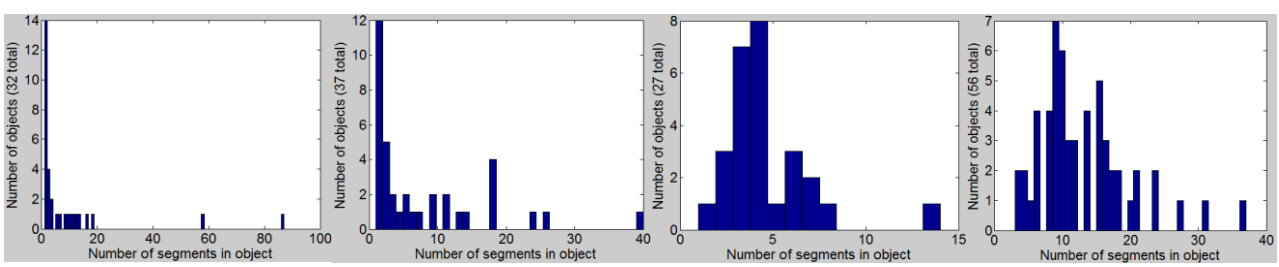

Figure 11: A) Lakes 281+/899- B) Clear-cut 284+/570- C) Ski-runs 119+/869- D) Houses 721+/ 910- Histograms for each problem showing the number of segments within each merge. The numbers below the histograms indicate the total number of positive and negative edges associated with all merges.

4.1 Loss Function Results

Our first experiment compares the performance of segmentations, designed with the two loss functions used in Figure 8 (Equations 15 and 17), on the real-world datasets. For each trial we randomly picked half the objects for training and used the other half for testing. We perform 20 trials and report the mean and standard deviation of the test scores in Table 2. We compare three different scores. The first column shows the performance of the edge classifier designed with the Connected Component (CC) loss function (Eq. 17) and reports error estimates with the same loss function. In the second column we show the edge classifier designed with the Local loss function (Eq. 15) but we report estimates from Eq. 17. The final column is the same classifier as the second column, but we report estimates from Eq. 15 (i.e. we do not run inference on the classifier output).

Table 2: Comparison of Connected-Component (Eq. 17) and Local (Eq. 15) loss functions on real-world data.

Dataset	CC-Eval/ CC-Design	CC-Eval/Local-Design	Local-Eval/Local-Design
Lakes	0.084 ±0.068	0.123 ±0.109	0.031 ±0.007
Clear-cut	0.169 ±0.020	0.288 ±0.114	0.162 ±0.020
Ski-runs	0.118 ±0.013	0.122 ±0.015	0.120 ±0.014
Houses	0.398 ±0.015	0.498 ±0.054	0.381 ±0.016

We observe that there is a large difference in performance between the two design criteria, justifying the additional complexity of the global method versus the local method. This performance difference is less pronounced in the ski-runs dataset which we attribute to low-connectivity within the merges. Much like the synthetic experiment on the left in Figure 8, we observe the two evaluation methods produce similar results. The low-connectivity also manifests in the large class imbalance reported in Figure 11.

4.2 Cascade Classifier Results

Figure 12 summarizes results on the four problems using the cascade classifier approach. The black-line indicates performance of the cascade, where each stage was trained to minimize the connected-component loss function in Eq. 17. For the blue dashed line we reweight the training data at each stage so that each class has equal weight. This is motivated by the fact that only positive examples are propagated through the cascade, and when there is large class imbalance premature convergence can occur. For the red dotted line we investigated a different weighting strategy based on Multiple Instance Learning (MIL) [28]. MIL places high weight on predicting *at least* one edge from each merge example in the training set instead of placing equal weight on all edges within the merge. Note, we are able to implement a MIL strategy for Eq. 17 directly because we are using simple classifiers based on exact error minimization [29]. Results are averaged over 10 trials and we report test error in terms of the absolute number of mistakes (corrections required) to provide a better idea of how these algorithms are working in practice.

The overall performance, and the relative performance of the three weighting strategies, varies greatly over the datasets. The overall performance we attribute to our relatively simple feature set and hypothesis class. It appears adequate for largely spectral problems (such as dataset A) but insufficient for more structured object problems (such as dataset D). We attribute the variability in weighting strategies to the variability in the merge statistics (as highlighted in figure 11) that spans both the typical number of segments within an object, as well as the typical connectivity within an object. The 50/50 strategy appears to do poorly, which could be attributed to the fact that its class probability priors are not representative of the dataset. However in problem C it did the best and this dataset is the most highly skewed. The MIL strategy appears to do best on datasets A and B, which both have a large number of pair-wise merges. In this regime the MIL strategy becomes similar to a 100% Detection Rate design criteria. The performance of standard weighting is best on dataset D, which has the most even class representation. In future work we plan to investigate these trade-offs in more detail and suggest a systematic cascade classifier design method that can be tailored to the characteristics of the merge problem will be required.

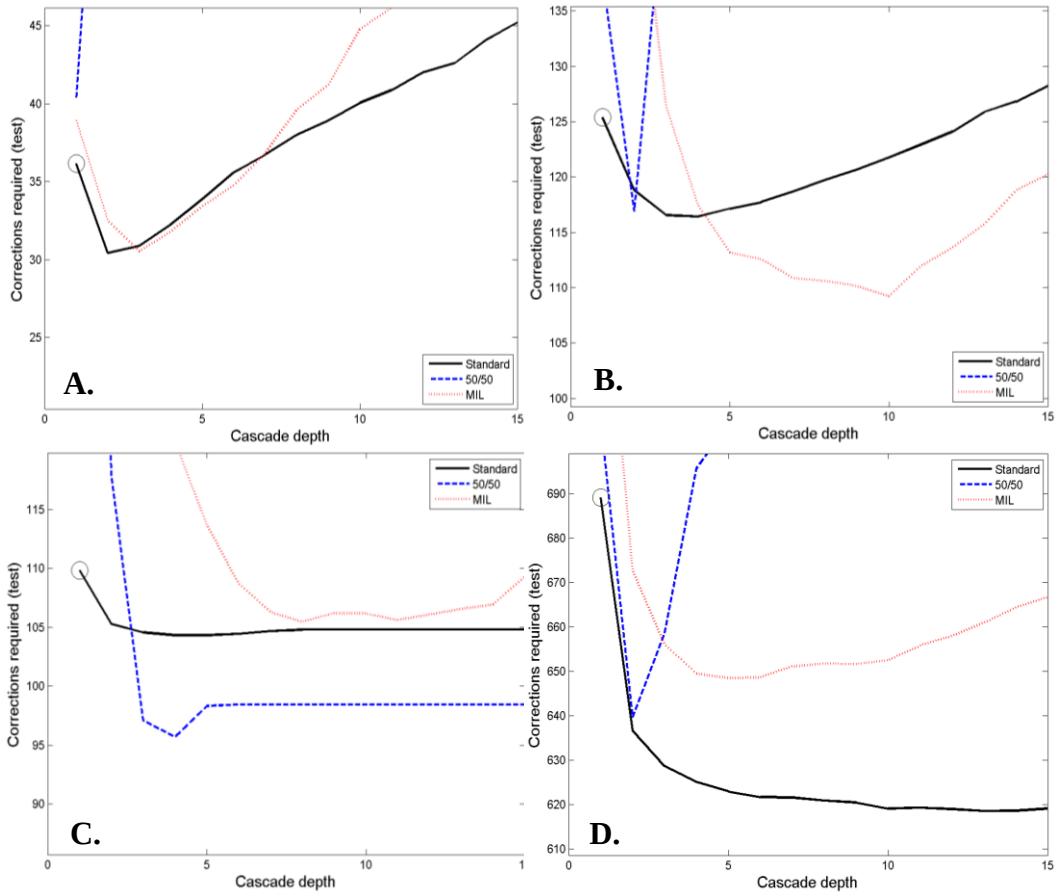

Figure 12: Performance of algorithms (in terms of the number of merge and split corrections required) as a function of cascade size for three different weighting strategies used at each stage.

5. SUMMARY

Tools that can *"learn to merge"* hold great promise for interactive mapping applications. In this paper we have outlined some of the key components required to provide this general purpose capability. Connected component energy functions provide a tractable, exact approach to the problem, and the improved performance we observed in experiment justifies the additional computational cost. In future work we plan to explore the design space further and investigate if other, more complex (yet still tractable), segmentation methods lead to even better performance. We also observed the large variability in merge statistics from one application to the next. This suggests a general purpose segment merging tool is a much more challenging tool to develop than a general purpose pixel classification tool. We plan to make the dataset used in this paper (as well as others) available online to facilitate further tool development.

REFERENCES

[1] Harvey, N.R., et al., "Comparison of GENIE and conventional supervised classifiers for multispectral image feature extraction," Geoscience and Remote Sensing, IEEE Transactions on, **40**(2): p. 393-404, (2002).

[2] Bucha, V. and S. Ablameyko, "Interactive Objects Extraction from Remote Sensing Images," in *Geographic Uncertainty in Environmental Security*, A. Morris and S. Kokhan, Editors. Springer Netherlands. p. 225-238, (2007).

[3] Hichri, H., et al., "Interactive Segmentation for Change Detection in Multispectral Remote-Sensing Images," Geoscience and Remote Sensing Letters, IEEE, **10**(2): p. 298-302, (2013).

[4] Caelli, T., A. McCabe, and G. Briscoe, "Shape tracking and production using hidden Markov models," in *Hidden Markov models: applications in computer vision*. World Scientific Publishing Co., Inc. p. 197-221, (2002).

[5] Porter, R., J. Theiler, and D. Hush, "Interactive Machine Learning in Data Exploitation," in *Technical Report*. Los Alamos National Lab, (2013).

[6] Hahn, H.K. and H.-O. Peitgen. "IWT - Interactive Watershed Transform: A hierarchical method for efficient interactive and automated segmentation of multidimensional grayscale images," in *Proc. of SPIE* (2003).

[7] Jain, V., H.S. Seung, and S.C. Turaga, "Machines that learn to segment images: a crucial technology for connectomics," Current Opinion in Neurobiology, **20**: p. 1-14, (2010).

[8] Wienert, S., et al., "Detection and Segmentation of Cell Nuclei in Virtual Microscopy Images: A Minimum-Model Approach," Sci. Rep., **2**, (2012).

[9] Wang, S., J. Waggoner, and J. Simmons, "Graph-cut methods for grain boundary segmentation," JOM, **63**(7): p. 49-51, (2011).

[10] Couprie, C., et al., "Power Watershed: A Unifying Graph-Based Optimization Framework," Pattern Analysis and Machine Intelligence, IEEE Transactions on, **33**(7): p. 1384-1399, (2011).

[11] Boykov, Y. and M.-P. Jolly. "Interactive Graph Cuts for Optimal Boundary & Region Segmentation of Objects in N-D images. ," in *International Conference on Computer Vision*, (2001).

[12] Grady, L., "Random Walks for Image Segmentation," Pattern Analysis and Machine Intelligence, IEEE Transactions on, **28**(11): p. 1768-1783, (2006).

[13] Vincent, L. and P. Soille, "Watersheds in digital spaces: an efficient algorithm based on immersion simulations," Pattern Analysis and Machine Intelligence, IEEE Transactions on, **13**(6): p. 583-598, (1991).

[14] Xue, B. and G. Sapiro. "A Geodesic Framework for Fast Interactive Image and Video Segmentation and Matting," in *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, (2007).

[15] EECS, B. "The Berkeley Segmentation Dataset and Benchmark," Available from: http://www.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/, (2012).

[16] Wick, M., et al. "Inferning 2012: ICML Workshop on interaction between Inference and Learning," Available from: http://inferning.cs.umass.edu/2012, (2012).

[17] Bansal, N., A. Blum, and S. Chawla, "Correlation Clustering: Theoretical Advances in Data Clustering (Guest Editors: Nina Mishra and Rajeev Motwani)," Machine Learning, **56**(1-3): p. 89-113, (2004).

[18] Finley, T. and T. Joachims, "Supervised clustering with support vector machines," in *Proceedings of the 22nd international conference on Machine learning*. ACM: Bonn, Germany. p. 217-224, (2005).

[19] Turaga, S.C., et al. "Maximin affinity learning of image segmentation," in *NIPS*, (2009).

[20] Jain, V., et al. "Learning to Agglomerate Superpixel Hierarchies," in *Proceedings of Neural Information Processing Systems*, (2011).

[21] Schwing, A.G., et al. "Efficient Structured Prediction with Latent Variables for General Graphical Models," in *Int.'l Conf. on Machine Learning (ICML)*, (2012).

[22] Tarlow, D., R.P. Adams, and R.S. Zemel, "Randomized Optimum Models for Structured Prediction," Journal of Machine Learning Research, **Proceedings Track 22**: p. 1221-1229, (2012).

[23] Vilain, M., et al., "A model-theoretic coreference scoring scheme," in *Proceedings of the 6th conference on Message understanding*. Association for Computational Linguistics: Columbia, Maryland. p. 45-52, (1995).

[24] Fowlkes, C., D. Martin, and J. Malik. "Learning Affinity Functions for Image Segmentation: Combining Patch-based and Gradient-based Approaches," in *CVPR*. Madison, WI, (2003).

[25] Cannon, A., et al., "Simple Classiers," in *Technical Report*. Los Alamos National Lab, (2003).

[26] Vega-pons, S. and J. Ruiz-Shulcloper, "A Survey of Clustering Ensemble Algorithms " International Journal of Pattern Recognition and Artificial Intelligence, **25**(03): p. 337-372, (2011).

[27] Viola, P. and M. Jones. "Rapid object detection using a boosted cascade of simple features," in *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, (2001).

[28] Dietterich, T.G., et al., "Solving the multiple instance problem with axis-parallel rectangles," Artif. Intell., **89**(1-2): p. 31-71, (1997).

[29] Cannon, A. and D. Hush. "Multiple instance learning using simple classifiers," in *Machine Learning and Applications, 2004. Proceedings. 2004 International Conference on*, (2004).