Towards Automatic Tree Ring Detection

Clément Guerner
cguerner@student.ethz.ch

Frederic Boesel
fboesel@student.ethz.ch

Luke Smith
lusmith@student.ethz.ch

Yutong Xiang
xiangy@student.ethz.ch

Abstract

The growth rate of trees, measured via tree ring width, is used in a wide range of downstream analyses, for example of climate. However, labeling tree rings in a sample still requires a tedious, time-consuming manual procedure. In light of this, we propose a solution to automate this task and allow researchers to save time. We take as input a high quality scan containing one or more core samples, to which we apply image segmentation methods to identify the individual samples, followed by classical image processing to retrieve the tree rings, and output a file containing the locations of the tree rings in the original image labeled via perpendicular lines. The final result is a reduction in the overall workload required for tree ring labelling, though manual checking is still required..

1 Introduction

1.1 Problem statement

Our stakeholder for the project was Justine Charlet de Sauvage. Our task was to assist Justine in automating the labeling of tree rings in samples taken in the field. Justine's workflow consisted of several steps:

1. Sample collection: collect two samples from each tree in the field, prepare them for analysis, and take a scanned image of samples (from one or more trees).
2. Ring labeling: for each core, manually label the tree rings. Rings are labeled with one or two points as follows:
   (a) Starting with the first ring on the left side of the image (to the right of the bark), find a straight line intersecting as many rings as possible with a roughly 90 degree angle.
   (b) When the line eventually goes beyond the boundary of the core (because the rings are generally at an angle), reset with a new line by labeling two points on a single ring.
   (c) Stop labeling either when the end of the sample is reached, the innermost center ring is reached, or an issue arises in the sample (large gap, staining, etc.) makes it impossible to reliably estimate the width between rings.
   (d) Labels stored in a .pos file format, with one .pos file per sample in an image.
3. Width measurement and post-processing: Justine ran a software called COFECHA [1] that computes the width between rings and assigns each ring a year (based on the year of the first ring, a manually entered data point). For a subset of samples belonging to the same tree species and collected in the same area, this software then compares the $year, width$ pairs across samples, returning correlation estimates and suggestions re. areas where adding or removing a ring might increase correlation for a sample.

Data Science Lab FS2022, Department of Computer Science, ETH Zürich.

Figure 1: Example scanned image with multiple cores

1.2 Project Goal

In our kickoff meeting, we defined the goal of the project with Justine to be: efficient, automated ring labeling, and width measurement with perpendicular lines between rings. To be specific, we were tasked with creating a tool that would take as input a raw scanned image with multiple cores (see Figure 1) and output separate .pos files with automatically detected labels for each sample in the image. Since the .pos file labels had to follow the perpendicular line format for width measurement, outputting width measurements was a trivial next step. Our tool had to be efficient and require as little manual input as possible to deliver quality results.

In order to reach this goal, Justine provided us with scanned images with 354 cores, 303 of which were manually labeled. For an example of a raw scanned image, see Figure 1. A labeled sample is shown in Figure 2. All of these samples were collected from conifer trees.

1.3 Methodology and Challenges

The task involved several challenges:

1. Read in image and automatically detect the samples in the image via an angled bounding box
2. Creating a cropped image for each sample, and for training purposes, rotating and shifting the labels provided by Justine
3. Predicting the location of tree rings in the sample
4. Measuring width between predicted rings via perpendicular lines through the detected rings
5. Outputting .pos file with one or two point labels for each ring

Prior to delving into these methods, we present the related work in Section 2. Section 3 presents tasks 1 and 2 in 3.1, task 3 is discussed in 3.2, and finally tasks 4 and 5 are discussed in SECTION NAME.

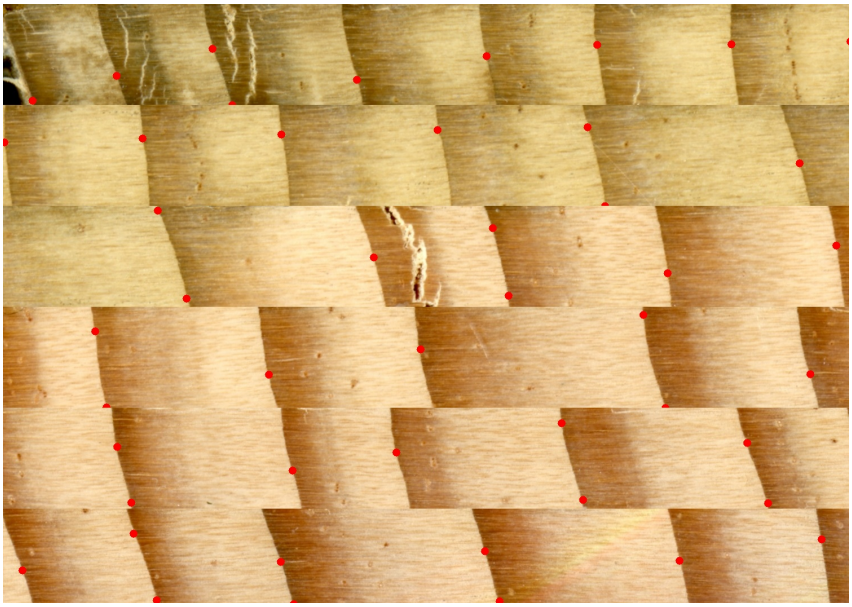


Figure 2: Example of a cropped sample with the labels provided by Justine

2 Related Work

2.1 Ring Detection

For ring detection, our research led us to two papers by the same authors. The first, published in 2017, is an image gradient-based method for detecting tree rings in samples like ours [2]. This heuristics-based approach had two notable advantages for us: it achieved .95 sensitivity, precision and DICE on samples of conifer trees, and it was evaluated on samples labeled in the same manner as our samples (one or two points per ring). The primary drawback of this approach is that it did not perform well on non-conifer samples, with on average .43 sensitivity and .57 DICE on ring-porous tree samples and roughly .85 sensitivity, precision and DICE on diffuse-porous tree samples. These performance drops can be explained by differences in the appearance of rings across these three categories. Conifers have very distinct, continuous rings that form nice lines and lend themselves well to image gradient-based methods.

In a subsequent paper, the authors leveraged deep learning methods to obtain a model that achieved significant performance gains for ring-porous trees, reaching roughly .95 sensitivity and precision for these trees [3]. Unfortunately, their model was trained on a dataset that included much more precise labeling of tree rings: each ring was fully labeled pixel-wise, along its entire length. Although we would have liked to experiment with such a model so that our tool could generalizable to more species, we chose to explore the gradient-based approach because of the label correspondence [2].

2.2 Core Cropping

2.2.1 Manual labeling

Before any kind of automated core detection, we had to manually label scanned images like Figure 1 to identify the location of the cores. We used LabelMe for this purpose, labeling each sample with various polygons (bark, inner crop, outer crop, center, cracks, etc.) [4].

2.2.2 Object Detection and Image Segmentation

Both detection and segmentation methods can be used to automatically detect and crop the core samples out of the input images. Based on our research and understanding of the task, we decided that the Faster R-CNN object detection architecture and its variants are potentially suitable for our needs. Section 2.2.3 is a brief introduction to such architectures.

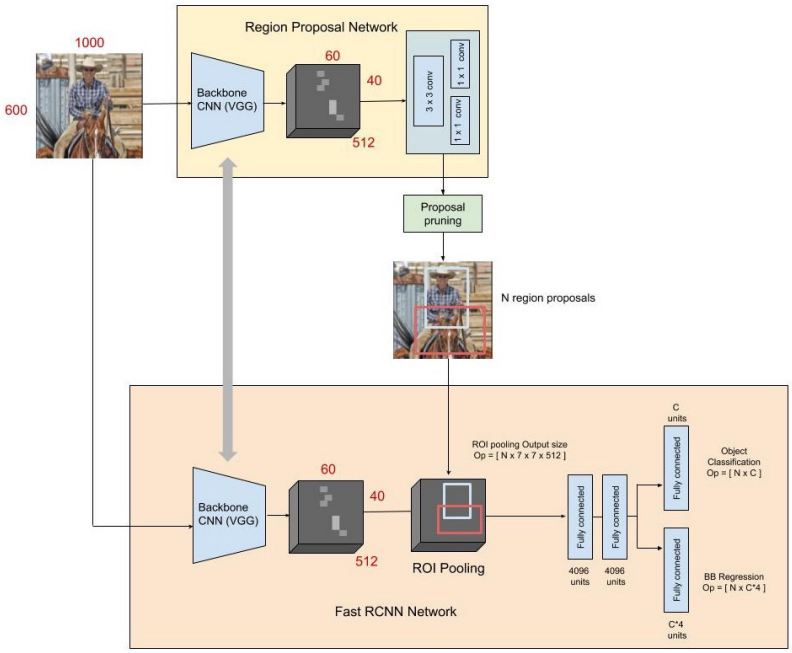


Figure 3: Faster R-CNN detection pipeline[5]

2.2.3 Faster R-CNN and Mask R-CNN

Faster R-CNN is a famous two-stage object detection model architecture. Compared to its predecessor, Fast R-CNN, it uses a CNN-based Region Proposal Network (RPN) module to generate region proposals instead of the selective-search algorithm. To elaborate further, RPN consists of:

1. some pre-trained CNNs (called a backbone) to extract features from the input images,
2. an anchor set placed on the feature maps suggesting the object locations and sizes, and
3. two following CNN heads doing bounding box regression and instance classification, respectively.

Then, after pruning, the proposals are to be leveraged by the Fast R-CNN module:

1. The module first uses an identical backbone that shares the weights with the RPN one to generate the feature maps from the input.
2. Then, the ROI pooling layer takes the regions corresponding to proposals from the feature maps, splits, and pools them to form the final features.
3. The features are fed to a fully-connected regression and classification head, respectively, to output the bounding box coefficients and class labels for detected instances.

Mask R-CNN further extends Faster R-CNN by adding another mask prediction head to segment the instances.

Figure 3 shows the whole Faster R-CNN detection pipeline:

3 Methods/Algorithms

3.1 Core Cropping

3.1.1 Challenges Addressed

The two major challenges for core cropping were:

1. **Extremely elongated shape of cores**: On the one hand, the core samples are extremely elongated, with a typical height-width ratio as low as 1:80; on the other, CNN architectures are by nature better at recognizing round- or square-shaped objects. Hence, we must curb such inclination of CNN and reduce the false-negative area of detected cores as much as possible during training.
2. **Rotated cores**: Although the core samples are supposed to be laid horizontally, small rotation angles are almost inevitable as the samples are placed by hand. Also, we have chosen quadrilateral polygons to annotate the cores. However, object detection models typically output strictly horizontal rectangular bounding boxes. If we want to cut out the core samples with high precision, we have to implement some workarounds to counter the effects of the polygon annotations and rotation angles.

For challenge 1, we successfully solved the problem by tuning the size and ratio of the anchors based on our prior knowledge of the cores. We will discuss the implementation details below in Section 3.1.4. For challenge 2, we used masks for Mask R-CNN to obtain the cores instead of rotated bounding boxes for Faster R-CNN, given that the angle values are relatively smaller and thus harder to regress, and masks are more exact and accessible from the polygons than rectangular bounding boxes.

3.1.2 Training Pipeline

By using Mask R-CNN, core cropping is designed in an end-to-end fashion. Specifically, the training pipeline includes:

1. **Preprocessing**: We transformed the `labelme` annotations into the desired input format. Specifically, we derived minimum horizontal rectangular bounding boxes from the `outer` polygons, which included the whole core samples. We converted the `inner` polygons, which were of our interest, into bitmasks. In this way, we could utilize Mask R-CNN without losing precision in the ground truth labels. Also, due to the large input size and limited memory, all images were resized to 15% of the original.
2. **Model Training**: By default, Mask R-CNN uses smooth-L1 loss for regression tasks, cross-entropy loss for classification tasks, and the average pixel-wise binary cross-entropy loss for segmentation tasks.

3.1.3 Inference Pipeline

The inference pipeline includes:

1. **Manual input**: for each new scanned image that the user wants to predict ring locations for, we require a CSV with a row for each core in the scanned image, with sample name and year the sample was collected (year of the first ring). We opted for this workflow with Justine because these data could not possibly be inferred from the images, and these data had to be present in the .pos file for downstream tasks.
2. **Preprocessing**: the input images were resized to 15% of the original.
3. **Model Inference**: The model outputs rectangular bounding box and mask predictions.
4. **Postprocessing**: obtain rectangular crops (note: not necessarily horizontal in the original images!) from the masks, and align the predictions from top to bottom with the data from the manual input CSV (names of the cores and first ring year). We use this information to infer the numbers of cores that should be in the image. With this information we filter all predictions (even ones with very low confidence scores) by size and alignment to infer the n most probable ones.

3.1.4 Implementation Details

We implemented the above algorithms and pipeline using `detectron2`, Facebook AI Research's next-generation library that provides state-of-the-art detection and segmentation algorithms[6]. In addition, the library includes a set of Mask R-CNN models pre-trained on the MS COCO dataset and many other functionalities that can facilitate pipeline development.

We used MSRA's original ResNet-50 + Feature Pyramid Network (FPN) as the backbone, with standard convolution and fully-connected heads for mask and box prediction. According to [6], such configuration obtains the best speed/accuracy tradeoff.

For the parameter setting, we inherited most default configurations except for:

- **Anchors**: As stated in Section 3.1.1, the anchor sizes and ratios were adjusted to match the typical characteristics of the core.
- **Pruning thresholds**: All thresholds that prune the predictions in RPN and ROI were set to zero since we wanted to process the predictions manually.

Apart from the changes above, some parameters that affect memory usage were also adjusted. Readers can consult `src/ringdetector/ringdetector/cropdetection/model_config.py` in the project GitLab repository for details.

Finally, the model was trained for 3660 iterations on two NVIDIA TITAN X GPUs. The dataset was randomly split into training, validation, and test sets at the ratio of 0.7:0.15:0.15.

3.2 Ring Detection

3.2.1 Classic Image Processing

For the ring detection we decided to adopt a classical image processing approach for reasons explained in Section 2. We were able to leverage some of the salient common features of the images we were processing and come up with some heuristics to improve accuracy.

We developed the pipeline presented in 3.2.2 using the cropped images obtained by manual labeling in labelme. An example of the inputs and corresponding desired ring locations is given in Fig 4.

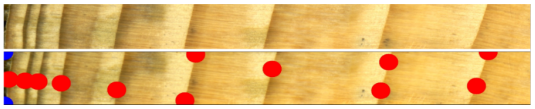


Figure 4: Example pipeline input and desired rings

3.2.2 Pipeline

The final pipeline we settled on contains the following initial steps,

- Denoising
- Contrast normalising (histogram equalisation)
- Canny edge detection
- Contiguous pixel collection into 'shapes'

The result was a number of 'shapes', with each shape being a collection of pixel coordinates that touch and that represent edge locations identified by Canny edge detection. Heuristics were needed to process these shapes as they were by no means of a high enough quality correspond directly to a ring, though the information was mostly there. These initial steps helped normalise the cores such that one set of heuristics provided sensible results for the majority of cores in our dataset.

The heuristics used to process these shapes and identify the rings are as follows:

- Keep right edge
  - Salient feature
    - Pixels to the left of each desired ring were lighter, while pixels on the right were darker. Noise was most present in the light part, hence canny edge detection often produced shapes with lots of noise on the left and a distinct edge on the right, and the right most edge corresponded to the actual ring location.
  - Heuristic

        * Keep only the right most pixel for each row of a shape.
- Remove inverted shapes
  - Salient feature
    * Pixels to the left of each desired ring were always lighter in colour, while the right was darker.
  - Heuristic
    * Remove shapes where on average the average of the n pixels left of the shape are darker than the average of the n pixels to the right.
- Merge shapes
  - Salient feature
    * Some shapes were not contiguous but clearly corresponded to a same ring. They were a sequence of lines separated at the tips by a number of empty pixels.
  - Heuristic
    * Fit a regression line through all shapes, if the regression lines of two shapes are close enough at their tips, and the angles of the regression lines are similar enough, merge the shapes.
- Filter length
  - Salient feature
    * Rings always span the height of the cropped core image.
  - Heuristic
    * Keep only the shapes above a certain length.
- Filter regression angles
  - Salient feature
    * The angles of the rings varied smoothly.
  - Heuristic
    * Fit a regression line to each shape and filter the shapes where the regression angles left and right are significantly different.

We ask that anyone interested in the specifics of the parameters chosen for particular stages in the pipeline to reference the published code.

An example of the pipeline in action is given in Fig 5.

4 Results/Discussion

4.1 Core Cropping

4.1.1 Evaluation Metrics

`detectron2` uses the same evaluation metrics as the COCO challenge for object detection and instance segmentation. Specifically, the metrics involve average precision (AP) and average recall (AR) under certain Intersection-over-Union (IoU) thresholds. Readers can refer to the COCO challenge website [1] for a detailed explanation of each metric.

4.1.2 Training Process

Figure 6 and figure 7 show the losses and evaluation results on the training set, respectively.

4.1.3 Results

Table 1 and 2 shows the detection and segmentation evaluation results on the validation and test sets, respectively. Figure 8 is an example visualization on the prediction results.

[1] https://cocodataset.org/#detection-eval

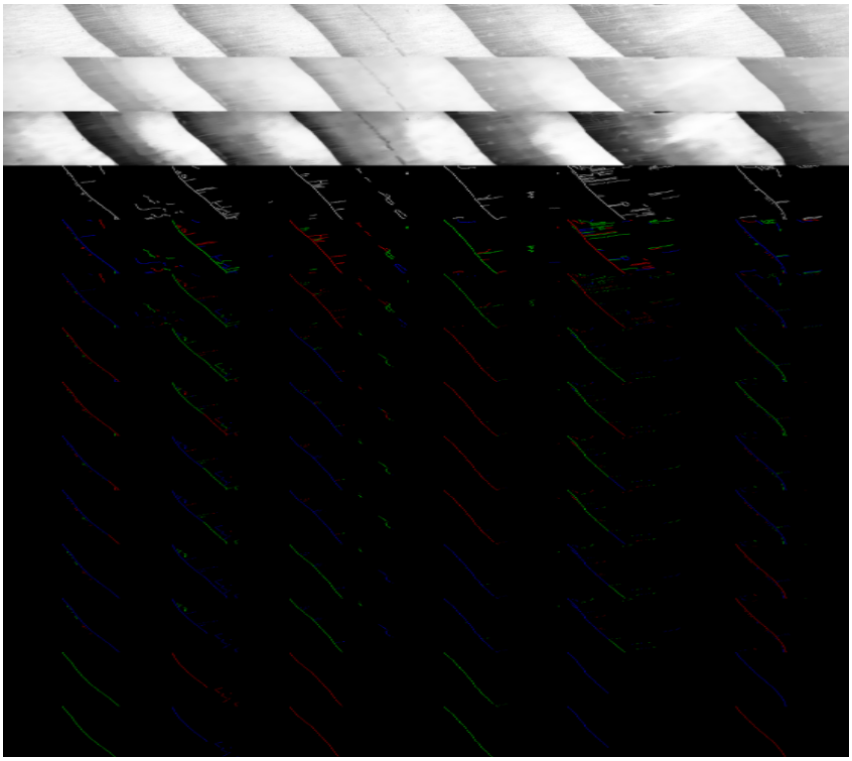


Figure 5: Example pipeline steps

dataset	AP	AP50	AP75	AR(maxDets=1)	AR(maxDets=10)	AR(maxDets=100)
validation	39.556	68.424	41.889	0.162	0.514	0.514
test	41.858	70.947	50.192	0.114	0.407	0.564

Table 1: Detection AP and AR on validation and test set

4.1.4 Discussion

The figures and tables above show that the model achieves quite satisfying results even though suffering slightly from over-fitting issues. The elongated-shape problem is also well solved. Note that the model's most interesting behavior is its false-negative tendency, which is unusual for Mask R-CNN models. Nevertheless, this is expected behavior for detecting elongated objects as the IoU values are more sensitive to the bounding box positions. Sensible predictions are more easily filtered out if the thresholds are too high. That is why we have chosen to filter the predictions manually in the end.

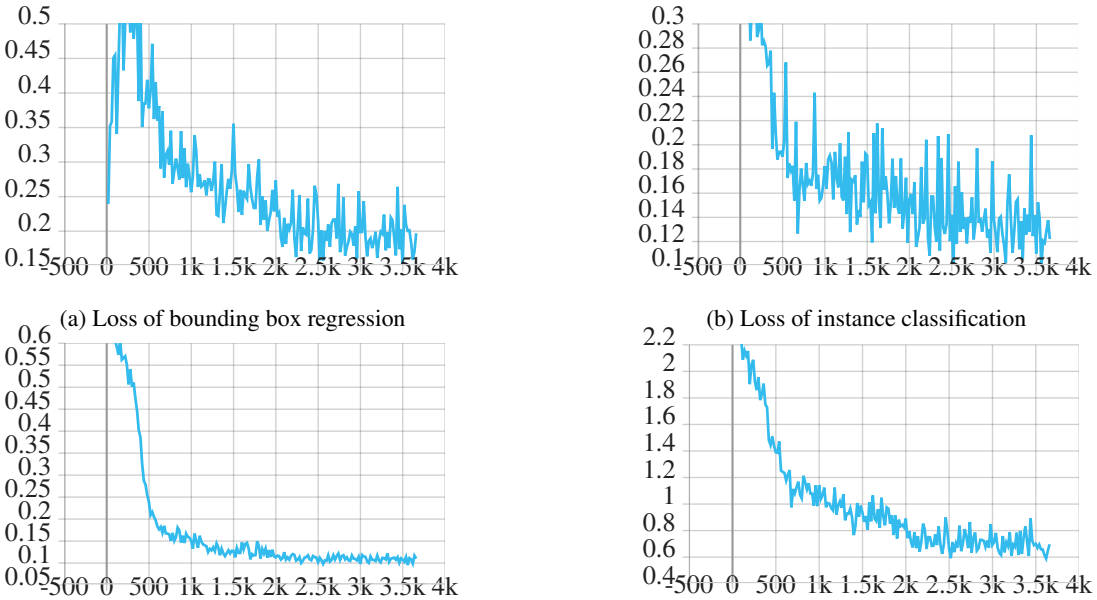
4.2 Ring Detection

4.2.1 Scoring method

We scored the output of our pipeline by fitting a linear regression line through each of the identified shapes. The shape was deemed a true positive if the line came within 10 pixels of one of the ground truth ring labels, a false positive if this was not the case. False negative were the ring labels that were not 10 pixels close to an identified rings.

4.2.2 Results

With this scoring method we could calculate the following for the whole dataset:



(a) Loss of bounding box regression

(b) Loss of instance classification

(c) Loss of instance segmentation

(d) Total loss

Figure 6: Detection and segmentation losses during training

dataset	AP	AP50	AP75	AR(maxDets=1)	AR(maxDets=10)	AR(maxDets=100)
validation	41.605	73.045	49.506	0.186	0.510	0.510
test	43.782	66.470	49.758	0.137	0.444	0.573

Table 2: Segmentation AP and AR on validation and test set

	Mean	Std
Precision	0.917	0.0807
Recall	0.933	0.0561

Table 3: Pipeline Accuracy

We have also plotted some results. First, Figure 9 shows us binned precision for the whole dataset. We can see that the tail is fairly thin and thus indicates, as did the low std in 3, that the pipeline performs well the majority of the time.

Secondly a plot of the recall is given in Fig 10.

Finally, we can see a scatter plot of precision vs recall in Fig 11. We note no particular correlation between the precision and recall.

4.2.3 Discussion

The precision and recall scores obtained match those obtained by Fabijanska et al. [2] for conifer tree samples, and are quite high for most samples. We note also that our samples had a lot more noise, e.g. due to cracks in the scanned cores, than those shown in Fabijanska et al [2]. We were also pleased to see a low std in the precision and recall scores, indicating that the majority of the time the pipeline performed well well.

Regarding the tradeoff between precision and recall, we chose to prioritize recall over precision after discussing this with our stakeholder Justine. She stated that it was easier for her to delete wrongly predicted rings than to add missed rings.

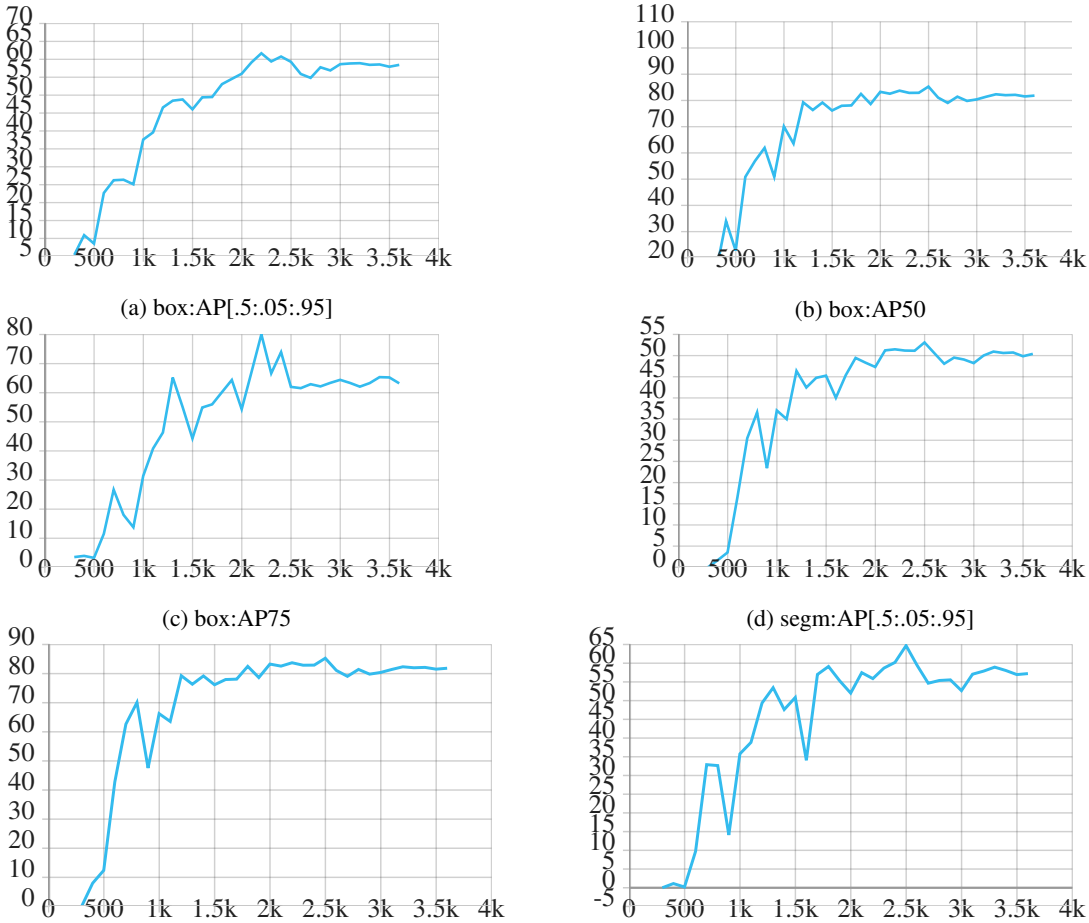


Figure 7: Detection and segmetation AP and AR during training

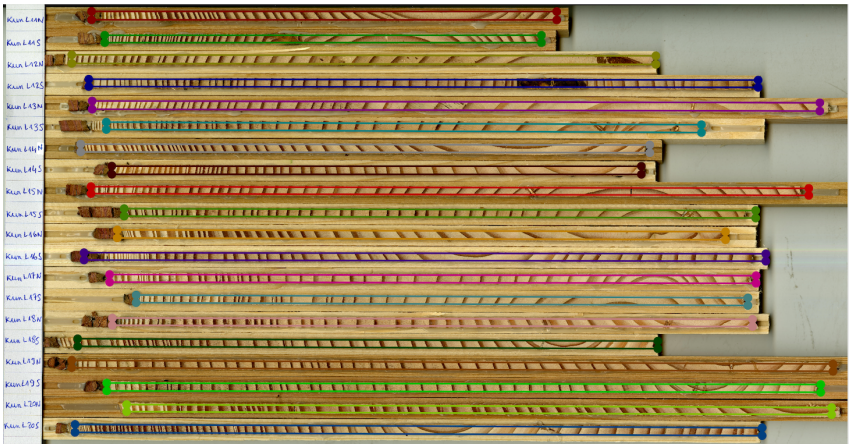


Figure 8: Prediction

It is interesting to note that the sample cores could become quite tricky, but we wanted to have a single pipeline that could handle all forms of input, rather than trying to classify how noisy/tricky a sample was and then processing it differently. A common issue, being that the light parts of the core actually contain a lot of dark spots (the same color as the right side of a ring), results in making the canny edge detector find far too much. In this case the 'keep right edge' heuristic worked well and

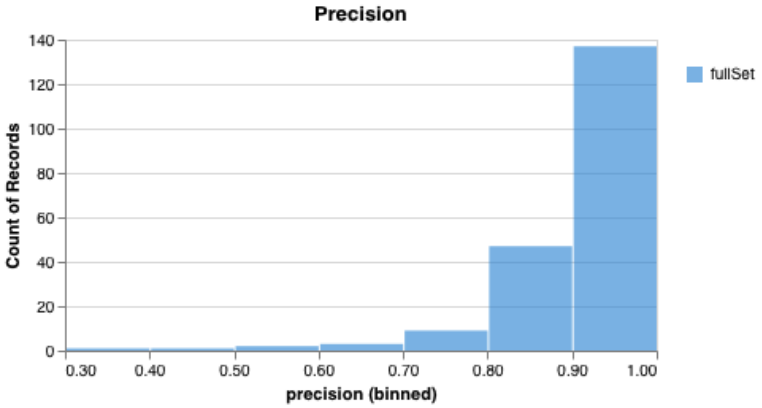


Figure 9: Precision

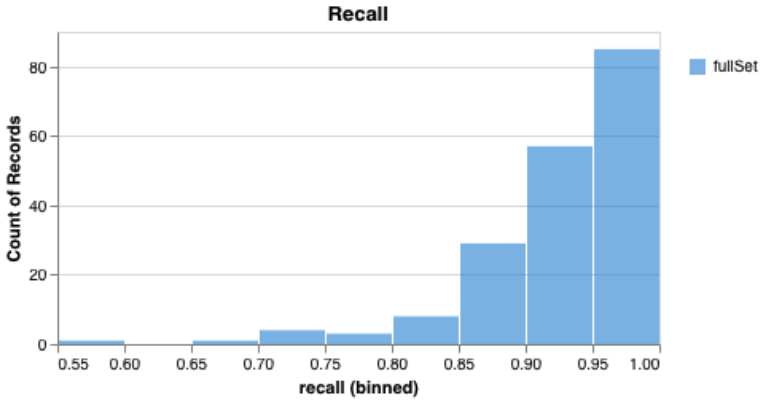


Figure 10: Recall

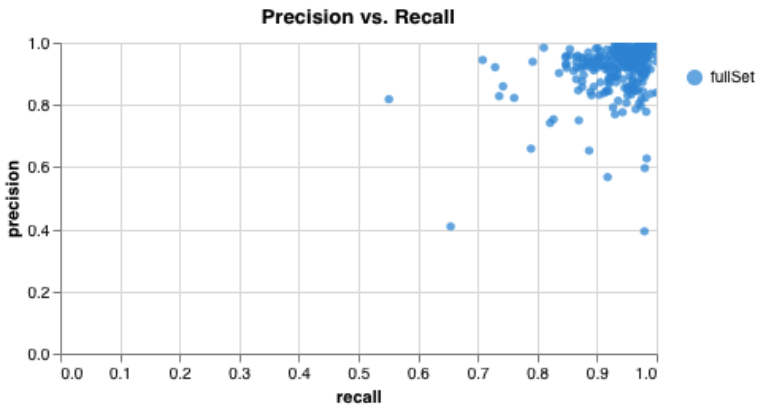


Figure 11: Precision v. Recall

we can see in Fig 12 a reasonable performance. In this figure, the lines and the dots are color coded as follows: blue lines indicate true positive rings, red lines indicate false positive rings, green dots

indicate ground truth labels matched to a true positive ring, orange dots indicate ground truth labels with no matching ring (false negatives).

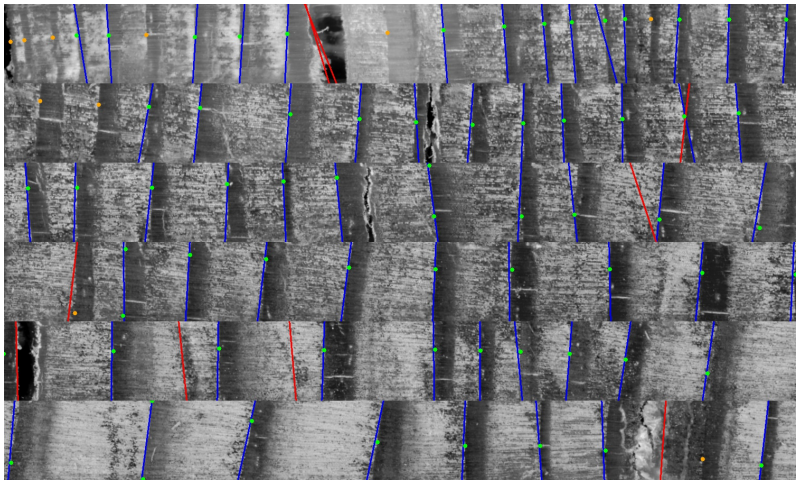


Figure 12: Example tricky sample results

One very common issue that we see in 12 is the result of gaps/cracks in the core (cracks present in rows 1,2,3,5 and 6 of the image). We did not develop any heuristics to try and deal with this issue, however we believe this might be possible. We did label all the cracks/gaps in our dataset and so another approach would be to train a segmentation model and remove any shapes intersecting an identified area by the model.

4.2.4 Ring Detection Pipeline Conclusions

Overall we believe that the results promise to reduce the workload of anyone using our tool. High precision and recall suggest that it will identify a large majority of the edges. Though we note that all edges must still be double checked (a much faster process than selecting their location manually).

5 Conclusion/Future Work

5.1 Conclusion

We have managed to provide a solution that significantly reduces the time needed to annotate tree rings compared to the analog workflow of our stakeholder. We present a pipeline that offers end-to-end detection (with an option to correct outputs in between steps) achieving high levels of accuracy for tree ring detection. The entire pipeline is displayed in 13. The user passes the image to predict on as well as a list of all the core names and sample collection years in form of a csv to the program, the program runs the core detection pipeline and optionally offers the user to correct the detected cores with LabelMe [4]. Then, the program runs the automatic ring detection on each of the detected cores and outputs a sample image for visual sanity checking as well as a pos file for each core, with perpendicular lines measuring width. The user can correct the detected rings by editing the pos file in CooRecorder [7].

Our final product can be run cross platform using Docker. We refer the user to the step by step manual to run the pipeline in the README of our repository.

5.2 Future Work

The work focuses on conifer trees, which have the most obviously defined rings. The approach could be extended to be able to handle new types of trees, by finetuning the core detection on such tree scans and improving the ring detection algorithm to be able to handle the harder to detect rings. This could possibly be tackled with a similar heuristics based pipeline or a Deep Learning based one, as shown in [3]. For this approach, careful labeling of entire tree rings via segmentation masks would

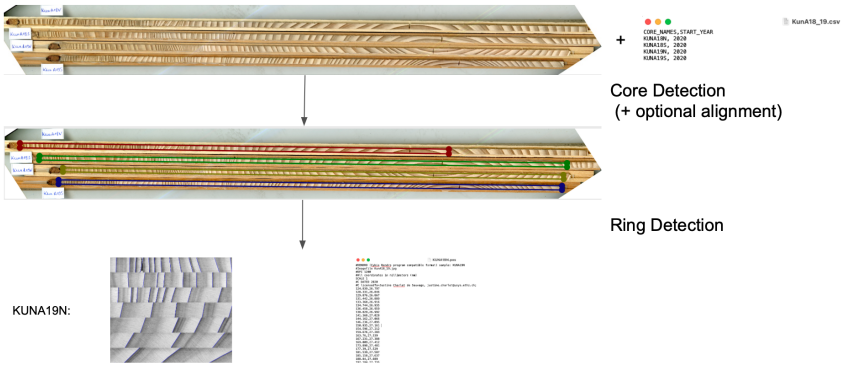


Figure 13: Full Pipeline

be required, thus many human annotation hours would be needed. Another starting point for future work could be the addition of an automatic crack detection system. Many cores have cracks that generate false positive rings and need to be subtracted from the width measure to get accurate growth phases. This still needs to be done manually and involves some guesswork, as to how much of the crack should be subtracted. A crack detection system could be trained, offering the user crack labels in the final output.

References

[1] Richard L. Holmes. Computer-assisted quality control in tree-ring dating and measurement. *Tree-Ring Bulletin*, 1983. ISSN 0041-2198. URL http://hdl.handle.net/10150/261223.

[2] Anna Fabijańska, Małgorzata Danek, Joanna Barniak, and Adam Piórkowski. Towards automatic tree rings detection in images of scanned wood samples. *Computers and Electronics in Agriculture*, 140:279–289, 2017. ISSN 0168-1699. doi: https://doi.org/10.1016/j.compag.2017.06.006. URL https://www.sciencedirect.com/science/article/pii/S0168169916312133.

[3] Anna Fabijańska and Małgorzata Danek. Deepdendro – a tree rings detector based on a deep convolutional neural network. *Computers and Electronics in Agriculture*, 150:353–363, 2018. ISSN 0168-1699. doi: https://doi.org/10.1016/j.compag.2018.05.005. URL https://www.sciencedirect.com/science/article/pii/S0168169918300413.

[4] Labelme. https://github.com/zhong110020/labelme.

[5] Shilpa Ananth. Faster r-cnn for object detection: A technical paper summary, Aug 2019. URL https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46.

[6] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2. https://github.com/facebookresearch/detectron2, 2019.

[7] Coorecorder - measure tree ring widths. https://www.cybis.se/forfun/dendro/helpcoorecorder7/.