universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

A multi-modal machine learning framework for forest plantation identification: supporting regulatory monitoring of illegal Christmas tree cultivation in Sauerland, Germany

verfasst von | submitted by

Felix Kausler BSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 856
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Kartographie und Geoinformation
Betreut von | Supervisor:	Ass.-Prof. Mag. Dr. Andreas Riedl

Acknowledgements

I would like to thank my thesis supervisor Ass.-Prof. Mag. Dr. Andreas Riedl, who provided guidance with helpful feedback and motivation throughout the writing process.

Furthermore, I would like to express my gratitude to my project colleagues Dr. Jan Dempewolf, Philipp Würth, and Rüdiger Müller. Dr. Jan Dempewolf offered expert advice in GIS analysis and machine learning. Philipp Würth and Rüdiger Müller supplied professional insights in forestry and Christmas tree cultivation – and a roof over my head while visiting Sauerland.

I also would like to thank Marc Messerschmidt and his colleagues from the *Regionalforstamt Oberes Sauerland*. Marc Messerschmidt started this project and his enthusiasm for the topic is contagious. While I could only spend a short time with him and his colleagues, I am deeply grateful for the detailed insights in Christmas tree cultivation and the local landscape, as well as for providing me with tremendously helpful sources on the matter.

A podcast from December 2023 about illegal Christmas tree plantations drew my attention to the topic. For this, I thank Jannis Carmesin, journalist of the ZEIT newspaper.

Finally, I am grateful to Emma Fullerton who provided useful stylistic input and proofreading.

Table of Content

Acknowledgements ..... **i**

Table of Content ..... **ii**

List of abbreviations ..... **vi**

Table of figures ..... **viii**

Table directory ..... **x**

Abstract DE ..... **xi**

Abstract EN ..... **xii**

1 Introduction **1**

2 Background of Christmas tree cultivation **4**

2.1 Historic development of Christmas tree cultivation in NRW 4

2.2 Christmas tree cultivation in the Sauerland 9

2.2.1 Location factors for Christmas tree cultivation 10

2.2.2 Production chain of Christmas trees 11

2.2.3 Cultivation cycle of Christmas tree plantations 11

2.2.4 Fighting pathogens 17

2.3 Impact of Christmas tree cultivation 19

2.3.1 Socio-economic impact 20

2.3.2 Ecological impact 22

2.4 Legal regulation of Christmas tree cultivation in NRW 25

3 Background of AI technology **28**

3.1 Definition of AI 28

3.2 Timeline of AI development 29

3.2.1 The first spring (1950s to 1970s) 31

3.2.2 The first winter (1970s) 33

3.2.3 The second spring (1970s to 1990s) 34

3.2.4The second winter (1990s) 40
3.2.5The third spring (1990s to now) 41
3.3 Contemporary image recognition methods 49
3.3.1Examples of state-of-the-art algorithms 49
3.3.2Challenges of CNN models in image recognition 56
4 ML and DL algorithms applications on forests and vegetation 60
5 Methodology 62
5.1 Data collection 62
5.1.1Normalised digital surface models 63
5.1.2True digital orthophotos 63
5.1.3Legal Christmas tree plantations 64
5.1.4Administrative boundaries 64
5.1.5Damaged areas 65
5.1.6Official woodland areas 65
5.2 Software 66
5.3 Outline of the workflow 67
5.4 Study area and evaluation area 69
5.5 nDSM processing 69
5.6 Generating training and inference datasets 72
5.7 Training and inference of YOLO models 77
5.7.1Overview of trained models 77
5.7.2Evaluation area sample test 78
5.7.3YOLOv5 run 03 – detection with RGB 81
5.7.4YOLOv12 run 08 – classification with RGB 83
5.7.5YOLOv8 run 04 – detection with RGBI 85
5.8 Alternative machine learning approach with Random Forest 86
5.8.1RF classifier with patches 87
5.8.2RF classifier with segmentation 89

5.8.3RF classifier models' results 91
5.9 Evaluation of EA inference results 92
5.10 Search for suspicious plantations 94
6 Conclusion 96
7 List of references 100
8 Annex 109
I Christmas tree cultivation background data 109
I.I Production data of a Christmas tree cultivation cycle (MESSERSCHMIDT 2024: 15-17) 109
I.II Legal classification of Christmas tree plantations based on § 1 LFoG (STANGE 2016: 20; translated) 110
II Maps and Datasets 111
II.I Westphalia and the Sauerland region 111
II.II Districts of the Sauerland (NRW) 112
II.III Municipalities of the Sauerland 113
II.IV Landcover in Bestwig-Heringhausen, NRW 2013 (MAURMANN 2015b) 114
II.V nDSM in the Hochsauerlandkreis 115
II.VI Legal Christmas tree plantations in the Hochsauerlandkreis and neighbouring areas 116
II.VII Damaged areas in the Hochsauerlandkreis and neighbouring areas 117
II.VIII Official woodland areas in the Hochsauerlandkreis and neighbouring areas 118
II.IX Study Area 119
II.X Evaluation area 120
II.XI Evaluation area sample dataset 121
II.XII Ground truth polygons in Evaluation area 122
II.XIII Training data examples 123
II.XIV Inference data examples 126
II.XV Evaluation area sample dataset results 130
II.XVI EA inference results 132
II.XVII Suspicious plantations – woodland and damaged areas 135

III Python code 138

III.I nDSM Processing (Visual Studio Code) 138

III.II Generating training dataset (Google Colab) 145

III.III Evaluation metrics calculation (example for v8 model runs on evaluation area sample test; Visual Studio Code) 152

III.IV YOLOv5 detection (v5_run03_025; Google Colab) 154

III.V YOLOv12 classification (v12_run08_cls_025; Google Colab) 160

III.VI YOLOv8 RGBI detection (v8_run04_TDv5_010; Google Colab) 168

III.VII RF patch classification (RF_2class_NDVI_64_100; Google Colab) 174

III.VIII RF segmentation classification (RF_Segmentation; Google Colab) 185

List of abbreviations

AI	artificial Intelligence
ASL	above sea level
AP	average precision
BB	bounding box
BWP	brushwood plantation
CNN	convolutional neural network
CT	Christmas tree
CTC	Christmas tree cultivation
CTP	Christmas tree plantation
CRS	(geodetic) coordinate reference system
DE	German
DL	deep learning
DLM	digital landscape model
DOP	digital orthophoto
EA	evaluation area
EN	English
FNs	false negatives
FPs	false positives
GIS	Geoinformation System
GSD	ground sample distance
GUI	graphical user interface
HP	hyperparameter
IoU	Intersection over Union
LM	Language model
LLM	Large Language model

LSTM	long short-term memory
ML	machine learning
nDSM	normalised digital surface model
NDVI	Normalised Difference Vegetation Index
NMS	non-maximum suppression
NLP	Natural-Language Processing
NN	neural network
NRW	North Rhine-Westphalia
PPs	potential positives
PR	precision-recall
LFoG	Forest Practices Act of North Rhine Westphalia; DE: *Landesforstgesetz für das Land Nordrhein-Westfalen*
relu	rectified linear units
RGB	red, green, blue (representation of light intensity in an image)
RGBI	red, green, blue, infrared
SVM	Support Vector Machine
TrueDOP	True digital orthophoto
TNs	true negative
TPs	true positives
TTA	test-time augmentation
WCS	Web Coverage Service
WFS	Web Feature Service
WMS	Web Map Service
WMTS	Web Map Tile Service

Table of figures

Figure 1. development of CTC areas in the Sauerland between 1970 and 2010 (MAURMANN 2015a) 5

Figure 2. disaster area after windstorm Kyrill near Neuenrade, NRW in 2007 (Wald & Holz NRW, n.d.) 6

Figure 3. brushwood plantation with noble firs near Bestwig-Velmede, NRW (Felix Kausler, December 2024) 8

Figure 4. natural growth of a mature noble fir (Abies nobilis) in an open location (BRAND n.d.) 9

Figure 5.approx. first or second year of cultivation on alleged CTP near Meschede-Freienohl, NRW (Felix Kausler, December 2024) 12

Figure 6. recent herbicide use in the foreground on alleged CTP near Meschede-Freienohl, NRW (Felix Kausler, December 2024) 13

Figure 7. discoloration from incorrect herbicide use; approx. fifth or sixth year of cultivation near Bestwig-Velmede (Felix Kausler, December 2024) 15

Figure 8. driving aisle and traces of harvest or thinning of stands; approx. fifth or sixth year of cultivation near Bestwig-Velmede, NRW (Felix Kausler, December 2024) 16

Figure 9. natural appearance of a mature Nordmann fir (Abies nordmannia) (Forest Research n.d.) 17

Figure 10. Christmas tree plantations near Bestwig-Heringhausen (Felix Kausler, December 2024) 21

Figure 11. Christmas tree plantations near Bestwig-Velmede (Felix Kausler, December 2024) 22

Figure 12. periods of AI development (DELIPETREV 2020: 3) 30

Figure 13. a perceptron: inputs with weights on the left , single output on the right (SEJNOWSKI 2018: 42) 32

Figure 14. schematic drawing of a Hopfield net (SEJNOWSKI 2018: 95) 35

Figure 15. LeNet neural network architecture (CHARNIAK 2024: 131) 38

Figure 16. relation between AI, ML, and DL (DELIPETREV 2020: 12) 41

Figure 17. language model based on recurrent NN (CHARNIAK 2024: 146) 43

Figure 18. error rate of winners in the ILSVRC; line indicates human error rate (CHARNIAK 2024: 135) 46

Figure 19. building block of residual learning (HE et al. 2015: 2) 47

Figure 20. simplified structure of transformer model (CHARNIAK 2024: 162) 48

Figure 21. Mask R-CNN architecture with ResNet (left) or FPN backbone (right) (HE ET AL. 2020: 389) 50

Figure 22. mask branch of Mask R-CNN for instance segmentation (HE ET AL. 2020: 387) 50

Figure 23. Mask R-CNN results (masks, bounding box, category, and confidence) on COCO test set (HE ET AL. 2020: 388) 51

Figure 24. architecture of different YOLO models (TIAN et al. 2025: 4) 52

Figure 25.oriented bounding boxes generated by YOLOv11 (Ultralytics 2024a) 53

Figure 26. Paper count for top ten CNN models (GALLAGHER & OUGHTON 2025: 11) .. 56

Figure 27. defaced stop sign to cause misclassification (CHARNIAK 2024: 137) 57

Figure 28. complete workflow of framework 68

Figure 29. nDSM processing workflow 71

Figure 30. CTP with mixed stands of different conifer species south of Ramsbeck, Hochsauerlandkreis (TrueDOP 2023) 72

Figure 31. training data image with artefact in the top left (TrueDOP 2023) 73

Figure 32. annotating single class training data with BBs in LabelImg 75

Figure 33. annotating two-class (young and old CTPs) training data with BBs in LabelImg 76

Figure 34. precision, recall and F1 score per YOLOv8 model run on evaluation area sample test 79

Figure 35. precision , recall and F1 score per YOLOv12 model run on evaluation area sample test 80

Figure 36. segment boundaries on positive training data GeoTIFF for RF classification. 89

Figure 37. precision, recall, F1 score, and correct ground truth rate per YOLO model run on EA dataset 93

Table directory

Table 1. definitions of artificial intelligence (recreated from RUSSELL & NORVIG 2016: 2) 28

Table 2. diagonal change of light intensity with one receptive field highlighted (recreated and modified table from CHARNIAK 2024: 87) 36

Table 3. convolution: kernel on the left, receptive field on the right (recreated and modified table from CHARNIAK 2024: 87) 37

Table 4. comparison of Faster R-CNN, SSD, and YOLO (recreated from MORE et al. 2024: 143) 55

Table 5. datasets used in the framework 62

Table 6. comparison of EA sample dataset inference results of model runs RF_Segmentation_6 and YOLOv5_run03_25 91

Table 7. evaluation results of the models v5 run 03, v8 run 04, and v12 run 08 92

Table 8. results of suspicious plantations per model 95

Abstract DE

Das Sauerland in Nordrhein-Westfalen ist mit einer Fläche von rund 4.700 ha das größte Weihnachtsbaum-Anbaugebiet Europas. Die Überwachung der Einhaltung der geltenden Vorschriften zur Anlage von Weihnachtsbaumkulturen (WBKs) stellt aufgrund derzeitiger arbeitsintensiver manueller Inspektionsmethoden eine Herausforderung für den zuständigen Landesbetrieb Wald und Holz NRW dar. Im Rahmen dieser Forschungsarbeit wurde ein Framework für ein Verfahren mit maschinellem Lernen (ML) zur automatisierten Erkennung von WBKs in Luftbildern unter Verwendung von YOLO-Architekturen entwickelt und evaluiert. Ein alternativer Ansatz mit einem Random-Forest-Classifier wurde ebenfalls untersucht und zeigte mit segmentierten Daten vielversprechende Ergebnisse, jedoch mit beschränkter Skalierbarkeit. Die Methodik umfasste eine systematische Bewertung von RF sowie YOLOv5, YOLOv8 und YOLOv12 in 36 Modellkonfigurationen unter Verwendung umfassender Datensätze, darunter True Digital Orthophotos (RGB und RGBI), normalisierte digitale Oberflächenmodelle (nDSM) und legale WBK-Polygone des Landesbetriebs. Um Trainingsdaten zu generieren, wurden diese Polygone einer höhenbasierten Filterung (Schwellenwerte von 0,3 bis 4 m) unterzogen, um eine Identifizierung aktiver Kulturen sicherzustellen. Die Ergebnisse zeigten, dass ML-Verfahren WBKs mit hoher Genauigkeit erfolgreich erkennen. YOLOv5 erzielte mit einem F1-Score von 84,32 %, einer Präzision von 74,10 % und einer Sensitivität von 97,81 % eine optimale Leistung. Alle drei Modelle wiesen durchweg hohe Sensitivität (> 97 %) auf und erfüllten damit die Projektanforderungen hinsichtlich falsch-negativer Fehler, da diese ein größeres Risiko darstellen als falsch-positive. Entgegen den Erwartungen führte die Implementierung von RGBI zu erhöhten Falsch-Positiv-Raten in städtischen Gebieten ohne entsprechende Verbesserungen bei Richtig-Positiven, was darauf hindeutet, dass RGB-Bilder ausreichende spektrale Informationen liefern. Die Integration von nDSM-Daten ermöglichte eine effektive höhenbasierte Filterung bei der Auswertung der Modellergebnisse, und Datensätze von Wald- und Schadflächen halfen bei der Identifizierung verdächtiger Kulturen unter den korrekten erkannten WBKs. Das Framework ermöglicht die Überwachung der Einhaltung der Vorschriften über ein automatisiertes, skalierbares Monitoring. Die Landesbehörde bestätigte die Erfüllung der Anforderungen, wobei die geplante Implementierung die praktische Anwendbarkeit für eine nachhaltige Waldbewirtschaftung in Nordrhein-Westfalen aufzeigt.

Abstract EN

The Sauerland region in North Rhine-Westphalia (NRW), Germany, represents Europe's largest Christmas tree production area, encompassing approximately 4,700 ha. The monitoring of regulation compliance of the establishment of Christmas tree plantations presents a significant challenge for the responsible State Agency for Forests and Wood NRW because of current labour-intensive manual inspection methods. This research project developed and evaluated a Machine learning framework for automated detection of Christmas tree plantations (CTPs) in aerial imagery using YOLO architectures. An alternative Random Forest classification approach was also investigated, showing promising results in segmentation-based workflows but with limitations in operational scalability. The methodology employed systematic evaluation of RF, as well as YOLOv5, YOLOv8, and YOLOv12 across 36 model configurations, utilising comprehensive datasets including True Digital Orthophotos (RGB and RGBI), normalised Digital Surface Models (nDSM), and legal CTP polygons from the State Agency. To generate training data, these polygons, were processed through height-based filtering (0.3 to 4 m thresholds) to ensure active plantation identification. Results demonstrated that Machine learning approaches successfully detect Christmas tree plantations with high accuracy. YOLOv5 achieved optimal performance with F1-score 84.32%, precision 74.10%, and recall 97.81%. All three models maintained consistently high recall values (> 97%), meeting the project requirements regarding false negative errors, as they represent greater operational risk than false positive detections. Contrary to expectations, RGBI implementation produced increased false positive rates in urban areas without corresponding improvements, indicating RGB imagery provides sufficient spectral information. Multi-modal integration of nDSM data enabled effective height-based filtering when evaluating model results. Datasets of woodland and damaged forest areas aided in identification of suspicious plantations in the correct detections of Christmas tree cultivation. The framework successfully addresses current enforcement limitations, providing automated scalable monitoring capabilities. The State Agency confirmed operational requirements were met, with planned future implementation demonstrating practical applicability for sustainable forest management in North Rhine-Westphalia.

1 Introduction

The Sauerland region in North Rhine-Westphalia (NRW), Germany, represents Europe's largest Christmas tree production area, covering approximately 4,700 ha of cultivated land. This concentration results from favourable soil and climate conditions, combined with extensive forest damage from disasters such as winter storm *Kyrill* in 2007, which created opportunities for establishing 3,000 ha of Christmas tree plantations (CTPs) in place of damaged mixed forests. Contemporary Christmas tree cultivation (CTC) predominantly employs Nordmann fir (*Abies nordmannia*) monocultures, maintained through intensive agricultural practices, including systematic fertilisation, herbicide applications, and pesticide treatments to optimise commercial yields.

The environmental implications of intensive CTC have necessitated comprehensive regulatory frameworks governing plantation establishment and management. The Forest Practices Act (DE: *Landesforstgesetz - LFoG*) of NRW has undergone multiple revisions, particularly following storm Kyrill, culminating in 2013 amendments that distinguish Christmas tree plantations from conventional forestry operations. These regulations require permits for new establishments on woodland areas and mandate sustainable certification for continued operation beyond 2028.

Despite regulatory oversight, compliance monitoring presents significant challenges for forest authorities. The State Agency for Forests and Wood NRW (DE: *Landesbetrieb Wald und Holz NRW*) has documented evidence of illegal Christmas tree cultivation, violating existing legal requirements. Current detection methods rely on manual inspection of digital orthophotos, with agency personnel searching for characteristic indicators, such as intensive tillage or consistent planting patterns. This approach has identified approx. 200 ha of illegal plantations, though the comprehensive scope of unauthorised activities remains un-clear due to methodological limitations. (CARMESIN 2023)

Manual detection approaches present substantial constraints regarding time efficiency, spatial coverage, and resource allocation. The labour-intensive nature of visual orthophoto interpretation limits regulatory agencies' abilities to conduct comprehensive monitoring across extensive forested landscapes.

These operational challenges coincide with significant advancements in Machine learning (ML) technologies across diverse application domains (MAXWELL et al. 2018). The development of sophisticated algorithmic frameworks, coupled with increased computational capabilities, has positioned ML as an effective tool for complex analytical tasks. Within the ML ecosystem, deep learning (DL) methodologies have emerged as the dominant approach for image recognition tasks, with convolutional neural networks (CNNs) established as the gold standard for image analysis applications (ALZUBAIDI et al. 2021).

Machine learning-based detection and classification methodologies offer advantages for remote sensing image recognition tasks, particularly regarding scalability, efficiency, and capacity to process high-dimensionality datasets, while identifying complex feature relationships (MAXWELL et al. 2018). Previous research has demonstrated successful applications of various ML algorithms, including Random Forest (RF) models and CNN architectures, for vegetation classification tasks involving tree species detection, forest type identification, and landcover mapping using diverse remote sensing data sources, including UAV imagery, LiDAR data, orthophotos, and satellite imagery.

Existing studies have employed ML methodologies for tree species identification using multispectral imagery, with research demonstrating effective differentiation of species through object-based approaches. Integration of multiple remote sensing data sources, including normalised Digital Surface Models (nDSM), RGB and RGBI orthophotos, and multi-temporal satellite imagery, has proven effective for achieving high classification ac-curacies. However, despite extensive usage of ML techniques for general vegetation classification and tree species identification, the specific application of these methodologies for detecting specialised conifer monocultures and distinguishing Christmas tree plantations (CTPs) from natural high forest remains unexplored in academic literature (as of August 2025).

The primary objective of this research project is to develop and evaluate a comprehensive workflow for automated detection of CTPs within the Sauerland region of NRW. This investigation will examine optimal integration of various remote sensing data sources with-in a GIS and ML framework to achieve reliable identification of Christmas tree cultivation areas. The research will specifically evaluate comparative performance of different algorithmic approaches, assess the impact of multispectral versus RGB imagery, and examine data

postprocessing strategies to prioritise output datasets for the project commissioner, the State Agency for Forests and Wood NRW.

Dr. Jan Dempewolf (geoecologist), Philipp Würth and Rüdiger Müller (forestry experts), as well as Marc Messerschmidt (forester at the State Agency) were consulted throughout the development process.

The project aims to answer the following research and working questions:

> *Can orthophotos be analysed using ML methods to detect Christmas tree plantations?*
> *Does involving RGBI data instead of RGB improve the model results?*
> *Can datasets of woodland and damaged areas help in finding suspicious plantations?*

The methodology encompasses systematic evaluation of contemporary deep learning architectures, with emphasis on You Only Look Once (YOLO) algorithms, which demonstrate exceptional performance in object detection tasks while maintaining computational efficiency suitable for large-scale applications. The framework development will incorporate multiple YOLO versions, including YOLOv5, YOLOv8, and YOLOv12, to assess relative performance characteristics for CTP detection under various operational conditions. An alternative ML approach with a Random Forest (RF) classifier was also tested on the data, to compare traditional ML and state-of-the-art DL methods.

To ensure practical applicability and robustness, this research project utilises a comprehensive dataset encompassing legal CTP polygons provided by the State Agency, high-resolution True Digital Orthophotos (TrueDOPs) with RGBI colour channels, normalised Digital Surface Models for height-based filtering, and damaged forest area and official woodland datasets for targeted analysis. The successful completion of this research will contribute to advancement of automated forest monitoring capabilities, while providing practical tools for regulatory compliance enforcement in Christmas tree cultivation management in NRW.

2 Background of Christmas tree cultivation

Over the last decades, Christmas trees have developed from a waste or by-product of forestry into the centre crop of lucrative silviculture business models in certain regions in Germany, as well as throughout Europe and North America. In the following chapters, the background of Christmas tree cultivation (CTC), especially the situation in North Rhine-Westphalia (NRW), Germany and the Sauerland region - will be explained. Furthermore, the ecological and socio-economic impact of CTC, as well as the legal regulations, will be explored.

2.1 Historic development of Christmas tree cultivation in NRW

Until the mid-20th century in Germany it was mainly specimen of the Norway spruce (*Picea abies*) that were harvested for use as Christmas trees (JAGEL & DÖRKEN 2019: 189). However, there was no specialised cultivation of spruces for this purpose. Instead, these boles originated usually from forest stand control (DE: *Läuterung*) and were thus a welcome by-product of ordinary timber production in the winter months (Landtag NRW 2013: 1; JAGEL & DÖRKEN 2019: 196). Following the post-war reforestation and afforestation efforts with Norway spruce stands, the supply became flooded (JAGEL & DÖRKEN 2019: 196). This led to the first specialised CTC in NRW in approximately 1955 (MAURMANN 2015a). Prompted by the oversupply of *Picea abies*, forest owners started planting blue spruce (*Picea pungens*) alongside Norway spruces in mixed stands (JAGEL & DÖRKEN 2019: 196). Blue spruces were of a more favourable shape and colour to the consumer, resulting in higher profit margins for forest owners (ibid.). In turn, this lead to the first specialised Christmas tree plantations (CTPs) in the region, as forest owners started planting whole stands consisting only of Blue spruce (MAURMANN 2015a). Meanwhile, there was also a general trend change in land use in the region. The Sauerland's and South Westphalia's low mountain ranges have vast amounts of marginal land (DE: *Grenzertragsböden*) (ibid.). These plots of lands are not or are barely profitable as a result of poor soil, excess or lack of water supply, and climatic conditions. Due to an agricultural oversupply and because of milk quota regulations in the 1980s, these lands lost their profitability for conventional agriculture (JAGEL & DÖRKEN 2019: 196).

Sheep farming, which previously took place on these marginal lands, also suffered a downturn (MAURMANN 2015a). While these developments caused agricultural lands in other parts of NRW to be abandoned, land owners in the Sauerland started afforestation with CTPs on them respectively (ibid.). At this point, Christmas tree producers focused more on fir plantations, namely Nordmann fir (*Abies nordmannia*) and noble fir (*Abies nobilis*) (JAGEL & DÖRKEN 2019: 196).

The combination of a rise in demand and increase of available arable area sparked a steep climb in CTC in the Sauerland. The majority of this region is located in the south of Westphalia (NRW) (see Annex *II.I*). Two municipalities of the neighbouring state Hesse also belong to the region (see Annex *II.III*). Between 1970 and 2014 the area of CTPs on agricultural land in Germany rose by 411% (MAURMANN 2015a). In Westphalia, the area even increased by 1,644% (ibid.). As CTC was expanding, the methods for increasing its yield also grew more vigorous. In the 1980s, this resulted in a surge of heavy pesticide use on CTPs (MESSERSCHMIDT et al. 2024: 2).

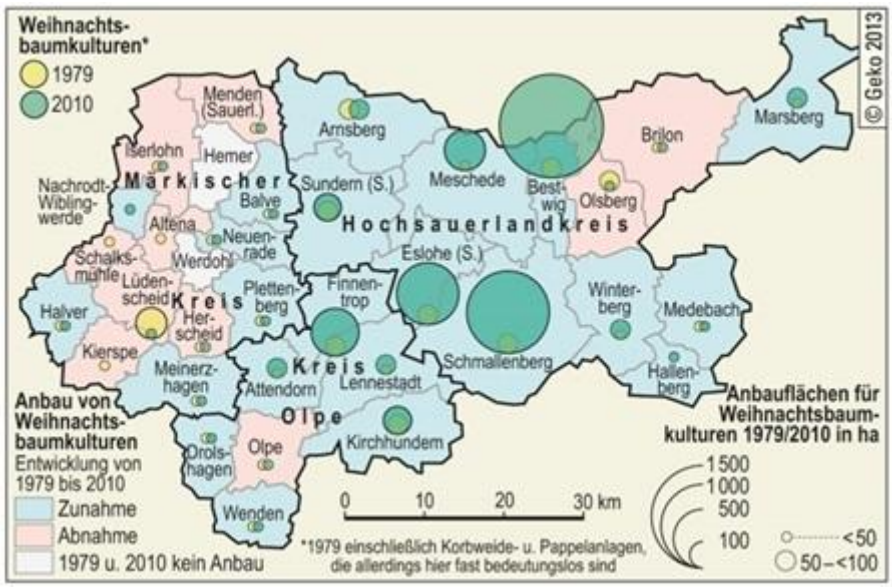


Figure 1. development of CTC areas in the Sauerland between 1970 and 2010 (MAURMANN 2015a)

In 2010, 28% of CTP area in Germany was in the districts (DE: *Landkreis)* Märkischer Kreis, Hochsauerlandkreis, and Olpe, which make up the majority of the Sauerland (see Annex *II.II*). Moreover, 20% of the cultivated area of Christmas trees in Germany was concentrated in just three municipalities (DE: *Gemeinde*): Bestwig, Eslohe, and Schmallenberg, where the area of CTC has doubled between 1999 and 2010. In the same timeframe the area

of CTPs in Meschede and Sundern rose even more significantly by 284% and 165% respectively. (MAURMANN 2015a)

However, while there is an overall increase in CTPs, with certain hotspots having increased considerably in size, there are also several municipalities where the area under cultivation has shrunk. This can be seen most predominantly in the west of the Sauerland, with the largest decrease in Lüdenscheid from more than 100 ha in 1979 to less than 50 ha in 2010 (see *Figure 1*).

While an increase in demand and changes in the agriculture sector have immensely contributed to the rise of CTPs in the region, natural disasters have also played a crucial role. Between 17 and 19 January in 2007, the winter storm *Kyrill* moved across the British Isles, Western, Central and Eastern Europe, leaving a path of destruction (FINK et al. 2009: 406). The maximum gusts of wind in Düsseldorf, the capital of NRW, reached a speed of 144 km per hour (ibid.).

Figure 2. disaster area after windstorm Kyrill near Neuenrade, NRW in 2007 (Wald & Holz NRW, n.d.)

The winter storm caused 46 fatalities (ibid.), with eleven lives lost in Germany alone (Landesbetrieb Wald & Holz NRW n.d.). The insured property damage amounted to €2.4 billion in Germany and between €4 billion and €7 billion in the whole of Europe (FINK et al. 2009: 406). *Kyrill* was the most consequential winter storm since *Lothar* and *Martin* in December 1999. However, it is important to note that there was more warning before the windstorm in 2007, compared to the 1999 storm, otherwise, the destruction would have been much greater (ibid.). The extreme weather event in 2007 was especially destructive to forests (see *Figure 2*), causing disaster areas of 50,000 ha in NRW (Landesbetrieb Wald & Holz NRW n.d.). A total of 62 million trees were destroyed in Central Europe due to windthrow and windsnap, the majority of which were spruces (FINK et al. 2009: 406).

After the calamity struck, many forest owners looked for an alternative to recover lost profits, as the timber from the disaster areas could only be sold at a fraction of the usual market value (Landesbetrieb Wald & Holz NRW n.d.). Since CTPs offer a quicker return on investment than conventional silviculture (MAURMANN 2015a), many people affected by *Kyrill* turned to CTC to mitigate the losses caused by the storm. This marked a turning point in CTC in NRW as CTPs were predominantly located on agricultural land before 2007. In the aftermath of the windstorm, until 2011, approx. 2,200 ha of woodland were transformed into CTPs, instead of being conventionally reforested (Landtag NRW 2013a).

This rapid transformation sparked the interest of NRW lawmakers as the changes in landscape became increasingly visible and stakeholders started to protest the development. As a result, the Forest Practices Act (*Landesforstgesetz NRW,* abbr. *LFoG*) was changed and stricter regulations on CTPs were implemented (see chapter *2.4*).

According to IT.NRW (2024), the total area of Christmas tree plantations amounted to 6,254 ha in NRW in 2023. MAURMANN (2015b) estimated approximately 8,000 ha of CTPs in 2015. The overall cultivated area of conifer plantations in NRW was approximated at 18,000 ha – 4,080 of which are in woodland area (Landtag NRW 2013a). The overall cultivated area also includes brushwood plantations. In these plots, noble firs (*Abies nobilis*) are almost exclusively grown for the production of ornamental brushwood (DE: *Schmuckreisig*). The brushwood is harvested every year in the autumn and the branches are made into advent wreaths or other seasonal decorations. The trimming results in a distinctive shape (see *Figure 3*), which differs noticeably from the noble firs' natural appearance (see *Figure 4*).

Figure 3. brushwood plantation with noble firs near Bestwig-Velmede, NRW (Felix Kausler, December 2024)

While this impacts the trees' vitality and interactions with their ecosystem, the boles remain in place after harvest and can be transformed into high forest. Thus, brushwood plantations (BWPs) are a separate issue to CTPs with Nordmann firs. Therefore, they were not part of the objective of this research project.

Figure 4. natural growth of a mature noble fir (Abies nobilis) in an open location (BRAND n.d.)

2.2 Christmas tree cultivation in the Sauerland

As noted in the previous chapter, the Sauerland has become the prime region for CTC in Germany, as well as the whole of Europe. The following chapter will highlight how Christmas trees are cultivated at present time and what conditions are needed for a successful plantation.

The majority of Christmas trees grown in the Sauerland are Nordmann firs (*Abies nordmannia*), with a market share of 80% as of 2021 (MESSERSCHMIDT 2024: 2). Other crop trees are the aforementioned Norway spruce (*Picea abies*), the blue spruce (*Picea pungens*), the noble fir (*Abies nobilis*), and the Austrian pine *(Pinus nigra)* (ibid.). Because of the dominance of the cultivation of Nordmann fir, this research project was focused on the identification of

the Nordmann fir and its cultivation. The other species are not included in further explanations if not stated otherwise.

Over the last decades, the (German) consumer market has shaped a specific appearance of Christmas trees as desirable. The *perfect* appearance includes evenly spaced whorls, resistant needles that do not drop easily, uniform and strong colouration (dark green or blue, depending on the species), and a strong scent. (MAURMANN 2015a) This appearance is defined as high quality in further discussion.

2.2.1 Location factors for Christmas tree cultivation

In order to accomplish a satisfactory yield of high quality Christmas trees certain location factors are very important. The first beneficial condition is a short vegetation period. This limits the likelihood of unwanted height growth, which would result in widely or unevenly spaced whorls (MAURMANN 2015a). If this occurs, the Christmas tree appears sparse or stooped. In order to ensure a brief vegetation period, plots are chosen that are within 400 to 900 m ASL (MESSERSCHMIDT et al. 2024: 5).

Another factor that can impact the appearance of the tree is the soil. Nutrient-poor, sandy loam soils without waterlogging are preferred, as soil rich in nutrients can trigger growth spurts in the shoots of branches, resulting in an uneven shape (MAURMANN 2015a; MESSERSCHMIDT et al. 2024: 5). Furthermore, plots of land that are unlikely to be subject to late frost are most desirable, as this can destroy entire stands of CTPs. Thus, producers try to avoid valleys, hollows, and south-facing slopes (MAURMANN 2015a; MESSERSCHMIDT et al. 2024: 5).

While marginal lands were widely cultivated with Christmas tree crops in the last century, they are no longer sought after in modern CTPs. Additionally, the inclination must not be too steep so it can still be accessed by farming vehicles. Vehicles also require a connection to roads, which means very remote plots of lands are not profitable. (MESSERSCHMIDT et al. 2024: 5)

A specific advantage of the Sauerland is its proximity to key markets like the densely populated Ruhr area (DE: *Ruhrgebiet*) or the Rhineland (JAGEL & DÖRKEN 2019: 197).

2.2.2 Production chain of Christmas trees

From its beginnings as a by-product, Christmas tree cultivation has since greatly intensified. Today, it involves a complex production chain spanning multiple countries.

It all starts with the seeds. In the case of the Nordmann fir, these are harvested in their original habitats in the Caucasus or the Black Sea coast in Turkey and then shipped to Europe. Likewise, blue spruce seeds are sourced abroad in the southwest of the United States. The seeds, however, are not planted in CTPs straight away. First, they grow for two years in seedling beds in nurseries in Northern Germany. This is followed by one to two years in the preschool bed. Here, seedlings are moved further south – in some cases to NRW – where the climatic conditions are more favourable for this growth stage. This used to take place largely in tree nurseries in Kirchhundem (see Annex *II.III*), but since the 1990s many producers moved north of the Sauerland to Lippstadt. After a total growth time of between three to four years the seedlings are moved to the actual CTPs in the Sauerland. They remain here for between eight to ten years, until they are harvested and sold. (MAURMANN 2015a)

2.2.3 Cultivation cycle of Christmas tree plantations

In the first year of cultivation, the soil is initially prepared for the plantation. For this, the ground is first mulched and then ploughed (MESSERSCHMIDT 2024: 7). These steps remove all vegetation and roots to reduce the vegetational competition for the crop trees and make the further cultivation easier. This is sometimes followed by harrowing and reconsolidation, which smooths out the surface and then compresses it slightly to reduce dehydration (ibid.). In addition to the tilling of planting lines, driving aisles of 3 to 4 m are mulched every 20 m (MESSERSCHMIDT et al. 2024: 5). Furthermore, the soil is analysed for its nutrients and acidity. The latter should be between pH 4.8 to 6 (ibid.), which can be modified by liming throughout the cultivation process, usually with low intensity at 2 t per hectare (ibid.). As for the nutrition, a chloride free fertiliser might be introduced at around 10 to 15 g per plant (ibid). Before plantation takes place, a fence with a height of around 2 m is erected to protect the plants from browsing deer (see *Figure 5*) (MESSERSCHMIDT 2024: 7).

Figure 5.approx. first or second year of cultivation on alleged CTP near Meschede-Freienohl, NRW (Felix Kausler, December 2024)

Afterwards, the tree seedlings are planted, which is done in evenly spaced lines with 0.8 to 1.2 m between each plant and each line (MESSERSCHMIDT et al. 2024: 5). This amounts to a total of between 5,900 to 13,200 seedlings per hectare (ibid.; MESSERSCHMIDT 2024: 7). The lines are planted in the prevailing wind direction or from north to south, which minimises the wind impact and optimises light exposure (MESSERSCHMIDT et al. 2024: 5). Depending on the root system, the size of the seedlings varies between 20 to 40 cm (deep-rooted) and between 40 to 70 cm (shallow-rooted) (ibid.) The Nordmann fir, for example, is a deep-rooted tree (STANGE 2016: 11).

When planting, trees of the same species and age are structured in stands together. While the placing can be done by hand, it is usually executed with the help of planters. This has been found to lead to a higher success rate in vitality than manual planting – around 90% instead of 40% (MESSERSCHMIDT et al. 2024: 5). After plantation, any unwanted vegetation remaining is mowed and herbicides, pesticides, and fungicides are sprayed against weeds and pathogens respectively (MESSERSCHMIDT 2024: 7).

Figure 6. recent herbicide use in the foreground on alleged CTP near Meschede-Freienohl, NRW (Felix Kausler, December 2024)

In the second year of the cultivation weeds are removed by using herbicides one to two times per year (see *Figure 6*) (MESSERSCHMIDT 2024: 7). This can also be achieved by manual trimming, mowing or grazing. The latter was more common in the 1990s using Shropshire sheep, as they do not browse the crops, however this practice is not widespread today (MAURMANN 2015b). High growing weeds are not just an unwanted competition for the tree seedlings, but can also increase vulnerability to late frosts (ibid.). If herbicides are applied correctly, they only eliminate undesired weeds, but if they are sprayed on young seedlings too early or at an angle, and saplings are contaminated with high amounts, it can result in stunted growth or discoloration of the trees (see light green foliage at the bottom of *Figure 7*). Completely bare soil is also not favourable in CTPs as it can lead to dehydration of the ground and to erosion during strong precipitation (MAURMANN 2015b). As well as herbicides, pesticides are also implemented at this stage to protect the young crops from pathogens (MESSERSCHMIDT 2024: 7).

However, even with all the counter-measurements there are losses after each first growth year. In some CTPs, supplementary seedlings are planted in the second year to make up for this deficit (MESSERSCHMIDT 2024: 7). Furthermore, fertiliser is added again along the planting lines, which is supposed to ensure optimal, stable growth conditions, supporting frost resistance and uniform colouration of the foliage (MESSERSCHMIDT 2024: 7; MESSERSCHMIDT et al. 2024: 5). The cultivation care also includes another cycle of mulching of the driving aisles, as well as the spaces in between the plant lines (MESSERSCHMIDT 2024: 7).

In the third year of the cultivation, the first pruning of the Christmas tree saplings usually takes place. Two thirds of fresh shoots are snipped manually when they have a length of 1 to 5 cm. This way, their length is reduced in the long term, resulting in an evenly shaped tree. When done correctly, the shoots continue growing again during the end of the vegetation period of the year, and the interference remains invisible to the untrained eye. This process is often repeated every year until harvest. (MESSERSCHMIDT 2024: 7)

In addition to this pruning technique, there is a variety of methods that take place throughout the seasons depending on the growth pattern of the trees. In the winter – sometimes as early as in the second year of cultivation – seedlings' tips are clipped if they are forming a bifurcation. In order to slow down their growth in height to between 40 to 50 cm, the terminal shoots are clipped in the summer. On the other hand, the terminal shoot might die off in some instances. Then, all branches but one are removed from the highest whorl and the remaining branch is tied up vertically to grow into the new terminal shoot. Furthermore, if the whorls of the trees grow too dense in the first years, some of them are trimmed to make for a more appealing appearance. All of these techniques combined can improve the yield by up to 80%. (MESSERSCHMIDT et al. 2024: 6)

Apart from pruning, the cultivation in the third year includes another rotation of herbicides, mowing of weeds, and mulching. The second cycle of analysis of soil samples also takes place. In accordance with the sample results, the ground is limed and fertilised. The upkeep also involves fence repairs. (MESSERSCHMIDT 2024: 7)

Figure 7. discoloration from incorrect herbicide use; approx. fifth or sixth year of cultivation near Bestwig-Velmede (Felix Kausler, December 2024)

During the fourth year of cultivation, the first harvests of small trees of lower quality may occur. This usually starts after the first frost in November (MESSERSCHMIDT et al. 2024: 6). Otherwise, year four includes the spraying of herbicides, fungicides, and insecticides, as well as the aforementioned usual tasks of upkeep (MESSERSCHMIDT 2024: 7).

In the spring of the fifth year of cultivation, together with the previously explained pruning techniques, the stomps are cleared of branches up to between 20 to 25 cm to ensure the best possible growth conditions by assisting in air circulation in the stand. (MESSERSCHMIDT 2024: 7; MESSERSCHMIDT et al. 2024: 6). Apart from this addition, the tasks remain the same including the spraying of fungicides and insecticides (MESSERSCHMIDT 2024: 7).

Usually, the sixth year of cultivation sees the first proper harvest of taller trees. At the same time, trees that are deemed unfit for sale are removed from the CTP to allow more room for the higher-quality trees in their final years of growth (MESSERSCHMIDT 2024: 7). This loss amounts to about 100 trees per hectare, whereas approximately 500 trees per hectare are

harvested and sold at this stage (see *Figure 8*) (ibid.: 7-8). For comparison to mature Nordmann firs, see *Figure 9*.

Figure 8. driving aisle and traces of harvest or thinning of stands; approx. fifth or sixth year of cultivation near Bestwig-Velmede, NRW (Felix Kausler, December 2024)

Usually, producers mark trees according to their quality level in category A, B, C, or D using coloured ribbons, with A as the highest quality (MESSERSCHMIDT et al. 2024: 6). In CTPs that allow customers to choose their own trees, they can use the coding system to select their favourite specimen. If there is no direct sale to the customer, the Christmas trees are packaged up in nets after the harvest in November and stored for shipment up to December. About 10% of Christmas trees are sold as so called *living trees*, meaning that their root ball remains on the trunk when purchased (MESSERSCHMIDT et al. 2024: 6).

These steps are repeated every year until the final, tenth year of cultivation. In the seventh year, approximately 700 trees per hectare are ready for sale and 100 unfit specimens are removed from the plantation. In the eighth and ninth year, the harvest numbers are the highest with about 1,200 trees sold each year. The removal of trees also rises on average in the

ninth year, up to 300 unfit trees. In the final year, around 600 trees can still be sold; the rest will be felled and removed. After ten years, the whole cycle of cultivation may start anew after ploughing and mulching the entire area. (MESSERSCHMIDT 2024: 8)

Figure 9. natural appearance of a mature Nordmann fir (Abies nordmannia) (Forest Research n.d.)

This means, in an average setup with 6,000 seedlings per hectare, the producer is able to sell about 4,200 Christmas trees after ten years, which amounts to a yield of 70%. For a detailed table of production data see Annex *I.I*).

2.2.4 Fighting pathogens

Pathogens in Christmas tree cultivation in Germany are comprised of various insect species and fungal diseases, which can wipe out entire stands from plantations if not protected of or treated effectively. The most hazardous insects are three species of aphids and one species

of weevil: *Dreyfusia sp.*, *Sacchiphantes viridis, Liosomaphis abietina*, as well as *Hylobius abietis* (MESSERSCHMIDT 2024: 9).

Dreyfusia sp. is an invasive species that was brought to Germany in the 1800s (SCHRÖTER et al. 2009: 2). Young trees of up to 4 metres are mostly affected (ibid. : 4). Thus, CTPs are particularly vulnerable to them. *Dreyfusia sp.* attack young shoots or needles in late spring or summer and can cause withering of the upper branches or even death of the whole tree or groups of trees (ibid.: 4-5).

Liosomaphis abietina, also known as *Elatobium abietinum*, mainly attacks spruce species; *Picea pungens*, the blue spruce, is especially prone to infestation and species of firs are also sometimes affected. Contamination can already occur in the tree nursery and then cause outbreaks at the CTPs or in storage facilities. It was in this way that *Liosomaphis abietina* was introduced to Iceland from imported Christmas trees from Denmark and to New Zealand by seedlings coming from the UK or USA. *Liosomaphis abietina* attack the needles of older shoots and cause them to yellow and wither. This can result in severe defoliation, making the Christmas trees unsuitable for sale. (CABI 2021)

A contamination with *Sacchiphantes viridis* leads to similar symptoms: yellowing and finally shedding of foliage. This infestation additionally builds swelling growths called galls on the shoots, in which the aphids lay their eggs and later larvae hatch. The pea to hazelnut-sized galls remain on the shoot and lignify over time. As opposed to *Liosomaphis abietina, Sacchiphantes viridis* attack only spruces – and in some cases *Larix spp.* – but not firs. (LOHRER 2013)

The largest of the insect pests on CTPs is the large pine weevil (*Hylobius abietis*), which belongs to the same family (*Curculionidae*) as the bark beetles (*Scolytinae spp.*). It is not limited to the contamination of pines, as its name would suggest, but it attacks all conifer species. The biggest risk factors for infestation is the vicinity to calamity areas, as the larvae of *Hylobius abietis* develop in dying conifer roots. During the vegetation period, the adult weevil can infest the trunks of seedlings or saplings of conifers that are up to approx. five years old. (BORK 2025)

The aforementioned pathogen insects are killed using insecticides that work through ingestion or contact with the poison. For this purpose, agents from chemical classes of

carbamates, pyrethroids, pyridine-azomethin, or neonicotinoids are used (MESSERSCHMIDT 2024: 9). In order to minimise the amount of chemicals introduced to the environment, these insects are supposed to be fought only curatively. Sometimes multiple interventions must take place, if the infestation persists. However, there are also cases in which producers take preventative measures by dipping the branches into contact insecticides to protect them from the large pine weevil (ibid.).

Apart from insect pathogens, the biggest threat to Christmas tree plantations are fungal diseases. The most prolific being spruce needle rust (*Chrysomyxa abietis*), fir needle rust (*Pucciniastratum epilobii*), and Kabatina-needle cast of firs (*Kabatina abietis*) (ibid.).

Pucciniastratum epilobii originates on willowherbs (*Epilobium spp.*), from which its spores are transferred by wind to the needles of firs, causing them first to discolour (rusting) and later to shed. Usually they do not lead to the loss of the entire tree, but the damage to their foliage reduces the quality of trees for sale in CTC (LOHRER 2013). *Chrysomyxa abietis* causes a similar disease pattern, but instead of firs, it infests spruces (NIERHAUS-WUNDERWALD 2000). Infection with *Kabatina abietis* leads to a syndrome called CSNN (Current season needle necrosis), which is displayed by a reddening or browning of the tips of the outer half of the needles (CECH et al. 2009).

Furthermore, there are fungal infections that result in branch or needle mould, which are attributed to *Rhizoctonia spp.*. These types of mould lead to discolouration and later shedding of the needles (ZÄPERNICK 2006).

Christmas tree producers intervene against fungal diseases both preventatively and curatively with contact agents (MESSERSCHMIDT 2024: 9). Since fungi species thrive in moist conditions, CTPs that are located in areas where annual precipitation exceeds 1,000 mm are at higher risk of infection. More widely spaced seedlings or the pruning of branches at the stomp aid air circulation and ensure stands dry quicker after rainfall (ZÄPERNICK 2006).

2.3 Impact of Christmas tree cultivation

Christmas tree cultivation influences its immediate surroundings or – if widely prevalent – the whole region. This impact can be observed not only on the ecosystem, but also on the

socio-economic setting. In the following chapter, these effects will be explored with a focus on the Sauerland.

2.3.1 Socio-economic impact

In favourable locations in the Sauerland, CTPs are a profitable business model – more so than conventional agriculture and animal husbandry. Average annual profits for the latter are only about €800 to €1,000 per hectare, whereas CTC can bring in about three times that at €3,000 per hectare. Thus, Christmas tree producers can pay higher rates for land leases to land owners than crop or livestock farmers. (MAURMANN 2015b)

This has a striking consequence: conventional agriculture has stagnated and in turn a shift in the cultural landscape of the Sauerland has taken place. Entire valleys – e.g. near Heringhausen – became dominated by CTPs in the 1970s (see *Figure 10*), which caused a change in legislature in 1980 (see chapter *2.4*) (ibid.). Since they are visually very distinctive and thus differ from more common landcover such as forests or agricultural land, they stand out in the landscape (see *Figure 10* and *Figure 11*). However, the appeal of a scenery is subjective. The Sauerland boasts a thriving tourism sector with consistently high numbers of visitors (Sauerland-Tourismus e.V. 2025), thus, the consequences of the change in the cultural landscape remain so far unmeasurable.

The significance of Christmas tree plantations for the region is undisputable. Overall, there are approximately 600 producers in all of NRW (MLV NRW 2023). 4,000 ha of CTPs were operated by 40 large to medium-sized enterprises and another 4,000 ha were run by 390 small producers with a respective area of less than 20 ha each in the Hochsauerlandkreis alone as of 2014 (MAURMANN 2015b). Large and medium-sized farming operations are usually specialised in the production of Christmas trees and brushwood, whereas the smaller farmers have a mix of crops or work part-time (ibid.).

According to the German Federal Association of Christmas tree and ornamental brushwood producers (DE: *Bundesverband der Weihnachtsbaum- und Schnittgrünerzeuger in Deutschland*) (n.d.), the average price for a Christmas tree in 2023 in Germany was €35 to €40. With approximately 30 million trees sold each year – a third of which are from NRW (MLV NRW 2023) – this suggests total sales of approx. €1 billion. Respectively, NRW would account

for €330 million alone. According to a report by Statista (RICHTER 2016), producers make approx. 8.5% profit per tree. Taking into account the turnover, this would mean earnings of about €28 million for the farmers in NRW each year.

Figure 10. Christmas tree plantations near Bestwig-Heringhausen (Felix Kausler, December 2024)

More conservative estimations put the annual revenue of the industry in Germany at €700 million as of 2015 (JAGEL & DÖRKEN 2019: 197) and €800 million as of 2024 (Sparkasse 2024).

However, all of these numbers are approximations as there is no concrete information available about the total turnover or profits of Christmas tree production specifically for the Sauerland or NRW. More so, inconsistent data is published regarding the total size of CTPs. The revenue calculation based on the statistics provided by the German Federal Association of Christmas tree and ornamental brushwood producers (n.d.) and the MLV NRW (2023) would suggest an approximate area of 9,000 to 10,000 ha of designated Christmas tree

plantations if you take the figure for annual profit (€3,000 per hectare) provided by MAURMANN (2015b).

As well as contributing to the GDP, this industry also creates jobs. In 2014, the production of Christmas trees provided 1,000 permanent employments and 2,500 seasonal jobs (MAURMANN 2015b). Even with technological developments taken into account, a majority of effort that goes into cultivation, harvest, packaging, and sales still requires a big labour force.

Figure 11. Christmas tree plantations near Bestwig-Velmede (Felix Kausler, December 2024)

2.3.2 Ecological impact

As stated previously, CTPs are visually very different from other land uses. This also translates directly to a variance as an ecological feature in the landscape interacting both with flora and fauna. These effects have been analysed extensively by researchers specifically in the Sauerland, and also other parts of Germany and Europe, where CTPs are prevalent. Since

the cultivated species and the conditions vary significantly in other geographic locations like North America, studies from outside Western and Central Europe were not considered.

KÄMPFER et al. (2022) sought to understand the role of CTPs as a habitat for declining farmland bird species, namely the Common linnet (*Linaria cannabina*), Tree pipit (*Anthus trivialis*), Yellowhammer (*Emberiza citrinella*), and Woodlark (*Lullula arborea*). In this study, which was conducted in the northwest of the Hochsauerlandkreis, they found that all four threatened bird species' territories had high proportions of CTPs favouring them over other landcover types such as agricultural land, grassland, or woodland. The Woodlark and the Tree pipit preferred young stands of CTPs (less than 6 years), whereas the Common linnet and the Yellowhammer established their territories in older stands (more than 6 years) (KÄMPFER et al. 2022: 374). Additionally, the Yellowhammer was found in brushwood plantations. The researchers found that specifically for the threatened species studied, the usage of herbicides for weed control provides more food compared to management through sheep or mechanical removal. The insectivorous and granivorous birds need a mixture of bare ground and herbal plants that flower to produce seeds (ibid.: 376). Nevertheless, the researchers noted that the regular use of herbicides is proven to have harmful effects on biodiversity and ecosystem processes (ibid.).

GAILLY et al. (2017) researched CTPs as a habitat for threatened farmland birds, as opposed to intensive farmland, in the Ardenne region in southern Belgium. Their model organisms were the Common linnet (*Linaria cannabina*), Yellowhammer (*Emberiza citrinella*), Dunnock (*Prunella modularis*), and Stonechat (*Saxiola torquatus*). They found that in areas with limited hedge density (less than 70 m per hectare), CTPs increase local diversity among these species during breeding season (GAILLY et al. 2017: 94-95). However, the plantations provide low food supply for the nesting birds which is sourced elsewhere. Thus, the researchers concluded that if CTPs dominated the landscape, there would be a scarcity of food for the birds (ibid.).

In a study on beetle biodiversity by HAGGE et al. (2019), a group of researchers investigated anthropogenic habitats (maize fields, CTPs, and spruce plantations) in two study areas in southern and eastern Bavaria respectively. They found that across all parameters of beetle diversity (e.g. number of species, abundance, number of threatened species), the more intensified the anthropogenic landscape, the lower the diversity. A total of 8649 individual

beetles were recorded, which belonged to 484 species. Of these 484 species, 352 were found in spruce plantations (4935 individuals), 253 in CTPs (1524 individuals), and 200 in maize fields (2190 individuals) (HAGGE et al. 2019: 567). Only 26 of the 484 species that were recorded in the study areas were of conservation concern, representing less than 1% of all individuals sampled (ibid.: 570). This "[biodiversity] decline with land use intensity is linked to fertilisation, pesticide use, frequent operation of machinery and rotation period length" (ibid.: 568).

Alongside the exploration of implications of CTPs on local fauna, it is also pertinent to consider the flora. STREITBERGER & FARTMANN have investigated this in two studies. In 2020, they analysed habitat structure, plant-species richness, and soil samples of CTPs and compared them to other dominant land types (windthrow areas, spruce forests, and intensively managed grasslands) in the study area (Hochsauerlandkreis). In 2021 they did a similar analysis, however they compared different CTP management types with each other.

In their 2020 study, STREITBERGER & FARTMANN found that CTPs have high levels of potassium and phosphorus in their soil, similar to the grassland habitats in the study areas, however, unlike grasslands, CTPs do not have high soil contents of nitrogen. Regarding phytodiversity, CTPs display a unique mix with a high number of ruderal species and neophytes (ibid.: 880-881). This is largely determined by the herbicide use. Thus, CTPs are of low significance for threatened plant species (ibid.: 882) On the other hand, windthrow areas contain a highly diverse range of plant species, which thrive in acidic soils. This diversity is promoted through low anthropogenic influence and natural succession within the habitats. Therefore the researchers concluded that the establishment of CTPs should be avoided on windthrows due to their high structural heterogeneity (ibid.: 883).

When comparing different types of Christmas tree plantations and brushwood plantations (conventional CTPs in open landscapes and on former windthrow areas, organic CTPs, and BWPs), STREITBERGER & FARTMANN (2021) found that they have similar soil conditions. However, phytodiversity and habitat structure of conventional CTPs varied strongly in conventional CTPs compared to organic CTPs and BWPs. There was a much higher number of detected species (organic CTPs: 130, BWPs: 129, conventional CTPs on open landscapes: 105, conventional CTPs on windthrow areas: 81) as well as more stress-tolerant plant species in organic CTPs and BWPs (ibid.: 181). The researchers state that the greater richness

in species may be linked to the reduced or lack of herbicide use (ibid.: 185). Overall, none of the four plantation types proved to be important habitats for threatened plant species (ibid.: 186).

In conclusion, CTPs have remarkable implications in both a socio-economic and an a eco-logic sense. They provide employment and a staggering revenue for the region. However, the visual domination in some valleys has led to criticism (MAURMANN 2015b). As an eco-system, CTPs can provide a refuge for threatened farmland birds (KÄMPFER et al. 2022; GAILLY et al. 2017), but this positive effect is diminished if there is a high density of CTPs in an area. For beetles, HAGGE et al. 2019 found no such advantageous influence. While they are favourable over maize fields as habitats, the possible biodiversity is limited due to intensive agricultural interventions such as pesticides or fertilisers. STREITBERGER & FARTMANN reached a similar conclusion regarding phytodiversity in their studies from 2020 and 2021. Compared to windthrow areas, the herbicide use in CTPs restrains species-richness. Organic CTC and BWPs contain a higher phytodiversity, however they do not provide an important habitat for threatened plant species.

2.4 Legal regulation of Christmas tree cultivation in NRW

The legal regulation of Christmas tree cultivation in North Rhine-Westphalia has been revised several times over the past decades. This is in part due to the stark increase of CTC area in the region, particularly in the Sauerland. As the area of CTPs expanded, the impact of CTC on the environment and the region was more seriously scrutinised, resulting in stricter regulations.

When the Forest Practices Act (DE: *Landesforstgesetz,* abbr. *LFoG*) was introduced in NRW in 1950, the law did not distinguish Christmas tree plantations or brushwood plantations from ordinary high forest (Landtag NRW 2013a: 2). That means that producers of Christmas trees were not bound to additional regulations differentiating from conventional silviculture, but at this time the industry was in its infancy (see chapter *2.1*).

CTPs became more prevalent in the 1970s on previously agricultural lands, which led to the tightening of regulations in 1980. CTPs established on agricultural land were henceforth regarded as woodland and, as such, required a permit from the forestry office (DE: *Forstamt*)

(MAURMANN 2015b). However, some Christmas tree producers found a way around this new rule. If their request was denied, they instead proclaimed their plantation to be a tree nursery, which did not require a permit to be established. This development led to a high number of tree nurseries in areas dominated by CTPs (see Annex *II.IV*) (ibid.).

In 1987, the status of CTPs was revised again based on where they were situated. From this time forward, CTPs (as well as BWPs) that are located on agricultural land were not regarded as woodland anymore (MLV NRW 1994). Thus, they do not fall under the jurisdiction of the forest authority but require a permit from the Lower landscape authority (DE: *Untere Landschaftsbehörde*). This also applies to tree nurseries, that are larger than 1 ha and are used for CTC or brushwood cultivation (Kreisverwaltung Warendorf n.d.). To establish CTPs or BWPs on agricultural land compensatory measures had to be met, such as planting a strip of broadleaf trees around the plantation, which makes up 10 to 15% of the total plantation area (ibid.). This revision of the regulation suggests that the authorities wanted to counteract the aforementioned loophole of tree nurseries, however it completely disregarded the status of CTPs and BWPs located on woodland.

This finally changed in 2013, when the *LFoG* was altered following the spatial expansion of CTPs in the aftermath of winter storm Kyrill in 2007 (see chapter *2.1*): § 1 *LFoG,* as of 2013, states that CTPs and BWPs on woodland are not regarded as forest by law (Landtag NRW 2019: 10). There are exceptions to this regulation: CTPs and BWPs that have an accumulated size of a maximum of 2 ha per operator and are at least 60 m (Landtag NRW 2013b: 2) from another CTP or BWP. Woodland area cultivated with CTPs or BWPs underneath power lines is also exempt (Landtag NRW 2019: 10).

Furthermore, CTPs or BWPs on woodland that existed before 11 December 2013 are allowed to be maintained without an additional permit until 31 December 2028, which applies to plantations with a combined area of 3,374 ha (Landtag NRW 2020: 2-3). Christmas tree producers that want to continue their operation beyond 2028 are required to establish a contract with the forest authority before 12 December 2016, which states that the CTC is continued sustainably (STANGE 2016: 7). This must be certified under the PEFC standard for Christmas tree cultures (see PEFC 2017) or a similar independent certification. The standard's regulations are stringent considering the common use practices of intensive CTC (see chapter *2.2.3*) and include limits on herbicides, pesticides, and fungicides, restrictions

regarding vehicle use, and erosion protection (STANGE 2016: 29). Furthermore, the inner edges of plantations have to be lined with endemic shrub and tree species or successional areas that are exempt from mechanical removal (ibid.: 28). If the whole of a plantation is used for CTC, it must not be larger than 2 ha to avoid large barren clearings in the landscape. The operator can only employ qualified personnel, in line with labour law and union regulations (ibid.). From the original 3,374 ha of plantations on woodland that were protected until 2028, about 2,100 ha were concluded in contracts with the forest authority by their operators (Landtag NRW 2020: 3). The remaining CTPs that are not part of a contract must be converted to high forest by 2029.

Two exceptions to CTPs on woodland are non-intensive pre-use or secondary use Christmas tree plantations. Since woodland owners are free in their choice of tree with which to reforest their land, Nordmann firs (or other conifers determined for Christmas tree use) are allowed to be planted and harvested as Christmas trees, as long as the majority of the stand (at least 60%) can grow to be high forest (Landesbetrieb Wald & Holz NRW 2019a: 1-2). Additionally, the aforementioned regulations for sustainable CTC need to be obeyed (ibid.: 2-3).

According to § 39 *LFoG*, new establishments of CTPs or BWPs after 11 December 2013 or continuation of conventional intensive CTC after 2028 on woodland require a permit for forest conversion (DE: *Waldumwandlung*). This is only granted if strict conditions are met, such as proof that the conversion is in the greater interest of the common good (DE: "*für das Gemeinwohl von größerem Interesse*", Landtag NRW 2020: 5), as well as compensatory measures for the woodland lost (ibid.; Landtag NRW 2019: 39). CTPs that were established after 2013 without acquiring a permit are illegal and hence prosecuted by the forest authority (Landtag NRW 2020: 5). However, in the first years of the cultivation it is very hard to differentiate between a CTP or regular reforestation (MESSERSCHMIDT 2024: 10). There are indicators like the choice of species (e.g. Nordmann fir), milling or mulching, fertilisation, and herbicide (see *Figure 6*) or pesticide use (MESSERSCHMIDT 2024: 10). Considering the size of the region and the patchwork of regulations and exceptions (see Annex *I.II*), it is difficult to keep track of all CTPs and their legal status.

3 Background of AI technology

The project of developing a framework for detection or classification of CTPs is based on current artificial intelligence (AI) technologies. Therefore, the most important concepts and methods of this field are explained in the following chapters. Initially, the meaning of AI and its historic development will be explored. This includes methods from Machine learning (ML) and deep learning (DL), two essential concepts of the field. The last chapter will provide a close look at different current convolutional neural networks (CNNs), as well as their challenges.

3.1 Definition of AI

The term *artificial intelligence* (AI) was coined for the first time at the Dartmouth conference in 1956 (CHARNIAK 2024: 19). The project proposal established the first definition of artificial intelligence: "the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" (MCCARTHY et al. 1955: 1).

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think ... machines with minds, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI . . . is concerned with intelligent behavior(sic!) in artifacts." (Nilsson, 1998)

Table 1. definitions of artificial intelligence (recreated from RUSSELL & NORVIG 2016: 2)

Since, the definition of AI has been explored by countless researchers of the field. RUSSEL & NORVIG (2016: 2) grouped their results in four categories (see *Table 1*) – which comprise two dimensions: reasoning (top) and behaviour (bottom); as well as two measurements of success: resemblance to human performance (left) and the ideal concept of rationality (right).

The High-Level Expert Group on Artificial Intelligence of the European Commission gave a more recent definition of AI which is in light of contemporary AI use cases (HLEG 2019: 6):

"Artificial intelligence (AI) systems are software (and possibly also hardware) systems designed by humans that, given a complex goal, act in the physical or digital dimension by perceiving their environment through data acquisition, interpreting the collected structured or unstructured data, reasoning on the knowledge, or processing the information, derived from this data and deciding the best action(s) to take to achieve the given goal. AI systems can either use symbolic rules or learn a numeric model, and they can also adapt their behaviour by analysing how the environment is affected by their previous actions.

As a scientific discipline, AI includes several approaches and techniques, such as Machine learning (of which deep learning and reinforcement learning are specific examples), machine reasoning (which includes planning, scheduling, knowledge representation and reasoning, search, and optimization), and robotics (which includes control, perception, sensors, and actuators, as well as the integration of all other techniques into cyber-physical systems)."

3.2 Timeline of AI development

While AI has recently become the major component for groundbreaking innovation, its academic origin dates back as early as the beginnings of Christmas tree cultivation in Germany: the 1950s (CHARNIAK 2024: 19). The first step towards developing AI was made by Alan Turing in 1950, when he explored the possibility for machines to imitate human intelligence (LUGER 2025: 17). He developed the *Turing test* for this purpose, in which a human interrogator uses text-based communication with a machine and a human without knowing which of the two is answering. If the machine succeeds in deceiving the interrogator and they are not able to distinguish between human and the computer, the machine must be considered intelligent, Turing argued (ibid.: 17-19).

The aforementioned Dartmouth Summer Workshop in 1956 marked the beginning of the field, as the topics of the discipline were outlined over the course of two months (LUGER 2025: 24). Furthermore, the conference attendees included Herbert A. Simon, Oliver Selfridge, Allen Newell, Trenchard More, Arthur Samuel, and Ray Solomonoff, who all became AI pioneers, highlighting the huge impact of the workshop for the development of the field (DELIPETREV 2020: 7).

What followed was the first *spring* of AI development. Historically, the field's popularity has grown and declined periodically, which DELIPETREV (2020) labelled *seasons* (see *Figure 12*).

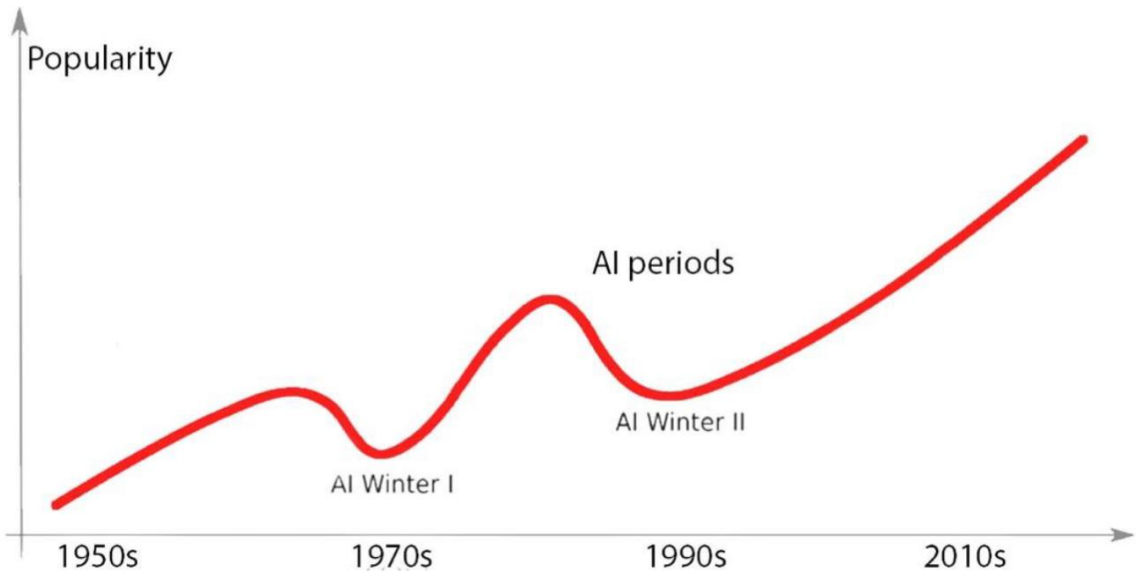


Figure 12. periods of AI development (DELIPETREV 2020: 3)

The first growth phase of the 1950s and 1960s saw the invention of many AI principles. A *winter* period ensued in the 1970s, caused by issues that arose during the initial spring. However, in the late 1970s another boom phase commenced with the shift to symbolic algorithms, which was again interrupted by a downturn in the 1990s. With the dawn of Machine learning (ML) in the 1990s, there has been an ongoing spring, which DELIPETREV (2020: 3) states could culminate in a *summer* or yet another AI winter.

3.2.1 The first spring (1950s to 1970s)

The early years of AI development were all about pushing the boundaries of what machines were capable of. The achievements are mundane for current computers, but the hardware restrictions of the 1950s and 1960s make them all the more impressive, as computers were mostly regarded as mere calculators capable of basic arithmetic (RUSSELL & NORVIG 2016: 18). Furthermore, the access to these early computers was extremely limited, as there were only a few units in the world, and a multitude of researchers wanted to test their hypothesis with them, which, for example, resulted in some testing having to occur at night (ibid.: 19).

Possibly the first AI software to be ever written was a checkers-playing program, designed by Arthur Samuel of IBM in 1952 (CHARNIAK 2024: 20). It was able to compete with amateur human players, proving that machines can learn (DELIPETREV 2020: 7). In 1958, John McCarthy, the author of the proposal for the Dartmouth conference, and his student Steve Russel, created the programming language *Lisp* (List Processing), which was optimised for symbolic processing, rather than numeric processing, as other programming languages of the time were (CHARNIAK 2024: 139-140). It became the main language in AI programs for the next 30 years (RUSSELL & NORVIG 2016: 19). Furthermore, McCarthy developed a hypothetical program called the *Advice Taker* in the same year, which is regarded today as the first complete AI system (ibid.). Provided with knowledge, it was supposed to search for solutions to problems from the memorised information. More remarkably, it was designed to accept new axioms during operation, thus taking in information from new areas without being reprogrammed. This deductive process is still relevant for programs today (ibid.).

In 1958, Frank Rosenblatt published a paper about a hypothetical nervous system, the *perceptron*, a discovery he made the previous year (ROSENBLATT 1958). Based on neurons in the human brain, perceptrons are binary classifiers with connections of multiple inputs and a single output, whereby every input is associated to a weight (see w_1, w_2, w_3, w_n in *Figure 13*) (SEJNOWSKI 2018: 39-42). The learning algorithm developed by Rosenblatt could be trained on a set of images to identify a digit as long as it was provided with their labels, which set the weights to recognise the correct class. The computer would learn to recognise the digit based on pixel values (0 to 255), which make it up in an image. By being trained on the labelled images, the weights would be above zero 0 (see $\sum$ in *Figure 13*) for pixels that are aligned with displaying the correct digit, and below zero if they display an incorrect

digit. If the final value is positive, the image is classified as the correct digit; vice versa with a negative value (CHARNIAK 2024: 39-40). Perceptrons were an enormous innovation presenting the idea of connectionism, and effectively **the first (single-layer) neural network** (NN). Thus, they were the foundation for deep learning (DL) and, specifically, NNs (DELIPETREV 2020: 7). However, the model was also limited to very specific use cases. Marvin Minsky and Seymour Papert exposed this in their publication *Perpectrons* in 1969, which ultimately led to perceptron research being shut down (CHARNIAK 2024: 40).

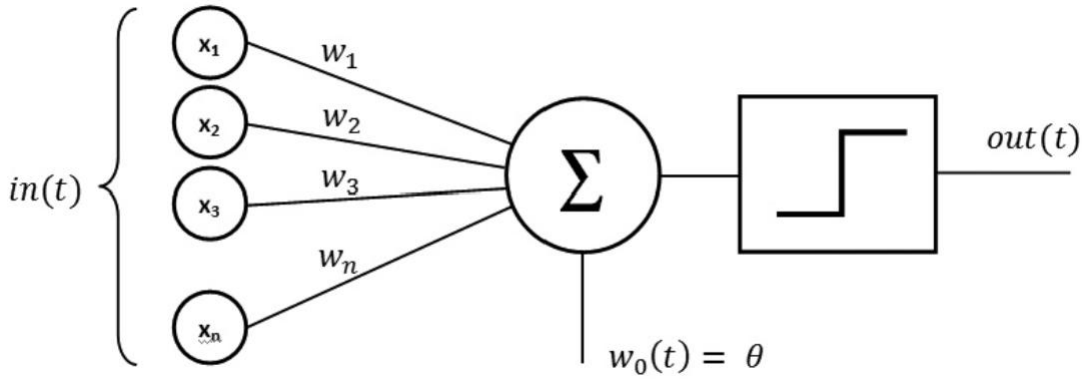


Figure 13. a perceptron: inputs with weights on the left , single output on the right (SEJNOWSKI 2018: 42)

While many of the initial milestones in the field were theoretical designs, there were also fast advances with practical programs. Hence, Herbert Gelernter from IBM created the *Geometry Theorem Prover* in 1959. This early AI program was proficient in solving advanced mathematical theorems (RUSSELL & NORVIG 2016: 19). A complete theorem-proving algorithm, called the resolution method, was constructed by J. A. Robinson at Stanford in 1965 under McCarthy. It implemented a complete first-order logic, without propositional reasoning to solve problems (ibid.: 19, 276). The quick advances were not only restricted to Mathematics, though. While natural-Language Processing (NLP), which enabled programs to handle human languages – e.g. translating from one language to another – was initially not successful in the early years of AI, a first milestone was achieved in 1964: STUDENT solved text-based mathematical problems (CHARNIAK 2024: 29). The following year, the NLP program ELIZA was published. It was able to respond to questions imitating a psychotherapist, which initially fooled some users to think they are talking to a real doctor. However, it did have limitations and the conversations did not conclude sensibly (DELIPETREV 2020: 7). A

major breakthrough in NLP programming was SHRDLU by Terry Winograd from MIT in 1968. It offered a simulated environment called *blocks world,* with which the user could interact using text commands (CHARNIAK 2024: 29-31). The program came with a visual simulation that was displayed on a cathode-ray tube, a contemporary computer monitor. A striking feature of SHRDLU was the incorporation of syntax, semantics, and pragmatics, which enabled the translation of English-language commands into logical instructions. Furthermore, the program included physics rules, thus, blocks in the simulation could only be moved if there were no objects on top of them (ibid.).

With numerous breakthroughs reached in just a matter of years, experts at the time were full of excitement and bold predictions. In 1970, Marvin Minsky famously stated: "In three to eight years, we will get a machine with average human intelligence" (DELIPETREV 2020: 8).

3.2.2 The first winter (1970s)

Following the craze of the initial AI boom came a downturn marked by a remarkable decline in projects by both academia and industry. After the US government spent millions of USD on research in the field, which led to academic breakthroughs, the success did not translate into solving large-scale real-world problems (DELIPETREV 2020: 8). Thus, the return on investment was not delivered, which lead to great disappointment in government agencies and some scientific communities. This was underlined by two reports, the ALPAC report (1966) in the United States and the Lighthill report (1973) in the United Kingdom (ibid.).

The Automatic Language Processing Advisory Committee (ALPAC) was established in the United States to report on the development of machine translation. The focus at the time was an automation of translations of Russian to English, as part of American intelligence efforts in the Cold War. Deemed promising and important, massive funds were spent on projects in the field, which had academic success, but ALPAC was not satisfied with the return on investment. There was no improvement in speed, cost, or quality compared to human translators, which made early machine translation ultimately a failure. (ibid.)

James Lighthill of the University of Cambridge wrote *Artificial Intelligence: A General Survey*, published by the British Science Research Council in 1973 (DELIPETREV 2020: 8). The report highlighted how AI had been falling short of expectations, which reflected both public

and scientific opinion (CHARNIAK 2024: 187). A major issue that held back its capabilities was the so-called "[c]ombinatorial explosion" (DELIPETREV 2020: 8). This term describes that exponential growth in problem complexity results in exponential processing time, which was entirely unfeasible to deliver with the computational limitations at the time (ibid.). In comparison, other solutions that required conventional engineering were much more promising. Thus, Lighthill recommend the termination of government funding, and British AI research fell into hibernation (CHARNIAK 2024: 187).

3.2.3 The second spring (1970s to 1990s)

Although the issues of the first AI winter caused a dramatic decrease in funding, not all research seized in the 1970s altogether. Notable milestones were still reached, and a new spring came along, with innovation accelerating in the 1980s (DELIPETREV 2020: 9). This was influenced by the rise of Personal Computers (PCs), which made computational resources more easily available to researchers and private corporations (ibid.: 10).

In 1972, Alain Colmerauer and Philippe Roussel developed and implemented *Prolog* in Marseille (DELIPETREV 2020: 9). Prolog (Programming in Logic) is a symbolic programming language based on logical rules, which became the main language of the period alongside Lisp, which was still popular in the USA (ALPAYDIN 2016: 49).

After research on connectionism (perceptron) came to a halt at the dawn of the first AI winter, the field was revived in the 1980s. **Hopfield net was a recurrent NN** that worked in a similar way to the connections in the hippocampus of the human brain, combining the fields of physics, neuroscience, and AI (SEJNOWSKI 2018: 94). It was the first nonlinear network model that "was guaranteed to converge to a stable state, called an 'attractor'" (ibid.: 93). In addition, these attractors could be programmed to become *content-addressable memory*, with which information could be accessed by beginning with a piece of memory and the network filling in the gaps – which is comparable to how our brain recalls information (ibid.).

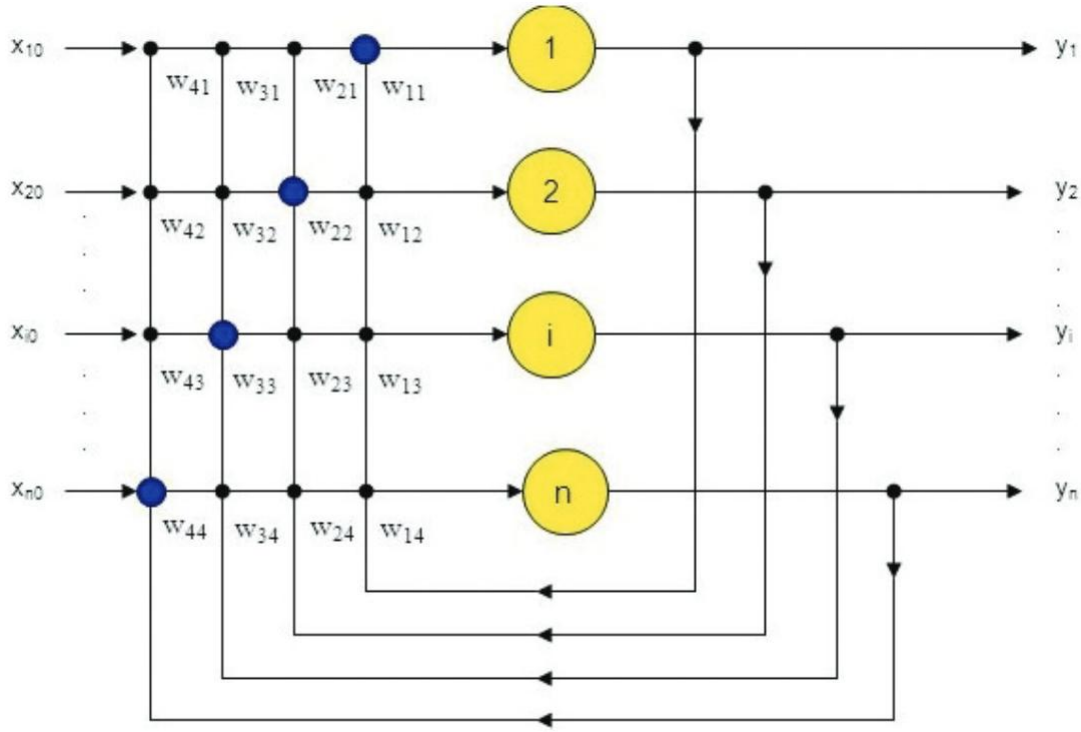


Figure 14. schematic drawing of a Hopfield net (SEJNOWSKI 2018: 95)

Figure 14 shows a schematic drawing of a Hopfield net. In this network, all units are connected, with inputs on the left (x_i), symmetric weights ($w_{ij} = w_{ji}$), and outputs on the right (y_j). One unit is calculated at a time by adding inputs and comparing them to a threshold. If the threshold is exceeded, the output is 1, otherwise it is 0. The Hopfield net has an energy function, which never increases with any update in the network: $E = \Sigma\, w_{ij}\, x_i\, x_j$
Eventually, the system will stabilise at the *attractor* state, which is the local minimum energy. This state can also be recovered with a part of its information exhibiting content-addressable memory. The weights of these stored vectors are learned by the Hebbian Synaptic Plasticity $\Delta w_{ij} = \alpha\, x_i\, x_j$, in which Δw_{ij} is the change of the weight, α is the learning rate, and x are the stored vectors. (SEJNOWSKI 2018: 95)

The Hebbian Synaptic Plasticity is a principle originating from neuroscience and describes learning functions in the brain. It refers to the connection (synapse) between neurons becoming stronger if they are activated together, which physically alters the structure of the brain (SEJNOWSKI 2018: 101). Thus, if one practices a skill, the neurons involved will become stronger, which makes the ability or memory more permanent.

Another milestone in the NN field was the **Boltzmann Machine** developed by Geoffrey Hinton and Terrence Sejnowski in 1985. Like the Hopfield net, it used symmetric connections between binary units, but it also added statistical mechanics principles – the *simulated annealing*. The units update stochastically, hereby the probability of activation is dependent on the energy change of a temperature parameter. The system starts at a high temperature, which allows the exploration of random states. Gradually, the temperature is lowered until finally arriving at the global energy minimum. The Boltzmann Machine also introduced hidden units between the input and output layers, which enabled unsupervised learning and laid the groundwork for multilayer NNs. (SEJNOWSKI 2018: 96-101)

In 1989, towards the end of the second spring, a huge leap also took place in the field of computer vision, as **Yann LeCun** from New York University **published the first modern CNN** (CHARNIAK 2024: 131). The *LeNet* neural network introduced backpropagation to DL, in order to solve tasks of image recognition on greyscale images of handwritten digits (ibid.: 131-132). This structure was founded on the concept of the first convolutional network called the *neocognitron* by Kunihiko Fukushima in 1980 (ALPAYDIN 2016: 85). Fukushima based his NN on successive layers imitating the make-up of a cat's visual cortex, which had been analysed in 1962 by neurobiologists David Hubel and Torsten Wiesel (ibid.: 84-85). They observed a pattern of firing neurons that responded to lines appearing on a screen, with sets of neurons being focused at different line orientations (e.g. 90° or 95°) (CHARNIAK 2024: 86).

5	**0**	**0**	0	0
5	**5**	**0**	0	0
5	**5**	**5**	0	0
5	5	5	5	0
5	5	5	5	5

Table 2. diagonal change of light intensity with one receptive field highlighted (recreated and modified table from CHARNIAK 2024: 87)

Table 2 displays pixels representing light intensity of a diagonal line running from the top left to the bottom right. For simplicity, this light intensity display is in greyscale rather than red, green, blue (RGB). Based on the cat's visual cortex response, in convolutional

processing the image is handled in patches, also referred to as receptive fields (e.g. of 3 × 3) (ibid.: 87).

-1	-1	-1		5	0	0
1	1	-1	×	5	5	0
1	1	1		5	5	5

Table 3. convolution: kernel on the left, receptive field on the right (recreated and modified table from CHARNIAK 2024: 87)

In this example, the top left patch of the image (see bold digits in *Table 2*) is processed first. For this purpose, it is multiplied by a kernel that is made up of -1s and 1s (see *Table 3*). Afterwards, all results are added up, which equals a sum of -5 for the first row, 10 for the second row, 15 for the third row and the total sum of all rows of 20. However, if we had a receptive field that did not display a diagonal, but instead no greyscale change (all fields equal 5 or 0), the sum of the convolution would be 10 and 0 respectively. Thus, the kernel on the left functions as a detector for an intensity gradient, with a larger multiplication sum indicating a likeness to a diagonal line running from left to right (ibid.: 88).

The other principle that LeNet was founded upon is **backpropagation of errors** (or *back-prop* for short), which was popularised in 1986 by David Rumelhart, Geoffrey Hinton, and Ronald Williams at the University of California (SEJNOWSKI 2018: 109). Backprop enables the calculation of how much weights contribute to an overall error within a neural network. The process is started at the output, from which point every layer gradient is backtracked to calculate the previous layer of weights – all the way back to the input. Thus, the network can be trained to strengthen or weaken connections (ibid.: 112).

LeCun combined both convolution and backprop in the LeNet architecture and added pooling to the structure (see *Figure 15*). The input of the network is a 28 × 28 image and the output on the right is a 10 × 1 array. To get there, LeNet starts off with convolution – assigning kernels to receptive fields. In the exemplary architecture in *Figure 15* six different convolutional kernels – for different orientations – are represented by the six squares in the second step. This convolution already includes the aforementioned backpropagation to improve the prediction to this point. The following layer is pooling, where the size is further

reduced (as the input is 28 × 28 = 784 and the output is 10). Hereby, the average information – lines or texture – of 4 × 4 receptive fields from the input are merged (*pooled*) together. Thus, the original 28 × 28 is transformed to six 14 × 14 arrays, which are focused on different features picked by the kernels of the previous layer. This is followed by a second convolution, where sixteen kernels of 10 × 10 are applied to look for patterns in lines. The result is pooled again in sixteen 5 × 5 arrays. At this point, the information that was previously a numeral for light intensity, in the input image, has been transformed into a vector, which now describes the appearance of the image. The last layers are connected and condensed, with the maximum value of the output indicating the system's prediction of which digit is displayed in the initial image. (CHARNIAK 2024: 131-132)

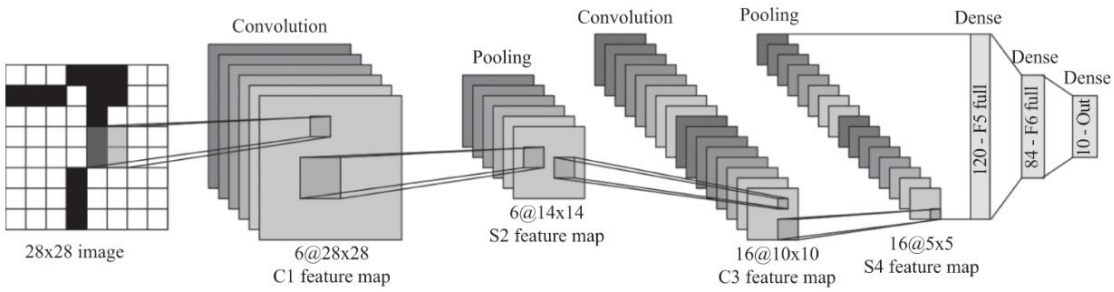


Figure 15. LeNet neural network architecture (CHARNIAK 2024: 131)

Effectively, LeNet was already a full-functioning application of NN on computer vision tasks. However, as with prior innovations, like the perceptron, this milestone received little notice at the time (ibid.: 133). Instead, the field was dominated by an application for real-world problem solving: the expert system. The idea was to contain vast *expert* level knowledge of a domain in a software that can be questioned by the user, drawing logical conclusions from the knowledge (DELIPETREV 2020: 9). They were sold as software packages to the consumer market to be used on PCs, or were implemented by large corporations to streamline their administration, production, or services (ibid.). Expert systems include two parts: the knowledge base is a database that stores information, rules, and relationships about a certain domain, e.g. medical diagnostics; and the inference engine is the computational component, that applies logical rules to deduce new information using the knowledge base (ALPAYDIN 2016: 49).

One of the first successful expert systems was MYCIN, which was developed by Edward Shortcliffe and Bruch Buchanan at Stanford in 1975 (CHARNIAK 2024: 52). It specialised in

the diagnosis of bacterial infections and their treatment (ibid.). The conclusions drawn by the program were not solely based on the dichotomy of true or false, but introduced a degree of probability with a certainty factors model (RUSSELL & NORVIG 2016: 548). Another noteworthy expert system was developed in 1980 by the Digital Equipment Corporation (DEC) (CHARNIAK 2024: 50). R1, which was later renamed to XCON, was one of the most viable programs of its type and among the few that were used regularly on the problems they were designed to solve (ibid.). DEC developed R1 to support the orders of mainframe computer systems, which usually contained 90 individual components that needed to be purchased separately. As a result of the complexity and large number of items, there was a recurring issue with mainframe orders that did not include all required components (ibid.). R1 had approximately 2,500 *if-then-rules*, which were designed by the programmer alongside domain experts (ibid.). This type of expert system is referred to as a *production system* and would iterate its database of rules over a database of facts about the order, and then add information based on rules that are fulfilled (ibid.: 51). These conditional rules stated e.g., if the order contains component *x* it must include associated component *y*, and if this is not the case, it is added to the order. R1 proved to be highly profitable for DEC as it saved the corporation $40 million a year between 1980 and 1986 (DELIPETREV 2020: 9).With this success, it does not come as a surprise that by 1988 the AI department of DEC had a total of 40 expert systems deployed with one still being in development (RUSSELL & NORVIG 2016: 24). This rapid implementation of AI into corporate structures was not limited to companies associated with IT, as almost every major US corporation had an AI department developing expert systems. For instance, the chemical corporation DuPont was saving $10 million a year with 100 expert systems in use, and they were developing 500 more (ibid.). The total US investments in expert systems exceeded $1 billion by 1985 (DELIPETREV 2020: 9).

AI applications also took off in other parts of the world, quickly sparking competition between countries and resulting in even larger investments, which fostered collaboration between academia and industries. Japan started the fifth-generation computer project, which received $850 million in funding (RUSSELL & NORVIG 2016: 24). The aim was to develop computers running on Prolog that were able to complete tasks like translation, image interpretation, or communication with humans (DELIPETREV 2020: 9). The project also included programs on expert systems (ALPAYDIN 2016: 49). The European Union launched the European Strategic Program on Research in Information Technology (ESPRIT) in 1984, which

pursued research and development projects and industrial technology transfers, as well as microelectronics, with a budget of roughly €1.5 billion between 1984 and 1988 (Commission of the European Communities 1988: 2). In addition, the UK had its own AI programme rolled out, the Alvey Programme, which reinstated funding that had been cut in the aftermath of the Lighthill report (see chapter *3.2.2*) (RUSSELL & NORVIG 2016: 24).

3.2.4 The second winter (1990s)

After the tremendous popularity of expert systems, and huge investments in aspiring AI projects across a multitude of nations in the 1980s, another winter followed. Similarly to the first slowdown in the 1970s, this was partly due to overly confident expectations that were hard to meet (RUSSELL & NORVIG 2016: 24). The initial success of expert systems saving corporations millions of dollars prompted only a brief hype, which came to an inevitable halt, caused by two major flaws: laborious knowledge acquisition and unsuitable logic for real world representation (ALPAYDIN 2016: 49).

The development of the knowledge base was a manual process that did not only require expert knowledge of a domain, but also expertise on the use of the AI software. The system itself was not able to learn from the information, an experienced operator needed to make the correct inputs, setting logical rules between pieces of information to ensure correct deductions from the knowledge base are possible for the user (ibid.). However, even after this arduous development process, the expert system was bound to the limitations of said logic, which made them applicable for some domains, but many real world problems were simply too complex for this structure. While expert systems were able to represent some uncertainty in their deductions, large sets of rules resulted in unwanted interactions, which meant that uncertainty factors of old rules required adjustments when new rules were introduced (RUSSELL & NORVIG 2016: 549). Thus, expert systems also had high-maintenance constraints.

Furthermore, the symbolic programming languages Lisp and Prolog and their development environments could not keep up with compiled language efficiency (e.g. C), which meant the downfall of the Lisp machines – the predominant environment used in the US – by the late 1980s (DELIPETREV 2020: 10).

3.2.5 The third spring (1990s to now)

Although the field was deep in its second dry spell, with investments discontinued and corporation departments made redundant, progress did not seize altogether. In order to still receive funding, researchers chose different names for their projects such as cognitive systems or optimisation algorithms instead of AI (DELIPETREV 2020: 11). Ever since the last winter, technologic innovation in the field has rapidly increased with no slowdown in sight. This was led by two main fields: machine learning (ML) and its subset deep learning (DL) (see *Figure 16*). The main difference between the two is the depth of the system architecture: the structure of conventional MLs such as SVMs (see chapter *3.2.5.1*) is usually shallow, whereas DL methods such as LeNet (see *Figure 15*) use *deep*, complex structures. Furthermore, DL algorithms thrive on large complex datasets (see chapter *3.2.5.2*), while ML techniques work well on small structured data.

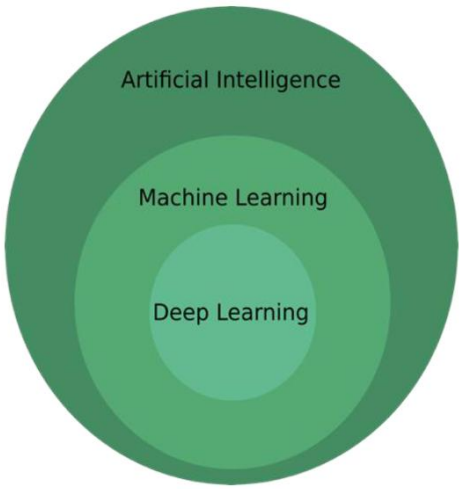


Figure 16. relation between AI, ML, and DL (DELIPETREV 2020: 12)

While basic ML algorithms were of great importance in many fields until the 2010s and are still used for some tasks, eventually DL algorithms became the gold standard and are nowadays at the forefront of AI innovation. The difference in structures of ML and DL techniques, as well as the milestones that paved the way, will be further explored in the following chapters.

3.2.5.1 The dawn of machine learning and deep learning (1990s to 2000s)

During the slowdown of the early 1990s, many academics continued to work on AI related research under a different name. One of the first breakthroughs of the decade was TD-Gammon, developed by Gerald Tesauro from IBM in 1992 (CHARNIAK 2024: 161). The ML program used a type of reinforcement learning (RL) algorithm called temporal difference learning (TD) to teach itself backgammon. After only a few years, the program was proficient enough to beat professional players (ibid.). RL algorithms are a type of unsupervised learning technique. They are slower than supervised algorithms, but they offer greater potential as there is no bias by the supervisor (ALPAYDIN 2016: 107). TD-Gammon learned by playing against a copy of itself, as Tesauro only programmed the general rules of backgammon, but not knowledge of what constitutes a good move (SEJNOWSKI 2018: 146). After 1,500,000 games of self-training, TD-Gammon was able make new moves that were superior to professionals, which marked a turning point for the AI field (ALPAYDIN 2016: 107).

However, the reception of TD-Gammon was in no comparison as to IBM's Deep Blue, a chess program that beat Gary Kasparov in 1997, the world champion at the time (SEJNOWSKI 2018: 149). While it became an important milestone for computing power, the publicity does not reflect the importance to the AI field. Deep Blue did not teach itself chess, but instead was fed with information about chess strategy and then won by brute force, using state-of-the-art processors to predict further moves than any human could (RUSSELL & NORVIG 2016: 185). Thus, it did not showcase the contemporary advances in ML or DL (DELIPETREV 2020: 13).

Another important milestone in ML in the 1990s was Vladimir Vapnik's **Support Vector Machine** (SVM), which was published in 1995. SVMs were used both for regression and classification tasks including image recognition (DELIPETREV 2020: 13). They still are part of the supervised ML toolkit as of today, especially relevant for off-the-shelf use (RUSSELL & NORVIG 2016: 744). SVMs can deal with nonlinear data by using the kernel trick. This embeds data into a higher-dimensional space, which makes it linearly separable. The aim of this method is the definition of a maximum margin separator – a line or hyperplane in higher dimensions – which has the largest possible margin between itself and the closest data points from the classes (ibid.). These data points are the support vectors (SV), which are the defining points for the separator line or hyperplane. This means that not all training data is equally

important to the model, only the SVs cause a shift of the boundary, other data points will not influence it even if they are removed. This method is nonparametric, which means that the training samples are retained, but this only includes a small subset of data that makes up the SVs. Thus, SVMs are both flexible and resistant to overfitting (also see chapter *3.3.2.1*) (ibid.).

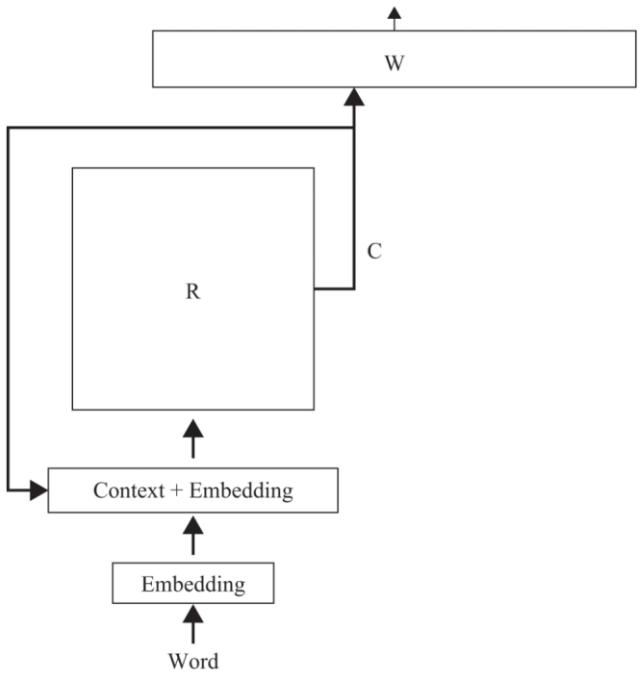


Figure 17. language model based on recurrent NN (CHARNIAK 2024: 146)

In 1997, a big breakthrough in DL was made by Sepp Hochreiter and Jürgen Schmidhuber when they introduced **long short-term memory (LSTM) to recurrent NNs** (also see Hopfield net, *Figure 14*) (CHARNIAK 2024: 147). They used this new technique in language models (LMs), which process text one word at a time (ibid.). *Figure 17* shows the architecture of this model, where an additional context (C) is derived from the output of the perceptrons in step R. The final output is generated in this structure by 50,000 perceptrons in W, using both the context and the embedded words from the processed text (ibid.). Up to the introduction of LSTM, the context C would be rebuilt after every iteration. Hochreiter and Schmidhuber instead decided to keep passing the context C from iteration to iteration, except if the new word or output R removes or adds parts to it (ibid.). This ingenious technique resulted in a tremendous improvement of LM capabilities and was used in NNs for decades (ibid.).

BREIMAN (2001) established a ML method in 2001 in his journal article on Random Forests (RF), which is still used today for regression and classification tasks, as well as image recognition. This method is based on a previously established ML concept called Decision trees. It is a partitioning model of *decisions*, where an entire training dataset is split into subsets along branches to narrow them down to a more homogenous group to reach the target. This continues until a stopping condition is met, e.g. all data belongs to one class or a minimum node size is reached (HO 1995: 278-282). BREIMAN (2001: 5-7) expanded this concept by creating an ensemble of decision tress that are grown using a random vector with bootstrap sampling, which creates multiple new datasets from the original data by randomly drawing samples with replacement. While the bootstrap sample is the same size as the original dataset, it contains duplicate datapoints, which helps to train the different trees on slightly different datasets, thus preventing overfitting and improving stability. Finally, the majority class vote (classification) or the average prediction (regression) of the ensemble of decision trees is aggregated for the output of the RF model (ibid.: 8).

The 2000s brought with it a paradigm shift in AI. In 2006, Fei-Fei Li from Stanford University theorised that the main barrier for AI models was the limited data quantity, rather than the architecture of models (DELIPETREV 2020: 11). As the new millennium saw a rapid acceleration in hardware development, the scalability of AI techniques, especially DL algorithms could be proven. Larger datasets would produce much better results and thus, techniques such as CNNs that were developed in the 1990s or even earlier only showed their level of performance when better processing units became available (SEJNOWSKI 2018: 137-138).

After having worked on it since 2006, **Li published the ImageNet in 2009**, which was a revolutionary milestone for computer vision (DELIPETREV 2020: 11). Instead of working on better algorithms like most of her colleagues, she was looking to prove her theory of higher data quantity leading to improved model results by developing a large image training database (ibid.). The first version in 2009 contained 3.2 million labelled images classified in more than 5,200 categories divided into 12 subsets (like “vehicle” or “mammal”) (ibid.). ImageNet was also extended over the years and by 2012, it contained 15 million labelled images in over 22,000 categories (SEJNOWSKI 2018: 128-129). This extensive dataset provided an opportunity for researchers all over the world to train and test their algorithms. At the same time, it also worked as a benchmark test to compare state-of-the-art techniques and

promote the rapidly improving capabilities of DL algorithms (RUSSAKOVSKY et al. 2015: 211).

3.2.5.2 The deep learning revolution (2010s to now)

The founders of ImageNet organised a yearly contest called ImageNet Large Scale Visual Recognition Challenge (ILSVRC), in which teams of researchers would compete for the lowest error rate in image recognition involving two tasks: image-level annotation (classification of a whole image, e.g. image includes/does not include cars) and object-level annotation (object detection with bounding boxes around objects within the image) (RUSSAKOVSKY et al. 2015: 211). The classification task was part of the contest from the beginning in 2010 (ibid.: 213). Object localisation was added in 2011, however, only with detection of a single instance per object category (e.g. a bounding box around one car in the image even if more cars are present) and in 2013, object detection of every instance was included in the competition (ibid.). This quick alteration of the rules is a remarkable representation of the swift improvement of AI algorithm performance in image recognition tasks during the 2010s. In the first two years of the ILSVRC, classic ML methods such as SVM still dominated the contest, but since 2012, DL networks such as CNNs almost exclusively won across all disciplines (ibid.: 233).

The turning point of the contest in 2012 was accomplished by the SuperVision team comprised of Geoffrey Hinton and Alex Krizhevsky, when their **deep CNN AlexNet** achieved a major error rate drop from 25 percent in the previous year to 16 percent (see *Figure 18*) (CHARNIAK 2024: 135). AlexNet has a similar architecture to LeNet (see *Figure 15*), but instead of two convolutional layers, AlexNet was equipped with six, and instead of two pooling layers there were three. Just like LeNet, the network was concluded by three dense layers (ibid.).

After the striking success of AlexNet, the error rate continued to decline over the following years, with 2014 as the first time in the competition that the winner accomplished a lower error rate than a human. However, this comparison to human intelligence must be taken with a pinch of salt, since 1,000 categories must be remembered for the image classification or

object detection, which is an easy task for a computer. Nevertheless, even matching human capabilities in image recognition is a remarkable feat. (ibid.)

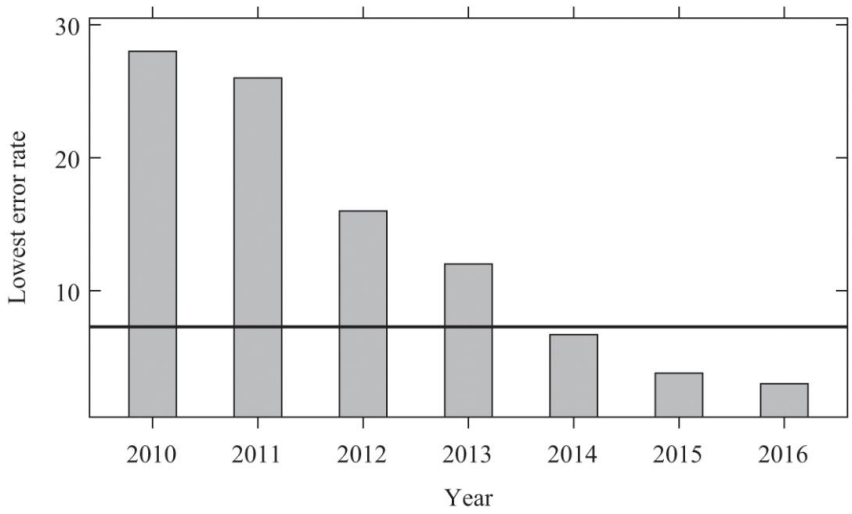


Figure 18. error rate of winners in the ILSVRC; line indicates human error rate (CHARNIAK 2024: 135)

The winner of ILSVRC 2015 was **ResNet** by the Microsoft Research team, under the leadership of Kaiming He, which caused a rise of very deep NNs as they introduced ***deep residual learning***. HE et al. (2015) presented this concept as a solution to the degradation in accuracy and lack of optimisation in deep NNs with a high number of stacked layers. The team observed that an increase in network depth leads first to a saturation in accuracy and then a rapid decrease (HE et al. 2015: 1). Overfitting as a possible cause was ruled out (also see chapter *3.3.2.1*). The standard approach of a deep network before the new framework by HE et al. was each layer trying to map a new function, H(x), to reach a desired output. Every layer would use the output of the previous layer as input and learn from it (ibid.: 2). The new approach sought to reduce complexity by making the layers learn the difference – the residual – between the input and the desired output: $F(x) = H(x) - x$ (ibid.). That means the output of a building block in residual working becomes $F(x) + x$. This new method is enabled through shortcut identity connections (see *Figure 19*). This adds the input (x) of a layer directly to the output without requiring extra parameters or increasing computational complexity (ibid.: 2). After each layer is a rectified linear unit (*relu*), which replaces outputs with a negative value with zero (CHARNIAK 2024: 131).

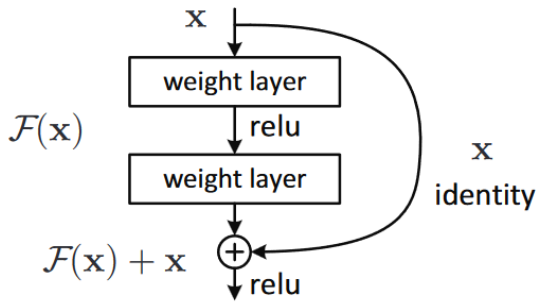


Figure 19. building block of residual learning (HE et al. 2015: 2)

With residual learning HE et al. (2015: 6-7) were able to demonstrate deep NNs that still improved their accuracy after a high number of layers, reaching better results than NNs with fewer layers and the traditional learning approach.

Apart from the remarkable recent developments in computer vision, which will be further explored in chapter *3.3*, the 2010s also saw the emergence of transformer architecture, which paved the way for today's Large Language models (LLMs). *Attention Is All You Need* was a NN model for machine translation by a Google research team (VASWANI et al. 2017). However, their transformer model was quickly adopted by OpenAI, which used the concept on language models (LMs) (CHARNIAK 2024: 161).

Figure 20 shows a simplified structure of a **transformer language model**. The four boxes at the bottom signify the word input, which in this case can be 1,000 words at once and more. This is a huge step from recurrent NNs, in which systems iterate word for word with each providing context for the next (ibid.: 162). Furthermore, the transformer model adds context-sensitive embeddings of up to e.g. 1,000 surrounding words to the word that is being processed, while paying attention to words that are more important for the context. This last step is done in the next layer after the input titled *attention*, where the context is added to the words from left to right, meaning that the transformer model uses the input and derived context to predict the next word within a sentence (ibid.: 163).

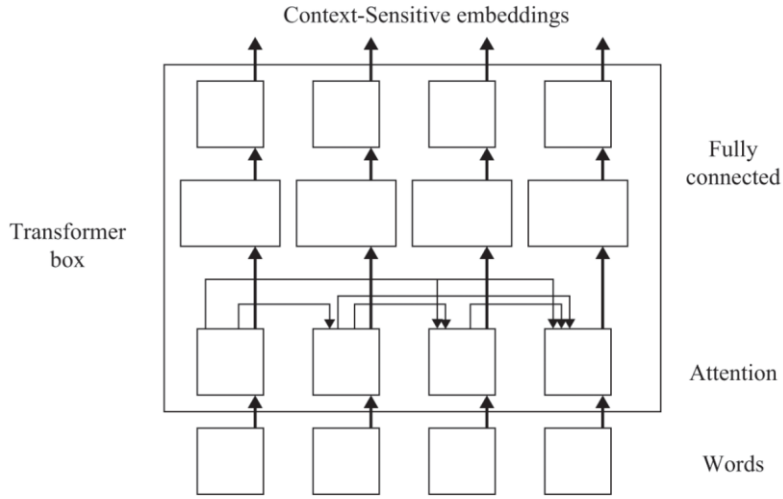


Figure 20. simplified structure of transformer model (CHARNIAK 2024: 162)

The next two layers are fully connected, and in the case of actual LLMs billions of parameters expand the dimensionality of embeddings (ibid.). Hereby, the words are not treated as individual units, but split into word pieces, which reduces the size of the necessary vocabulary for comprehension and prediction (ibid.: 164-165). This process is called word piece tokenisation and permits LLMs to handle vast vocabularies and rare words. At the end, the output is an efficient, context-aware prediction about the next token, or more simply put, the next word, and this process enables *intelligent* responses by LLMs like Chat-GPT (ibid.: 165-166).

3.3 Contemporary image recognition methods

The findings of many years of AI development have contributed to an array of different image recognition algorithms. Following the major achievements in deep neural networks in the 2010s, they have now become the gold standard for recognition tasks. Since the range of applications in the sector has grown immensely, the requirements for algorithms have become more specific as well. Different models are suited to different tasks – there is no single most accurate model. Nevertheless, a few very effective models have been dominating the field in recent years, two of which will be presented in this chapter. The challenges of image recognition will also be explored.

3.3.1 Examples of state-of-the-art algorithms

A comparative review on object detection algorithms by ZHAO et al. (2024) focuses on differentiating two-stage and one-stage architectures when comparing state-of-the-art CNNs. **Two-stage detectors** such as R-CNN (Regional CNN) models (Fast R-CNN, Faster R-CNN, or Mask R-CNN) first generate region proposals, i.e. where sought objects might be located, and then classify these regions (ibid.: 27-28). **One-stage algorithms**, such as YOLO, process the entire image in one pass without initial region proposal, hence the name *You Only Look Once* (ibid.: 30). Both approaches have their advantages and disadvantages and will be explored in the following chapters.

3.3.1.1 Mask R-CNN

Introduced in 2017 by HE et al. (2020), Mask R-CNN is the latest of the R-CNN based algorithms. Additionally, it is also the most accurate (ZHAO et al. 2024: 29), which is why it is exemplary for two-stage detectors. Mask R-CNN introduced *instance segmentation* to the Regional CNNs – **combining semantic segmentation with object detection** (HE et al. 2020: 386). HE's et al. algorithm includes the classification of "each pixel into a fixed set of categories without differentiating object instances" (ibid.). This is achieved by adding a step in parallel to the object detection after the Regions of Interest (RoI) have been assigned (see *Figure 22*).

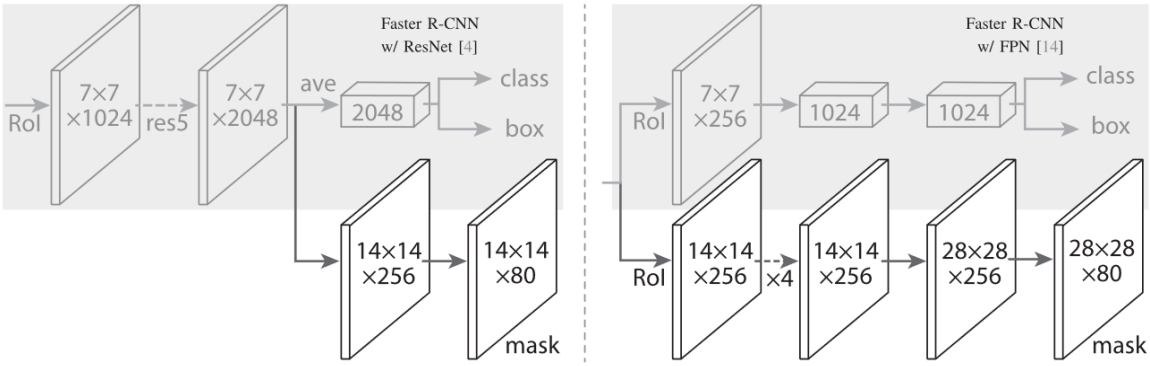


Figure 21. Mask R-CNN architecture with ResNet (left) or FPN backbone (right) (HE ET AL. 2020: 389)

This mask branch is applied to every RoI and was built on the already robust Faster R-CNN framework, however HE et al. modified the RoI creation with *RoIAlign*, which retains the integrity of spatial locations of objects and improves the mask accuracy. This step follows the initial feature mapping by the backbone of the framework, which is ResNet-50 (see chapter *3.2.5.2*) or the Feature Pyramid Network (FNP), respectively (see left and right on *Figure 21*).

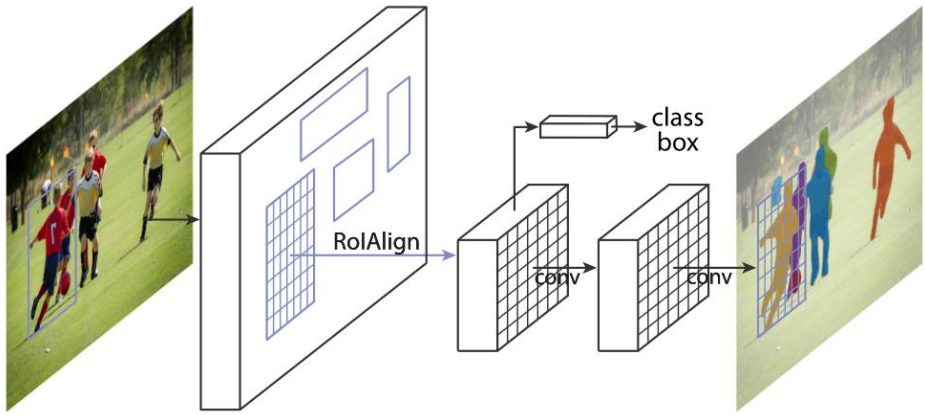


Figure 22. mask branch of Mask R-CNN for instance segmentation (HE ET AL. 2020: 387)

In the **ResNet version**, the RoI is followed by the res5 step, where the RoI 7 × 7 × 1024 dimension tensor (height × width × colour channels) is transformed into 7 × 7 × 2048. Afterwards, the global average pooling (*ave*) computes the mean for each channel (HE et al. 2020: 389). This results in 2048 values (one per channel), of which the class and bounding box are predicted. In parallel, the mask branch runs decoupled from class and box prediction (see *Figure 23*). The former is only a binary prediction for every class, whereas the classification branch performs the actual prediction of the category (ibid.: 387). The mask branch is fully convolutional, meaning that there is no flattening of the spatial dimension – this

remains 14 × 14, whereas the channel depth changes. This preserves the aforementioned alignment, enabling more precise segmentation predictions (ibid.: 389).

The Mask R-CNN version that uses Faster R-CNN with **FPN as a backbone**, has a slightly different architecture (see right side on *Figure 21*). The FPN was developed by Lin et al. in 2017 to enhance the performance of object detection systems in scenarios with high scale variance (SUN et al. 2024: 6). The network is structured like a pyramid with a bottom-up, a top-down, and lateral pathways. The bottom-up path follows the normal convolution, whereas the top-down path upsamples high-level (semantic) features and merges them with low-level features through the lateral connections. Low-level features in this context are general visual patterns like edges, corners, or textures, which have usually a high resolution. High-level – also referred to as semantic features – are object patterns like a wheel or an eye, which have a low level of detail compared to the low-level features (ibid.).

The RoI are directly assigned to each pyramid level of the FPN (HE et al. 2020: 389). The backbone in this Mask R-CNN architecture already contains res5 and creates a 7 × 7 × 256 tensor for every RoI. Because of the compactness of these tensors (7 × 7 × 256 vs. 7 × 7 × 1024 with the ResNet backbone), the head (i.e. where the detection takes place) is *deeper* with four stacked 14 × 14 × 256 convolutional layers followed by an upsampling to 28 × 28 × 256 and finally the mask (ibid.). This process requires overall fewer filters than with the ResNet backbone and is thus faster while maintaining accuracy (see *Figure 21*).

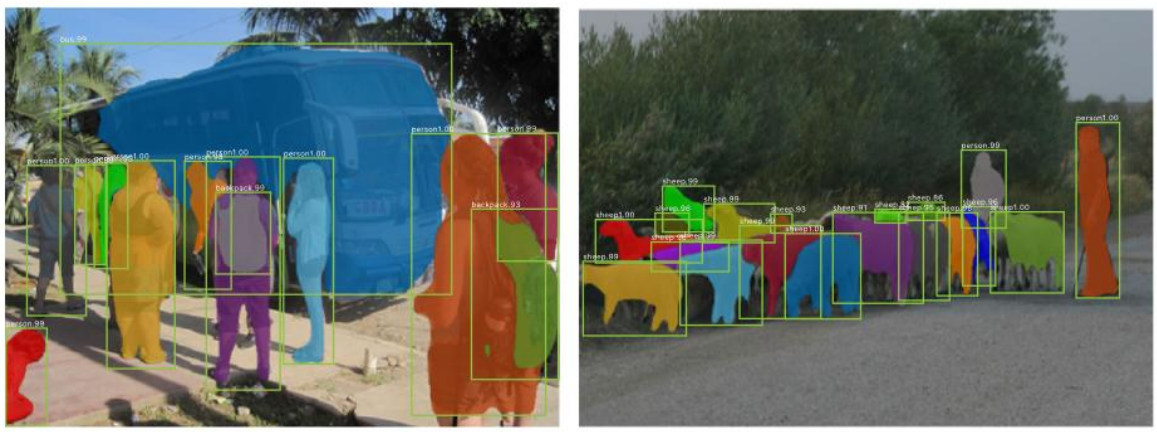


Figure 23. Mask R-CNN results (masks, bounding box, category, and confidence) on COCO test set (HE ET AL. 2020: 388)

At the end, where its quasi predecessor Faster R-CNN had only two outputs, a bounding box and a class label, Mask R-CNN adds a segmentation output for features in an image, the

object mask (see *Figure 23*). Combined with its accuracy, this makes the Mask R-CNN a significant milestone in object detection algorithms. Its influence is shown as the three winning teams of the COCO 2017 competition (a benchmark test like ILSVRC) used an extension of the Mask R-CNN framework (HE et al. 2020: 393).

3.3.1.2 YOLO

YOLOv1 was first launched in 2015 as a single-stage detector and has since been updated and newly released several times by different teams (ALI & ZHANG 2024: 8). Many of the updates included a transformation of the architecture, with different backbones and heads. As single-stage detectors, all the YOLO algorithms are designed for computational efficiency and speed (ibid.). Their inference (the prediction of classes and objects on non-training data) is quick enough to allow for real-time processing, for instance in surveillance or autonomous driving systems (ibid.). The two latest versions, YOLOv12 (TIAN et al. 2025) and YOLOv13 (LEI et al. 2025), were published in February and June 2025, respectively. However, due to its very recent release, the latest version, YOLOv13, was not taken into account in this project.

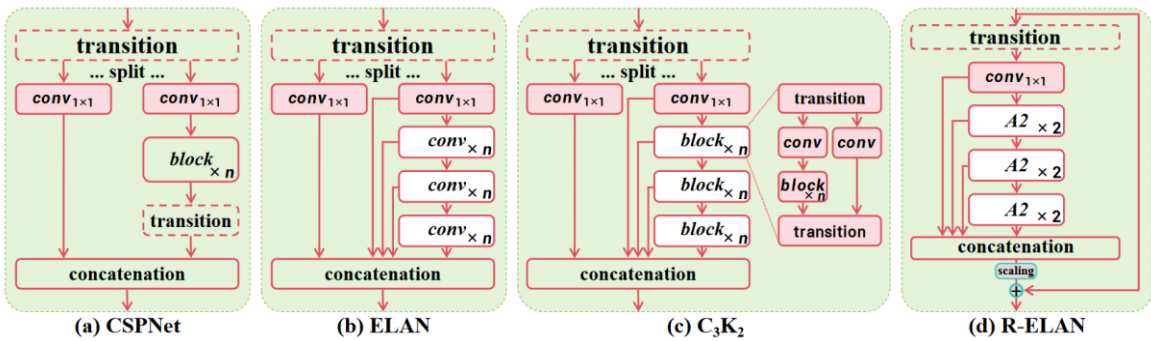


Figure 24. architecture of different YOLO models (TIAN et al. 2025: 4)

Figure 24 displays the architecture of YOLOv4 and YOLOv5 (a), YOLOv7 (b), YOLOv11 (c) and YOLOv12 (d). The general steps are alike in most of the architectures: first a 1 × 1 convolution transitions the initial image, which is followed by a split into two branches (TIAN et al. 2025: 4). In the **CSPNet** architecture (YOLOv5), the right, heavy branch is comprised of convolutional layers and blocks, which are sub-networks of multiple layers. Complex patterns are learned by this branch. The bypass stream on the left preserves the original information of the image. This method is similar to the aforementioned residual

learning (see chapter *3.2.5.2*) as it cuts redundancy and enables optimisation even with very deep networks (TIAN et al. 2025: 4). The split allows parallel, single-stage detection in one pass. The last step, the concatenation, joins the parallel streams together again, which is followed by a final transition to set the output channels (ibid.).

The **ELAN** (YOLOv7) model has a similar architecture, however, the heavy branch on the right features multiple blocks that produce intermediate outputs that are concatenated. Thus, features from different scales and depths are merged (see *Figure 24*). This process is described as Efficient Layer Aggregation Networks (ELAN). (WANG et al. 2023: 7466-7467)

The **C 3 K 2** (YOLOv11) was built on an enhanced design of GELAN (Generalised Efficient Layer Aggregation Network). Hereby, the right, heavy stream is made up of multiple blocks (K 2) with two convolution branches per block, creating a nested network (see *Figure 24*). Similar to ELAN, the intermediate outputs will be merged, while preserving information through the left bypass stream. (TIAN et al. 2025: 4)

Figure 25.oriented bounding boxes generated by YOLOv11 (Ultralytics 2024a)

R-ELAN (YOLOv12) had a fundamental design change as there is no split between initial transition and concatenation (see *Figure 24*). Instead, the feature map is passed through Area Attention blocks (A2). In these steps, they are divided into equal horizontal and vertical areas, with each area being applied attention to by reshape partitioning. This method halves performance costs while retaining a large receptive field (TIAN et al. 2025: 3). Like with ELAN and GELAN architectures, intermediate outputs are processed in concatenation.

However, the R-ELAN (Residual Efficient Layer Aggregation Network) adds an additional step at the end: a scaled residual shortcut. This direct, scaled shortcut (0.01 by default) from input to output improves the optimisation stability (ibid. 4-5).

Most newer YOLO versions offer different image recognition tasks beyond object detection. This includes classification and instance segmentation for the versions v5, v8, v9, v11, and v12 (Ultralytics 2023a, Ultralytics 2023b, Ultralytics 2024b, Ultralytics 2024c, Ultralytics 2025). Additionally, v8, v11, and v12 support oriented bounding boxes (BBs that are generated at angles to fit the object, see *Figure 25*), and pose estimation, with which the key points of (human) poses – the positioning of limbs in relation to the core body – are approximated (Ultralytics 2023c).

This versatility makes YOLO an excellent choice for a wide array of objectives. Combined with its high inference speed that even allows for real-time processing, it has gained a tremendous standing in state-of-the-art DL algorithms.

3.3.1.3 Comparison of R-CNN models and YOLO

Both model families have proven their potency and have been tested in different scenarios over the last ten years. However, their strengths lie in different areas of expertise.

When directly comparing inference times on the COCO dataset, **Mask R-CNN with FPN** takes **195 ms per image**, running on a NVIDIA Tesla M40 GPU (HE et al. 2020: 9); **YOLOv12-n** (*n* is this smallest model size) has a latency of **1.64 ms per image** (other model sizes: *s* – 2.61 ms, *m* – 4.86 ms, *l* – 6.77 ms, and *x* – 11.79 ms), and running on a NDVIDIA T4 GPU (TIAN et al. 2025: 1, 10). Granted, the T4 GPU is a much newer and more efficient processor compared to the Tesla M40, however when XIA et al. (2022: 9) ran a Mask R-CNN inference of COCO images with the T4 GPU, the latency still amounted to 70 ms, which used a ResNet-50 backbone, the fastest they tested.

Of course, the trade-off in speed is not for nothing. Mask R-CNN's instance segmentation allows for pixel-accurate precision and thus strict box alignment. The performance of detection models is measured by different metrics:

1. **Precision**: how accurate the predictions are, i.e. the percentage of correct predictions (STEVENS et al. 2020: 326)

2. **Recall**: the percentage of positives found in a dataset (ibid.: 324)
3. **Precision-recall (PR) curve**: balance between precision and recall at different thresholds, i.e. a low threshold resulting in a higher recall, but lower precision and vice versa (PADILLA et al. 2020: 238)
4. **IoU (Intersection over Union)**: overlap of the predicted box to the ground truth box; calculated through dividing the area of intersection by the area of union, which ranges from 0 to 1 (ibid.)
5. **AP (average precision)**: weighted average of precision values across recall levels; with the COCO dataset, the average over IoU thresholds ranging from 0.50 to 0.95. I.e., AP50 and AP75 are computed at IoU thresholds 0.50 and 0.75, respectively (ibid.: 241)
6. **mAP (mean AP)**: the mean AP across all classes of a dataset (ibid.: 238)
7. **F1**: harmonic mean of precision and recall, $F_1 = 2 \times \frac{precision \times recall}{precision+recall}$ (STEVENS et al. 2020: 328-329)

Rating YOLOv12 and Mask R-CNN on these metrics on the COCO dataset, the former has overall a higher AP – especially at AP50, but with strict IoU thresholding, at AP75, Mask R-CNN provides tighter fitting boxes, which makes it more accurate at small scale. (TIAN et al. 2025: 10; HE et al. 2020: 393)

Model	Speed	Accuracy	Ease of implementation
Faster R-CNN	Bad	Good	Bad
SSD	Good	Good	Bad
YOLO	Good	Good	Good

Table 4. comparison of Faster R-CNN, SSD, and YOLO (recreated from MORE et al. 2024: 143)

To conclude, R-CNN models are an excellent choice if the highest accuracy is paramount but speed is of less concern. When MORE et al. (2024) compared CNN object detection algorithms, they also took into account the ease of implementation, as this can require more time and resources in a project. They determined that in relation to SSD (Single shot detector, another one-stage detector) and Faster R-CNN, YOLO performs better in this area. (see *Table 4*).

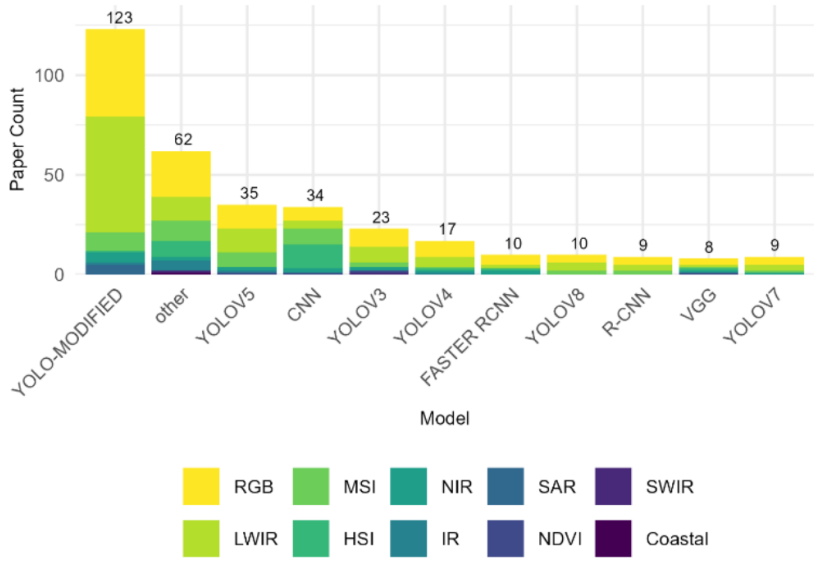


Figure 26. Paper count for top ten CNN models (GALLAGHER & OUGHTON 2025: 11)

As mentioned before, CNN algorithms were designed for different use cases. Thus, it is important to not only focus on results of benchmark datasets, but to also investigate performance reviews on tasks and datatypes that are similar to the intended task. When surveying YOLO multispectral advancements, GALLAGHER & OUGHTON (2025: 10-11) found that out-of-the-box or modified YOLO versions are very popular with multispectral image analysis (see *Figure 26*) – also in respect to agriculture plantations and forests. A more in depth look at the usage of detection and classification algorithms on forests and vegetation will be provided in chapter *3.3*.

3.3.2 Challenges of CNN models in image recognition

Image recognition models such as R-CNNs or the YOLO family come with a set of challenges that must be taken into account when working with them. Some may cause bad prediction metrics on new data, while others can lead to severe mispredictions.

3.3.2.1 Overfitting and underfitting

These two phenomena, overfitting and underfitting, are one of the main challenges of training NNs on data. If a model did not exhaustively learn the details of a dataset, it underfits its purpose and will not produce satisfactory results (LUGER 2025: 396). The other extreme is a model that has been overfit to data. This means that is has been overtrained on the

specific training dataset and will struggle with a new dataset that is more general, which will lead to bad accuracy on data that is unknown to the model (ibid.).

3.3.2.2 Adversarial images

As it has been established, CNNs do not *see* images the way humans do. They use different anker points in an image than the human brain to detect objects or classify it. Since this all comes down to pixel values, tweaking them just by a factor of 0.25 can throw of the algorithm and lead to misclassification or wrong detections (CHARNIAK 2024: 136-137). This action requires intent and does not accidentally occur when working on datasets from reliable sources in a closed system. However, it can be exploited if a CNN is processing live-real-world data that has not been vetted. For example, a street sign can be manipulated to be misread by the computer vision system of a self-driving vehicle (see *Figure 27*). Adversarial vulnerabilities have also been identified in YOLO detector algorithms (CHOI & TIAN 2022).

Figure 27. defaced stop sign to cause misclassification (CHARNIAK 2024: 137)

3.3.2.3 Occlusions, truncations and small objects

Another challenge of image recognition CNNs are occlusions. To an extent, the human eye and brain can disregard obstructions when looking at or through them, e.g. a fence or foliage, and still detect and identify objects behind. CNNs on the other hand, can be greatly influenced by such occlusions (KASSAW et al. 2024: 1). Truncated objects, i.e. objects that partly lie out of the image frame, can also be harder to detect (GUO et al. 2024: 1). Computer vision algorithms may also struggle with very small objects (NIKOUEI et al. 2025: 1). These detection issues must be taken into account when generating training datasets. Ideally, they

represent not just ideal, unobstructed examples of the target object or class, but also include fuzzy samples with truncations, variations in object size, and occlusions. However, these measures are only mitigations. For optimal performance, architecturally modified or specialised algorithms are required to address these obstructions (KASSAW et al. 2024: 16; NIKOUEI et al. 2025: 38).

3.3.2.4 Detection overlaps

Detectors such as YOLO can cause redundancies when multiple predicted BBs overlap on the same identified object (see Annex *II.XIV.I*). This can cause errors in crowded scenes or duplicate detections (HOSANG et al. 2017: 4507). It can be solved by applying a non-maximum suppression algorithm (NMS). This is a post-processing function, with which overlapping detections are filtered or merged, only keeping the highest scoring BB or the ones above a certain IoU threshold (ibid.: 4511).

3.3.2.5 Domain shift and dataset bias

Many CNNs perform tremendously on COCO or other general datasets as they were designed for and (pre-)trained on them. These training images are also referred to as *domain*. They were sourced from a specific camera, with particular light and weather conditions behind certain backgrounds (WANG & DENG 2018: 1-2). Now, if a model is forced to *shift* domains, it may have a hard time to adapt to the changed conditions and therefore produce insufficiently accurate results. A specific type of domain shift issue is the *dataset bias*. This term describes the natural bias of the dataset creator when preparing images for training by choosing them and annotating them (PATEL et al. 2015: 2). Ideally, this would happen with perfect distribution of different image and object qualities (e.g. lighting and truncations), but this is impossible to achieve. However, this can be slightly mitigated through image augmentation during model training (see chapter *5.7.4*).

3.3.2.6 Training complexity and reproducibility

State-of-the-art CNNs use a wide range of complex parameters such as data augmentation methods or hyperparameter tuning (see chapter *5.7*) (KOTTHAPALLI et al. 2025: 3). Only minor adjustments of these parameters can cause variations in inference results, which make exact reproductions of training and inference settings very difficult. Consequently, drawing comparisons across different study setups can be challenging (ibid.: 10-11).

3.3.2.7 Computational cost and data resources

Even with current fast-paced advances in hardware technology, processing power remains a bottleneck across research communities. Cloud computing is a simple fix to accessing computational resources; however, this comes at a financial cost. Large capacity models (e.g. the *l* or *x* model of a YOLO version) require immense processing power that may over-strain research budgets (KOTTHAPALLI et al. 2025: 8).

Another bottleneck is scarcity of labelled datasets when working on object detection (ALZUBAIDI et al. 2023: 1). As every image must be annotated by hand, this is a time-consuming process that requires manpower. Thus, acquiring a large dataset is not always within the resources of a research team.

3.3.2.8 Transparency and ethics

Transparency of data processing is essential for working ethically, especially in sensitive fields such as surveillance or healthcare. Yet, many NN applications are complex and *black boxes*, making it difficult – or sometimes impossible – to disentangle their decision making processes or any biases (DECK et al. 2024: 6). This poses a great challenge that is still to be solved. XAI (explainable AI) are applications that are supposed to help bridge this gap (ibid.: 1).

4 ML and DL algorithms applications on forests and vegetation

Numerous studies have been carried out on the use of ML and DL algorithms for detection of forests or vegetation. In the following chapter, some of the previous ML and DL methods will be briefly highlighted.

The majority of existing frameworks focus on the detection of individual trees and their species. FRANKLIN & AHMED (2018) utilised ML classification in object-based image analysis to identify four tree species with UAV images (RGBI, 0.11 m) in respect to different types of broadleaf trees. Hereby, a an object-based Random Forest (RF) model was trained on the classification of tree crowns to differentiate between species, which was possible thanks to the usage of high resolution data.

Researchers also explored the identification of a single tree species (*Pinus ponderosa*) using RGBI orthophotos and nDSM (both 1-m-resolution), with pixel-based Maximum Likelihood and RF classifications achieving promising result. (RIDDELL et al. 2020)

MIRAKI et al. (2024) coupled UAV generated nDSM (0.03 m) and Sentinel-2 (8 bands, 10 and 20 m) to identify a single tree species (*Populus capsica*) using object-based RF classification.

There are also numerous research examples utilising DL methods such as CNN. The *TreeSatAI Benchmark Archive* by (AHLSWEDE et al. 2023) was a successful approach to creating a multi-label database with the help of multiple sensing methods (aerial images, Sentinel-1 and -2). This used a ResNet, MLP, and LightGBM model for detections, achieving highest F1-scores on ResNet with aerial imagery. It also demonstrated a 2% increase of the F1 when incorporating the NIR band compared to just using RGB (ibid.). The approach differentiated between 20 different species (from 15 genera), including *Abies alba*, but not *Abies nordmannia* (ibid.). BAYRAK et al. (2023) deployed YOLOv8 using images from the *TreeSatAI* benchmark for classification of individual tree identification as well.

Other uses cases of YOLO models in this field, e.g. forest degradation or detection of orchards using RGB and UAV imagery, have also been noted by ALI & ZHANG (2024: 15)

The incorporation of different remote sensing data has also proven to yield accurate results when identifying tree species. The object-based CNN model (ResNet-18) by NEYNS et al. (2023) combines LiDAR (nDSM, 0.5 m) data, orthophotos (RGBI, 0.05 m), and multi-temporal satellite images (RGBI, 3 m) in order to differentiate between five broadleaf tree species.

Mask R-CNN has also been applied as a tool for detection on forests using aerial imagery. CHIANG's et al (2020) aim was to assess tree vitality in order to monitor forest areas for parasite infestation and wildfire prevention.

While there is a broad variety on studies using ML and DL for forest analysis, there are no studies as of August 2025 specifically on the detection or classification of forest plantations such as Christmas tree plantations Furthermore, the majority of the past research on forest analysis focused on individual tree detection. However, the framework in this project focused on aggregated detection or classification, i.e. detecting or classifying multiple rows of trees that are part of a plantation or the whole plantation, as this was suitable to achieve the research objective.

5 Methodology

The following chapters provide an overview of the methodology used to develop the framework in this project. First, the data collected and the software that was used will be described. Second, the different sections of the workflow will be explained. The final chapters will explore the evaluation of the model results and suspicious plantations found.

5.1 Data collection

An selection of datasets was used for drafting and later creating the framework (see *Table 5* for an overview). The different types of data and their origins and functions will be described in the following chapters. The aim was to use recent, coherent data. As the development project commissioned by the State Agency for Forests and Wood NRW took place in 2024, all datasets collected were from the year 2023, if possible. If a dataset has a different year of origin, it is stated in its chapter. All datasets are in the geodetic Coordinate reference system (CRS) ETRS89/UTM32 (EPSG 25832).

Dataset	Type	Date	Size	Resolution	Accuracy	Source
nDSM	GeoTIFF, WMS, i.a.	2023	1 km² per tile	0.5 m	± 0.5 m	Bezirksregierung Köln
TrueDOP	JPG2000, WMS. i.a.	2023	1 km² per tile	0.1 m	± 0.2 - 0.3 m	Bezirksregierung Köln
Legal CTPs	Shapefile	2016	polygons varying in size	n/a	n/a	Wald & Holz NRW
Admin. boundaries	Shapefile, WMS, i.a.	2024	polygons varying in size	n/a	n/a	Bezirksregierung Köln
Damaged areas	Shapefile	2019, 2023	polygons varying in size	n/a	n/a	Wald & Holz NRW
Official woodland	Shapefile, WFS, i.a.	2024	polygons varying in size	n/a	n/a	Bezirksregierung Köln

Table 5. datasets used in the framework

5.1.1 Normalised digital surface models

A normalised digital surface model (nDSM) is the difference of a digital terrain model (DTM) and a digital surface model (DSM). Thus, it shows the relative heights of objects (e.g. vegetation) above the ground surface. The DTMs and DSMs are created with the 3D measurements of airborne laser scanning (ALS) in NRW in a five-year-cycle. However, some of the DSMs are also based on digital aerial images, which are correlated by dense image matching. These image flights are conducted in a four-year-cycle. The nDSMs of the whole area of NRW are available to the public for free (Bezirksregierung Köln 2023a).

The nDSMs are provided by the state government as a Web Map Service (WMS), a Web Coverage Service (WCS), or GeoTIFFs – the latter format being used in the YOLO models. The nDSM in GeoTIFF format are in the CRS ETRS89/UTM32 horizontal and DHHN2016 (EPSG 7837) vertical. The positional accuracy is ± 0.5 m both horizontally and vertically. The GeoTIFF tiles have a side length of 1 km² and the ground sample distance (GSD) is 0.5 m (ibid.)

The nDSMs served two purposes in the framework: one, for providing a filter for the selection of legal CTPs as training data and, two, for evaluating the image classification results. For further use, the raw data was geoprocessed, which is described in chapter *5.5*.

5.1.2 True digital orthophotos

Another essential dataset of this framework were True digital orthophotos (TrueDOPs). Just like the nDSMs, they are provided for free by Bezirksregierung Köln (2023b). The TrueDOPs are a combination of aerial images and DTMs, while removing the distortions of the orthophotos and georeferencing them. The image flights are conducted in a two-year-cycle, alternating between spring and summer.

The TrueDOPs are available for the whole of NRW as WMS, Web Map Tile Service (WMTS), WCS, or JPG2000 (.jp2) with a corresponding worldfile (.j2w). The colour depth is 4 × 8 BIT (red, green, blue, infrared); the GSD is 0.1 m and the JP2s are compiled in tiles with a side length of 1 km². Their CRS is ETRS89/UTM32 (EPSG 25832) horizontal and

DHHN2016 (EPSG 7837) vertical. The positional accuracy is ± 0.2 - 0.3 m. (Bezirksregierung Köln 2023b)

The TrueDOPs were used both as a WMS and as JP2s for different applications. The TrueDOP as a WMS was a useful tool for gaining an overview over the study area, garnering ideas for solution approaches in the framework, and evaluating the results. The JP2-tiles were at the core of the framework, since they were the basis for the training data, as well as for the inference of the CNN model (see chapter *5.6*).

5.1.3 Legal Christmas tree plantations

The State Agency provided a Shapefile with 3,888 polygon features of legal CTPs that were on record in the state of NRW as of 2016 (Landesbetrieb Wald & Holz NRW 2024a). This dataset (see Annex *II.VI*) includes the geometry, information about the administrative affiliation, and the year of creation of the CTP. However, during the work with the geodata it became clear that not all of the CTPs were still active, which meant that the dataset was not up to date in its entirety. Nevertheless, this dataset was crucial for the generation of training data for the YOLO model as well as providing ground truth data for the evaluation of the results of the model (see chapter *5.9*). The legal CTPs Shapefile was preprocessed using the nDSM (see chapter *5.5*).

5.1.4 Administrative boundaries

The administrative boundaries of NRW (and Hesse) are available to the public (Bezirksregierung Köln 2024b; Zentrale Kompetenzstelle für Geoinformation 2025). The NRW dataset is provided as WMS, WCS, GeoTIFF, ASCII (American Standard Code for Information Interchange), or Shapefile. The Hesse dataset is available as WMS.

The datasets include boundaries for the state, region, district, and municipality boundaries. The Shapefile and the WMS were used in the framework for different purposes. Like the TrueDOPs, the administrative boundaries were important as a tool for orientation, overviewing, and the conceptualising of solutions. Furthermore, this dataset was used to create boundaries for the study area of the models.

5.1.5 Damaged areas

As aforementioned, the expansion of CTC has been closely linked to natural hazards in the Sauerland (see chapter *2.1*). Thus, it was important for the conceptualisation of the framework and also for the search for suspicious plantations (see chapter *5.9*), to investigate disaster areas or damaged areas in the region. There are two datasets (Shapefile) (see Annex *II.VII*) on this topic publicly available: the vitality decrease of conifer forest (DE: *Vitalitätsabnahme Nadelwald*) and blowdown areas of the windstorm *Friederike* in 2018 (DE: *Windwurfschadflächen Friederike*) (Landesbetrieb Wald & Holz NRW 2024; Landesbetrieb Wald & Holz NRW 2019).

The vitality decrease of conifer forests Shapefile shows conifer stands that have suffered a decline in vitality between 2017 and 2022. The dataset was generated through analysis of Sentinel 2 imagery and is updated every 6 months. The minimum aggregated area is 0.1 ha, while higher damage levels take priority in aggregation. The features are classified in integers from 0 (no negative change in vitality) to 7 (dead wood or clearing).

The blowdown areas of the windstorm *Friederike* were of lesser importance for the framework, as they include only damaged areas from one event rather than a time series. Furthermore, the windthrow and windsnap only occurred in the east of Westphalia with only very few instances of blowdown in the Sauerland. Thus, this dataset was used for explorative purposes rather than an operative function.

5.1.6 Official woodland areas

The final dataset used in the framework are the official woodland areas. These are derived from the Digital landscape model (DLM; DE: *Digitales Basis-Landschaftsmodell* (Bezirksregierung Köln 2024a). The DLM is publicly available and updated monthly. The DLM is based on DOPs and geodata from the land registry office and it is part of the German *AAA* data model, which consists of AFIS (DE: *Amtliches Festpunktinformationsystem*, EN: official fixed point information system), ALKIS (DE*: Amtliches Liegenschaftskatasterinformationssystem*, EN: official land survey register), and ATKIS (DE: *Amtliches*

Topographisch-Kartographisches Informationssystem, EN: official topographic-cartographic information system).

It is available as Web Feature Service (WFS), NAS (norm-based exchange interface*;* DE: *Normbasierte Austauschschnittstelle*), and Shapefile. The dataset that was used in the framework was a Shapefile (*veg02_f*) from 23 July 2024. It includes all official woodland areas as recorded by the land registry office (see Annex *II.VIII*).

The official woodland areas were important for the search for suspicious plantations in the model results (see chapter *5.9*) as CTPs located in the forest are more likely to be illegal than those on agricultural land.

5.2 Software

The framework was created using different software. Since the development of the CTP detection framework was a pilot project on a limited state budget, opensource programs were chosen over proprietary software, if possible.

The opensource CNN model YOLO was selected for the core of the framework – the prediction of detections or classifications on the TrueDOPs. This model family was chosen because of its versatility, training and inference speed at robust accuracy, and user-friendliness (see chapter *3.3.1.3*). The high versatility meant that different image recognition tasks could be tested with the same framework; a relatively high speed with adequate accuracy was also important as the goal of the project was a usable framework that would be able to handle large datasets efficiently – also beyond this project in the future, in order to provide a capable monitoring tool to the State Agency; and user friendliness was also desired, as it meant a shorter familiarisation period, which makes the process more time and cost effective. Furthermore, other use cases on vegetation and forest data have been conducted successfully with YOLO models (see chapter *4*). A traditional ML method, Random Forest, was also tested as an alternative approach for classification. This model type was selected due to successful applications on tree identification in previous studies (see chapter *4*).

The opensource GIS QGIS was used to view geodata for conceptualisation of the framework, evaluation of the inference results and for minor geoprocessing tasks. Larger datasets

were processed locally, either for testing or for operative use in Python with Visual Studio Code, which is an opensource graphical user interface (GUI).

As the scope of the data grew, the bulk of the operative computation was moved to proprietary software: the cloud-based computation service, Google Colab, and the data storage and management service, Google Drive. They were selected as they provided inexpensive and flexible subscription plans. Google Colab offers a GUI in which all Python code can be scripted and executed. The cloud computation allows for choosing different Tier-levels of CPUs, GPUs, or TPUs (Central, Graphic, or Tensor Processing Unit) depending on the requirements of the operation. This was an important reason to choose Google Colab for the operative execution of code, as it allows for flexible selection of required processing units, which kept the cost low. The use of TPUs was vital to reach an adequate performance as they are specialised in handling NN workloads. The operative training and inference of the YOLO models was executed on Google Colab with a NVIDIA T4. The labelling of aerial imagery for the training datasets was conducted using the software LabelImg (see chapter *5.6*), which was chosen as it offers a user-friendly interface and is available open source.

Furthermore, the Large language model (LLM) interface Perplexity was used as support during the conceptualisation and troubleshooting of Python code. Perplexity was selected as it allows access to all major LLMs through one interface (e.g. Chat GPT, Claude, Gemini, etc.) at a low monthly cost.

5.3 Outline of the workflow

The Hochsauerlandkreis district's data was the main focus of the framework. It was selected after consulting forestry specialists Philipp Würth and Rüdiger Müller, as well as Marc Messerschmidt from the State Agency. Its selection was for two reasons: it is the largest district in the Sauerland and, in comparison to the other districts (Olpe: 684 and Märkischer Kreis: 287), it has by far the highest proportion of legal CTPs (2,183 polygons). A high number of polygons was favourable, because it suggested a bigger selection in training samples. According to the experts, it was also more likely that illegal CTPs could be found there.

The framework was conceptualised using the nDSM and TrueDOPs to gain an overview of the area. The nDSM dataset proved essential due to its filtering capabilities as CTPs feature

a distinct height profile. Thus, the initial step of the framework was the processing of the nDSM to derive the relevant spatial information from it. This served as a filter for the legal CTPs dataset. In the second work step, the training and inference data for the YOLO models was generated. These training samples were garnered using TrueDOPs and the polygons of the legal CTPs. Afterwards, the model underwent iterative training with these images and testing with TrueDOP tiles from the evaluation area (EA). The initial aim was to develop a PoC for the development project, for which a long-term release version, YOLOv5, was chosen, as there is more documentation and literature available for it than for current releases. The third run with YOLOv5 achieved satisfactory accuracy metrics, and the model's effectiveness was assessed on the EA including ground truth data (legal CTPs). Afterwards, the results were also examined using the damaged and official woodland areas (see *Figure 28*).

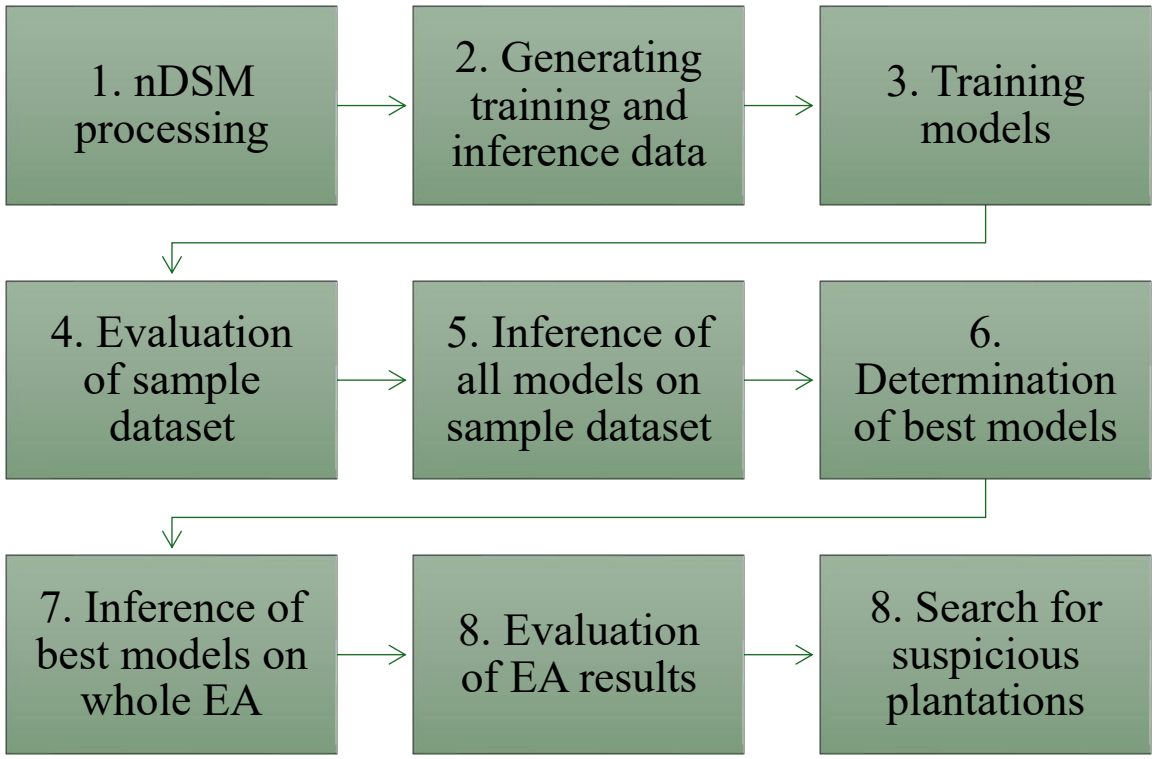


Figure 28. complete workflow of framework

After a satisfactory accuracy was accomplished with v5 model, a visit in the Hochsauerlandkreis took place to observe some results in the field, and to gain insight of the implications of CTC in situ. Furthermore, the findings were presented to stakeholders from State Agency for Forests and Wood NRW and the NRW Ministry of Agriculture and Consumer Protection at the regional forestry office (DE: *Regionalforstamt Oberes Sauerland*) in Schmallenberg.

After the PoC with YOLOv5, several other trial runs with varying configurations followed in order to examine the influence of different datasets, YOLO model versions, and image recognition task types and cross check them with each other (see chapter *5.7.1*).

5.4 Study area and evaluation area

The study area and evaluation area were determined in consultation with Philipp Würth and Rüdiger Müller. As mentioned in the previous chapter, the focus was on the Hochsauerlandkreis from the start. The whole district was thus chosen as the study area (see Annex *II.IX*). All training data originates from this area, and it is being further analysed or processed using filtering datasets (nDSM, damaged areas, and official woodland areas) from the Hochsauerlandkreis as well.

However, a second section had to be selected with which the YOLO model could be evaluated. The EA needed to be of a certain size and also represent an appropriate cross section of the district. Hence, the area needed to include forests, agricultural land, built-up areas, and water bodies. Of course, the evaluation area also needed to contain legal CTPs, as ground truth data and susceptible areas for illegal CTPs. The datasets legal CTPs, damaged areas, and official woodland areas were utilised as tools to narrow down probable EAs in respect to these parameters. The project participants Dr. Jan Dempewolf, Philipp Würth, and Rüdiger Müller made the final decision on an area, which can be seen in Annex *II.X*.

The selected EA is 40,495 ha in size and contains 252 polygons from the legal CTPs dataset. It is located in the northwest of the Hochsauerlandkreis, south of the city Arnsberg. Furthermore, the historic CTC centres Sundern, Meschede, and Bestwig are situated in this area. There is missing TrueDOP data in the northeastern part of the study area, where only older aerial images are available. Since none of the ground truth data indicated that CTPs are present in this section, this area was disregarded in the evaluation entirely.

5.5 nDSM processing

The nDSM was examined at the beginning of the framework. Because CTPs have a specific height restriction that sets them apart from high forest, it served as a useful filter for other

datasets. This threshold was determined to be at **4 m** after consulting forestry experts Rüdiger Müller and Philipp Würth. Above this point, any CTP is considered defunct and will not be monitored anymore because it has turned into regular woodland.

It seemed that not every polygon contained an active CTP when inspecting the legal CTPs Geopackage overlaid with the TrueDOP from 2023. The nDSM ≤ 4 m would discard the plantations that had turned into high forest, but there were also a lot of polygons that were barren with no visible planting. Even though tilling of an area is a strong indicator for intended CTC (see chapter *2.2)*, it is impossible to predict with certainty what use will follow at the point of monitoring. Thus, the polygons that did not display distinct signs of CTC had to be excluded to generate reliable training data. Subsequently, a minimum threshold of **≥ 0.3 m** was set in accordance with experts Philipp Würth and Rüdiger Müller.

It must be taken into account that the accuracy of the nDSM dataset is ± 0.5 m. This imprecision was acceptable, though, as the results were going to be reviewed for errors before further use.

QGIS was used to conceptualise the geoprocessing tools and process the nDSM according to both parameters (≥ 0.3 m; ≤ 4 m). However, while dealing with heavy workloads, the software frequently crashed. As a result, a Python script created and executed in Visual Studio Code was used to complete the operative sequence (see Annex *III.I*).

The first step of the workflow (see *Figure 29*) was Raster_calculation_height thresholds. Here, a raster calculation was performed on the GeoTIFF tiles using the height thresholds (≥ 0.3 m; ≤ 4 m).

This was followed by a second raster calculation, where the clipping mask for the CTP polygon was created. This was supposed to improve performance, as every raster tile could be intersected in the later process with only the polygons that are located on it instead of iterating all polygons over each tile. In this raster calculation, the whole tile was assigned the same value to get a mask for the clipping.

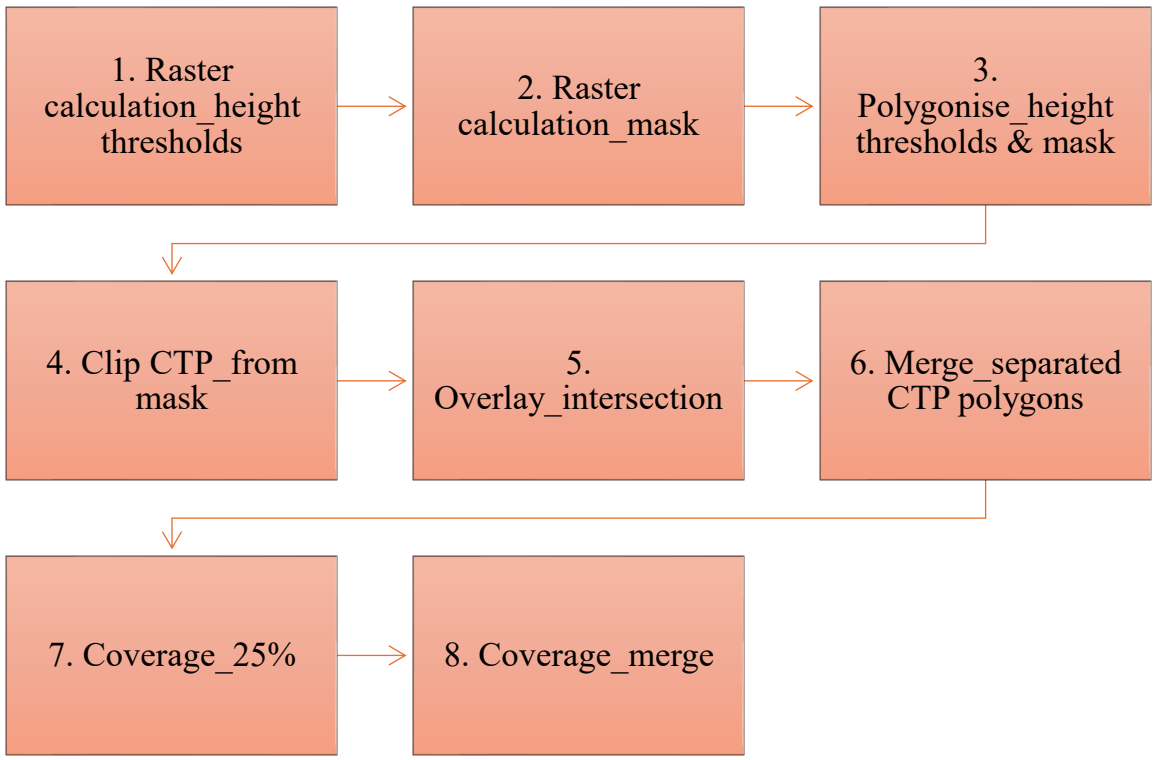


Figure 29. nDSM processing workflow

Afterwards, the raster with height thresholds and the mask raster were polygonised in order to use the following vector based tools on them.

In the fourth and fifth step, the CTP polygons were clipped from the aforementioned mask and intersected with the polygonised height threshold data. Since some polygons might be separated in the previous clipping step, as they may have been located on the border of two vectorised raster tiles, they were merged afterwards.

In the seventh step, the height threshold filter was applied to the CTP polygons to choose the legal plantations that would fall within the desired range. Here, it was decided in consultation with Dr. Jan Dempewolf to apply the filter based of a coverage percentage, in order to not discard CTPs that only had a small coverage of in the undesired height. This coverage percentage was set at 25% – all polygons that were above this threshold would be selected into the training data or ground truth data pool.

The whole workflow (see *Figure 29*) was repeated using the height threshold of **< 4 m** to generate a filter for the inference results of the models as well (see Annex *II.V*). This filter was as an effective tool when evaluating positively classified or detected images as they show directly, if trees are above the height threshold and no longer part of an active CTP.

5.6 Generating training and inference datasets

The YOLO model first requires training in order to *learn* to detect or classify an object class. In this framework, it was initially trained on one class: CTPs. Any image that does not contain a CTP is discarded and annotated as a *negative* result by the model, whereas all images that contain CTPs are annotated as *positive* – granted that it is no false positive (FP) or false negative (FN) prediction. To train the model, enough training images must be supplied. Dataset sizes per object class for YOLO or similar models come in a wide range – from double digits up to thousands (MOSKOVSKAYA et al. 2023: 163). In consultation with Dr. Jan Dempewolf, it was decided that 1,000 images for both categories should be adequate for the framework. 640 × 640 pixel was chosen for the image size, as it is the default extent for YOLO training and inference (Ultralytics 2023d).

Figure 30. CTP with mixed stands of different conifer species south of Ramsbeck, Hochsauerlandkreis (TrueDOP 2023)

Images that contain the item that the model is meant to learn to identify are referred to as positive training images. Any type of landcover (of the study area) that does not include CTPs is featured in the negative training data. This includes agricultural land, meadows, water bodies, normal high forest, and human infrastructure. However, landcover distribution is very diverse and often different types occur in close proximity, which means that multiple categories can be included on one image tile meant for training or detection. This diversity

must also be represented in the training data so the model can be applied to other areas. While a perfect prediction score, where all occurrences of a class are detected in the images, is barely possible, the aim of the framework is to detect CTC areas both on images that are completely or only partly covered by CTPs. This also means that a great diversity in object size in the training images is required.

Figure 31. training data image with artefact in the top left (TrueDOP 2023)

As aforementioned, the predominating species in CTPs in the Sauerland is the Nordmann fir (*Abies nordmannia*). However, it cannot be said with absolute certainty if an aerial image shows Nordmann firs, Norway spruces (*Picea abies*) or Austrian pines *(Pinus nigra).* Because of their blue tint, the common Christmas tree species blue spruce (*Picea pungens*) and noble fir (*Abies nobilis*) are easier to distinguish, but this also cannot be guaranteed, especially in early growth stages. This fuzziness was taken into account, but was regarded as negligible as the growth pattern of CTPs is very distinct, and even if other species are planted in the same way, it is likely that they are grown for this purpose and, thus, under special scrutiny of the authorities. While the aim was to exclude obvious brushwood plantations with noble fir, it is not uncommon that CTPs are grown in practice with mixed stands (see

Figure 30), which was considered when generating training data and evaluating model results.

After uploading the 2,043 TrueDOPs (JP2s with corresponding J2W worldfiles) that compose the study area to Google Drive, they were processed using Google Colab (see Annex *III.II*). In order to improve performance when processing the image files, the first step was to create a dictionary of the J2Ws. Afterwards the preprocessed legal CTP polygons were used to clip the image files. The output was saved as GeoTIFFs, instead of JP2s, as they do not require a separate wordfile, making the handling with GIS software easier. Finally, the clipped GeoTIFFs were separated into 640 × 640 pixel tiles to fit the training data requirements (see Annex *II.XIII.I*). These tiles were filtered in the separation process, on the basis that they were less than 50% comprised of null values. This process discarded images that were not suitable as training data since they were not showing enough CTC area on them.

This generated a pool of 1,892 GeoTIFFs, few of which displayed artefacts stemming from the clipping process (see *Figure 31*). Since there was a large number of possible images available – almost double than what was required – the images containing artefacts were discarded. After the image dataset was cleaned of any corrupted data, it was examined by forestry experts Rüdiger Müller and Philipp Würth to ensure the images show active CTC and are appropriate for training. Here, it was also taken into account to include CTPs of different growth stages, sizes, and shapes. Out of the remaining pool of possible positive training images, 1,000 were randomly selected.

Additionally, it was necessary to label the positive training datasets before use for the YOLO model with object detection algorithms. This was done using the software LabelImg, with which BBs are manually drawn for every image and each object on that image (see *Figure 32*).

In the three runs with YOLOv5 no BBs were passed during the training process. Instead, they were automatically generated to encompass every image as a whole. This error went initially unnoticed. During inference, the majority of the BB-predictions were drawn for the whole image like in a classification. As the results of the third run were excellent, this was considered negligible by the project team. While the models were trained to find any instance of CTC, regardless of size or legal status, the ultimate objective of the project was to search for unauthorised CTPs, which are generally too large to fit into one image tile. As

aforementioned, the general minimum limit for CTPs to be illegal is 2 ha. One image tile is just 0.4094 ha in size, so at least five tiles are needed to compose an illegal plantation. Thus, a classification of a tile in *positive* (CTP) or *negative* (not CTP) without smaller scale object detection was sufficient for the project requirements.

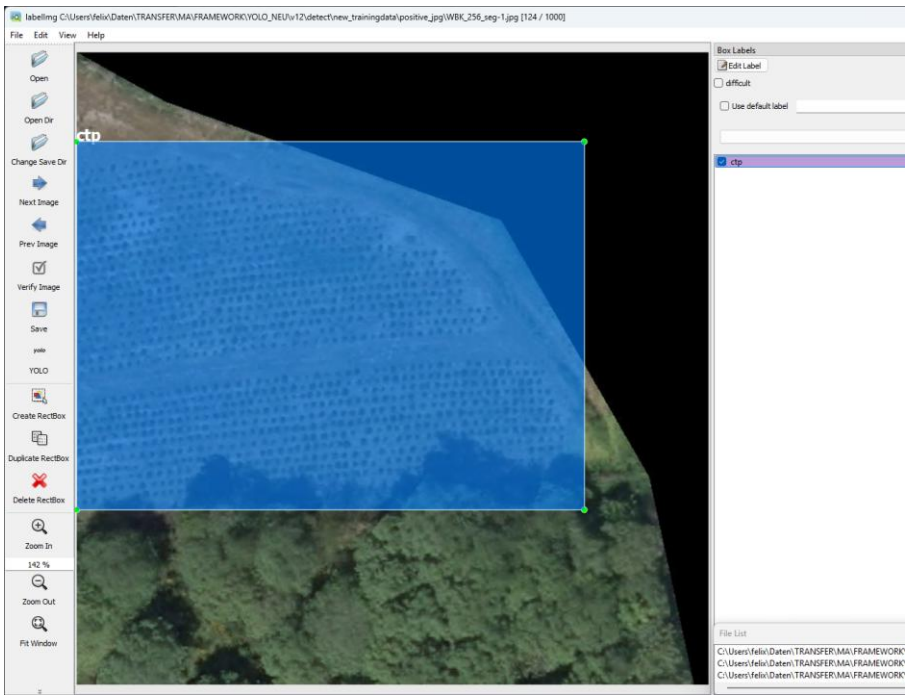


Figure 32. annotating single class training data with BBs in LabelImg

However, after the successful PoC, multiple different model runs were explored using labelled datasets, but also trials with unlabelled training datasets on different model versions or tasks (see chapter *5.7.1*) were conducted. Before using it on the new YOLOv12 model, the training dataset was scrutinised and some images containing very minor artefacts were updated. Initially, this update was done only using a single class (see *Figure 32*). However, a label dataset with two classes, *young CTP* and *old CTP*, was also created and tried on different model iterations (see chapter *5.7.1*). For this, the BBs were drawn again, but this time distinguishing between objects comprised of younger and older CT stands (see *Figure 33*).

The workflow for generating negative image tiles was simpler, as no ground truth data was required (see Annex *III.II.II*). 1,500 image tiles of 640 × 640 pixels were randomly clipped from the TrueDOPs in the study area using python code executed in Google Colab. For this, the previously created dictionary from the J2W file was used. Afterwards, the 1,500 image

tiles were inspected to make sure that no CTPs are displayed. Out of the remaining images that did not show CTPs, a thousand were randomly selected for training of the YOLO models. Furthermore, it was not required to label these images with BBs (see Annex *III.II.II*).

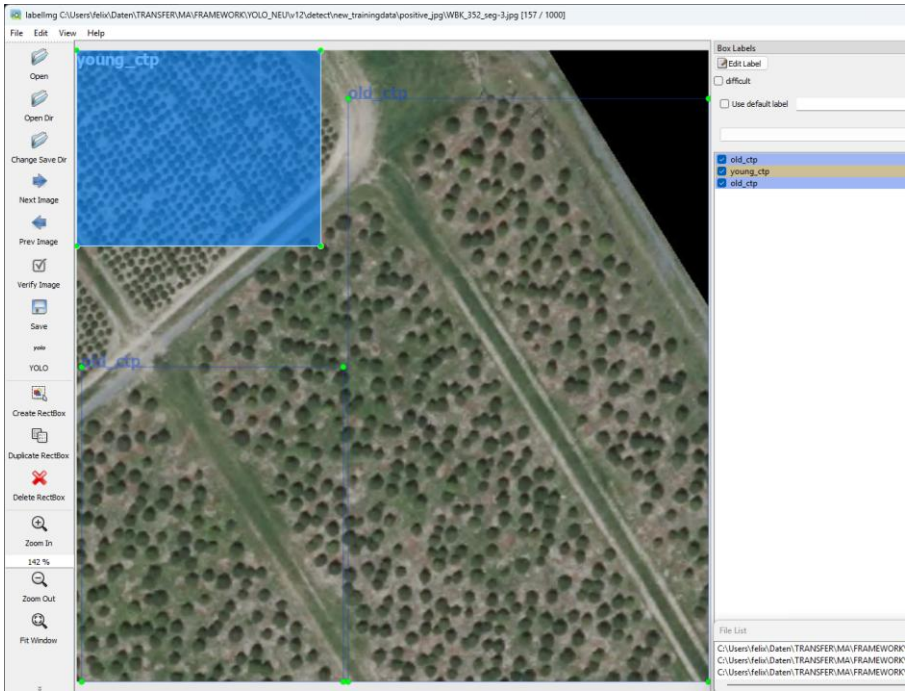


Figure 33. annotating two-class (young and old CTPs) training data with BBs in LabelImg

After the training data was created, the inference data was generated by clipping of GeoTIFFs (640 × 640 pixels) from the TrueDOPs that compose the evaluation area. The process was similar to the previous two work steps, but instead of using polygons as a mask, the whole EA was used. As the clippings did not perfectly fit into the TrueDOP size of 1 km², there are non-square inference data image tiles along the rim of every TrueDOP, but the inference worked on them nevertheless. Furthermore, there was an overlap with the boundaries of the EA, but as both positive and negative training data was taken only from areas that are well distanced from the EA, this was not regarded as an issue.

While generating of the inference dataset, duplicates were created, which was noticed when examining inference results of YOLOv5 run 03. The duplicate images were found and deleted before evaluating true and false negatives of YOLOv5 run 03 and before the inference with the other model versions.

5.7 Training and inference of YOLO models

After the Proof of Concept (PoC) was established for the project – commissioned by the State Agency – through YOLOv5 run 03 in autumn 2024, more test runs with different conditions followed. YOLOv5 run 03 remained the benchmark for the duration of the development of the framework, as it had excellent results and was part of the final submission in the pilot project for the State Agency.

However, as the PoC was achieved using an older version of YOLO, a newer version – the latest long term support release, v12, was also trained and ran on the evaluation data. As the initial results were less promising, different approaches were taken regarding training data, number of classes, image recognition task, and training parameters. The two main changes were double class training data (also see chapter *5.6*) and using classification instead of detection as the image recognition task algorithm. Double class training data was chosen as a new approach, since the appearance of CTPs changes significantly if there are younger or older stands on it. Thus, if there is one class for each, the model can *learn* their visual peculiarities better.

Since the TrueDOPs also include an infrared band (RGBI), a model capable of handling RGBI images was desired. After unsuccessful attempts with YOLOv12, YOLOv11, and YOLOv7, this was achieved using YOLOv8. This model was, yet again, executed with different parameters to compare performances.

5.7.1 Overview of trained models

A total of 22 YOLO model runs were trained and tested on a sample dataset: run 03 of v5, ten runs of v8, and eleven runs of v12 (see Annex *II.XV.I*) These runs varied in the image recognition task, the number of channels used in the TrueDOPs (RGB or RGBI), the training dataset, the classes present in the training dataset (one or two), and the model size – which was set at *m* on all runs after the PoC, as it is the balance between speed (smaller models sizes) and accuracy (larger model sizes). The model runs were also different by the confidence threshold that was selected as the cutoff between a positive and a negative prediction; and the modification of the model with hyperparameter (HPs) tuning or Test-Time Augmentation (TTA). HPs include all external parameters setting the training process of a model,

such as the learning rate, the weight decay, or augmentation methods like mosaic, where four images are combined when learning. The tuning is conducted before the actual learning process starts and narrows down the best possible learning and image augmentation parameter values (Ultralytics 2023e). TTA is applied after training during inference and is supposed to increase recall and mAP. When activated, the algorithm creates an augmented version of each input image (e.g. changing the size or flipping the image) and the detection is performed on all image versions, not just the original image (Ultralytics 2023f). The downside of both techniques is a great reduction in processing time: HP tuning often required multiple hours before training of a model could commence and TTA decreased the inference speed.

Because of the large size of the EA, inference and examination of results take a long time. Thus, not all model versions were assessed on the whole EA. Instead, they were tested on a sample of 3,000 images of one of the four parts (*batches*) that compose the EA, and the best-performing model of each version (v5, v8 with RGBI, and v12) was then assessed on the large dataset to have a more thorough results.

5.7.2 Evaluation area sample test

In the evaluation, the image tiles were examined manually in QGIS using the WMS of the 2023 TrueDOPs as a basemap. All images that contained CTC areas were categorised as true positive – across all cultivation stages, as long as the appearance was characteristic of a CTP. The visual parameters of CTPs, which have been described in chapter *2.2.3*, include uniform appearance, plant distance, driving aisles, or fence. Additionally, the nDSM height mask (< 4 m) was used as a tool to discard any plantations that have grown too tall to be used for Christmas trees.

However, there were also tiles that could be neither confirmed nor discarded, because they fulfilled some of the characteristics, but not beyond doubt (see Annex *II.XIV.II*). This is why the category *potential positive* (PP) was created. Images that fell into that category were neither counted as a true positive (TP) nor as a false positive (FP). However, when measuring the success metrics of the runs, this category played only a minor role as PPs amounted to a fraction of TP and FP.

The table (see Annex *II.XV.I*) displays the result of this evaluation: the TPs – of which were 95 in the sample dataset, the PPs (83), FPs, the true negatives (TN), the false negatives (FN), the total predicted positives.

In order to compare the model run results, the precision, recall, and F1-score (see chapter *3.3.1.3*) were calculated and visualised using python in Visual Studio Code (see Annex *III.III*). The mAP or IoU were not considered as evaluation metrics, since they only work with results that have BBs, but in this case, model results with and without BBs were compared.

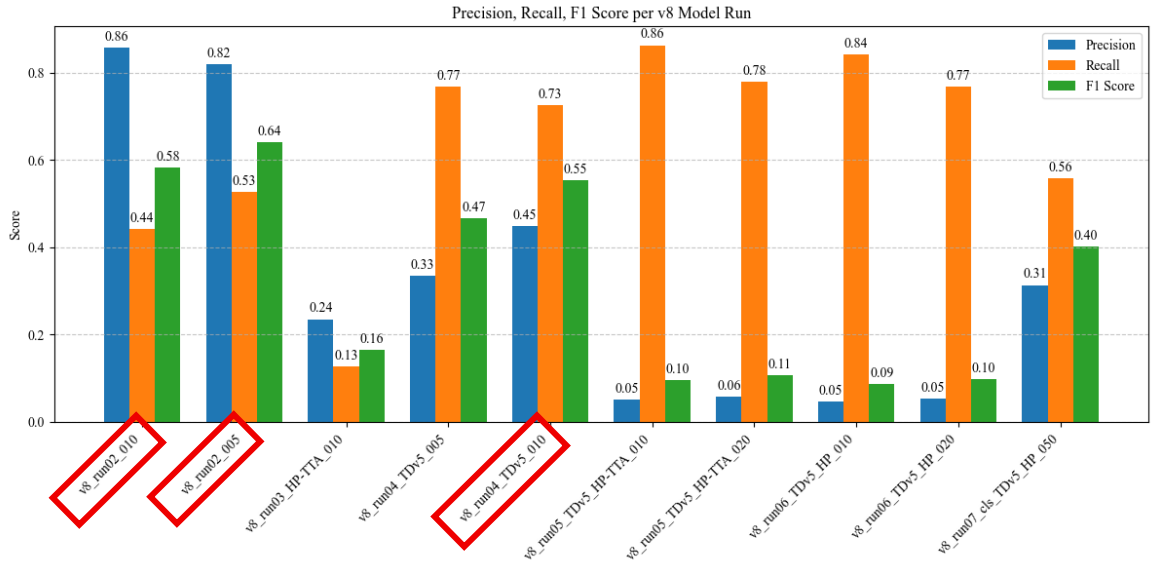


Figure 34. precision, recall and F1 score per YOLOv8 model run on evaluation area sample test

Out of the **v8 model runs** (see *Figure 34*), **run 02** (v8_run02_010 & v8_run02_005), with 0.10 and 0.05 confidence thresholds, had the highest F1 scores (**0.58** and **0.64**, respectively). In both iterations, YOLOv8-m was used for detection task, the single-class labelled v12 training dataset and without HP tuning and TTA. Run 02 also had the highest precision scores at 0.86 and 0.82. However, their recall was among the three lowest at 0.44 and 0.53.

The third best run by F1 was **run 04** (v8_run04_TDv5_010) at a 0.10 confidence threshold with a score of **0.55**. It was set with a detection task, the single-class unlabelled original YOLOv5 training dataset and without HP tuning and TTA. Run 04 with a 0.10 confidence threshold also had the third best precision at 0.45 and above the recall average (0.64) of 0.73. The fourth best run by F1 was run 04 with a 0.05 confidence threshold. It had a higher recall at 0.77 than the previous iteration with 0.10, but also a lower precision (0.33).

Apart from the top four, the majority of the model runs had considerably more extreme metrics. YOLOv8 run 03 used the same parameters as run 02, apart from the modification of HP tuning and TTA, which caused a drop across all three metrics. There were four iterations of run 05 and run 06, using detection on the same unlabelled original v5 training data as run 04, but in addition both HP and TTA were activated in run 05 and only HP in the 06. They exhibited the highest recall metrics, but also the lowest precision at around 0.05. The last v8 model conducted, run 07, was using a classification algorithm on the v5 unlabelled training dataset with a confidence threshold of 0.50 and HP tuning. Its results were the fifth best and displayed a more moderate distribution across the three metrics.

Since the focus of the objective was on recall over precision, as missed CTPs were considered worse than FPs that are found when checking results, run 04 was the ideal compromise between recall and precision in this case.

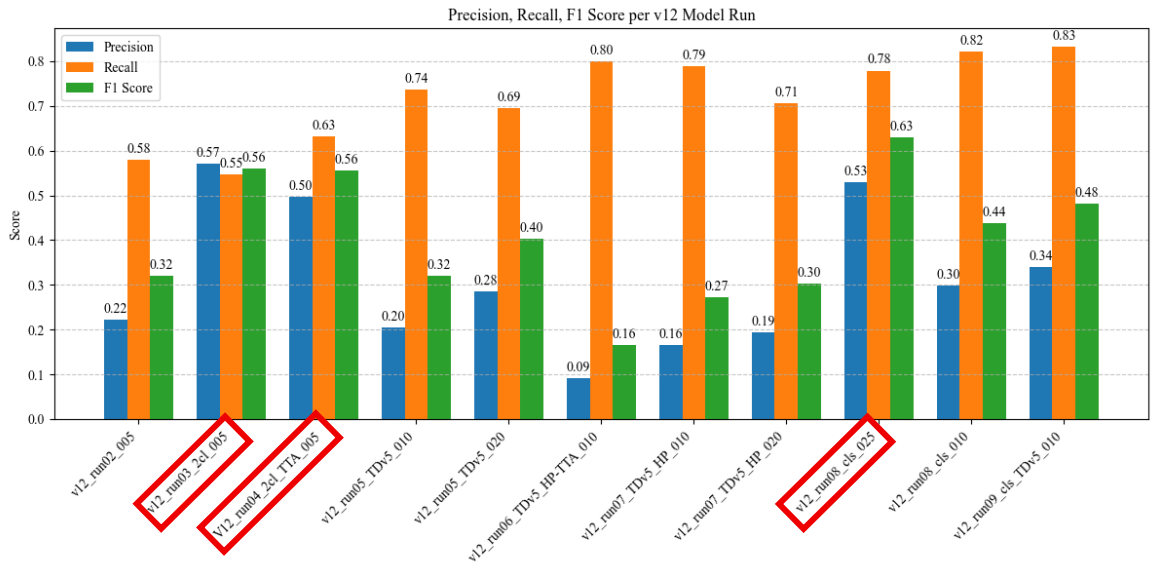


Figure 35. precision , recall and F1 score per YOLOv12 model run on evaluation area sample test

From the v12 model runs (see *Figure 35*), the best-performing by F1 score was the classification **run 08** (v12_run08_cls_025) at **0.63** using the unlabelled, but updated v12 training dataset and a confidence threshold of 0.25. Besides the best performing model run 08 with a 0.25 threshold, another iteration with 0.10 was conducted, which had a higher recall, but also a lower precision, causing the F1 to drop from 0.63 to 0.44. The second and third best performing model runs were **03** (v12_run03_2cl_005) and **04** (v12_run04_2cl_TTA_005)

at **0.56**, which both used detection on the two-class labelled v12 training dataset. Run 03 exhibited the highest precision out of all model runs; run 04's inference was set enabled with TTA, which resulted in a higher recall, but lower precision.

The two iterations of model run 05, which were trained in detection on the original v5 unlabelled dataset, performed well on recall, but badly on precision. Run 06 trained detection on the same dataset, but had HP and TTA activated, which resulted in a high recall, but also the lowest precision out of all v12 runs assessed. Two iterations of run 07 with a 0.10 and a 0.20 threshold, respectively, had only HP activated under the same parameters as run 05 and 06. The two run 07 iterations performed well on recall, but on precision below 0.20. Run 09 was trained using a classification algorithm, but on the original v5 dataset. It exhibited worse metrics than its counterpart, run 08, trained on the v12 training dataset.

As it performed the best by F1 and also had a high recall, run 08 with a 0.25 threshold was chosen out of the v12 model runs.

The **PoC model run 03** outperformed both v12_run08_cls_025 and v8_run04_TDv5_010 on the EA sample test with a **F1 of 0.765, precision of 0.72, and recall of 0.82**.

In the following chapters, the training and inference of the three best performing models of the three YOLO versions will be explained and in chapter *5.9*, their metrics on the whole EA will be examined.

5.7.3 YOLOv5 run 03 – detection with RGB

After installing the YOLOv5 repository and the required python libraries, the first step, before training the model, was setting up the dataset (see Annex *III.IV*). This was split into two folders, 80% for training and 20% for validation, which is YOLO's default setting (Ultralytics 2025b). Afterwards, a preprocessing function for the GeoTIFFs was set up to convert them from RGBI to RGB PNG-files, as they are easier to handle for the algorithm. Then the training dataset structure was established. In this process, the labels for the positive images were also automatically created (also see chapter *5.6*).

A crucial step in the dataset creation is the writing of the *dataset.yaml* (see below). This file holds the instructions for the algorithm regarding where the images are stored, the number

of classes and their names, and also colour channel specifications, if the default of three channels is not used.

```
# Create dataset.yaml file
yaml_content = f"""
train: {dataset_dir}/images/train
val: {dataset_dir}/images/val

nc: 1
names: ['positive']
"""
```

The training itself was initiated with a shell command, with which the train.py from the YOLOv5 repository was executed and the selected parameters were passed (see below).

```
!python train.py --img 640 --batch 16 --epochs 100 --data dataset.yaml --weights
yolov5s.pt
```

The image size was set as 640 × 640 pixels. The *batch* size decides how many images are processed before the gradient is calculated and the weights of a model are updated (Ultralytics n.d. a). A higher batch size requires more GPU memory, but can make the training faster (ibid.). On the other hand, a smaller batch size requires less memory and can help to prevent overfitting – but it can also be slower (ibid.). The *epoch* parameter defines how many times the training will pass over the whole dataset (Ultralytics n.d. b). A low epoch number can result in underfitting, whereas a high epoch number can cause overfitting (ibid.). However, this can be mitigated by *patience* (ibid.). When this function is activated, the training will automatically cease once there was no improvement for *n* epochs (ibid.). The default values were used for both parameters in this run (Ultralytics 2023d). Furthermore, the pretrained weights of the YOLOv5s model were loaded to start the training with. Any other training parameters were left at default. This training took approx. 1 hour.

After the training was finished, the inference on the EA was conducted. The roughly 100,000 images were split in four batches to increase efficiency. In Annex *III.IV*, the code snippets to process just one of these batches is included as an example, since they were repeated with different folder connections.

To commence the inference process, the trained model was loaded and the folder connections set. The results were saved as GeoTIFFs, including the bounding box and the confidence value, but also in a CSV-file. In this run, all images with a confidence value of over

0.5 were categorised as *positive*, between 0.5 and 0.25 as *potential*, and below 0.25 as *negative*. However, both potential and positive images were examined if they were TP or FP.

The inference speed fluctuated between the batches in a range from approx. 45 minutes to 2 hours and 45 minutes, with **processing rates between 2.44 to 9.4 images per second**. This fluctuation did not correlate with batch sizes – which varied by ± 3,000 images. Instead, the speed difference is probably due to connection and access to the processor, which can always vary during code executions in Google Colab.

After all batches ran through inference, the results were also saved in a Geopackage. Even though the results were available as georeferenced GeoTIFFs, this approach proved to be more efficient, as QGIS cannot handle displaying large numbers of GeoTIFFs at the same time well.

5.7.4 YOLOv12 run 08 – classification with RGB

Run 08 also commenced with the installations, imports, and dataset preparation (see Annex *III.V*). The train test split was again set at 80/20, but this time a random seed variable was added to ensure reproducibility. Furthermore, the RGBI-GeoTIFFS were again converted to RGB-PNGs. The Training on this run was not executed by shell command, but by python function (see below).

```
# Enhanced training configuration
model.train(
    data='/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/dataset',  # Ver-
ify exact YAML path
    epochs=200,
    imgsz=640,
    batch=16,
    patience=30,  # Early stopping after 30 epochs without improvement
    optimizer='AdamW', # Adam optimiser with decoupled Weight decay
    lr0=0.001,# Initial learning rate
    weight_decay=0.05,  # Increased L2 regularization
    hsv_h=0.3,  # Stronger hue augmentation
    hsv_s=0.8,   # Increased saturation variation
    hsv_v=0.5,   # Wider brightness range
    degrees=45,   # More rotation augmentation
    translate=0.2,# Position shifting
    scale=0.5,    # Zoom augmentation
    shear=15,     # Perspective distortion
    perspective=0.001,  # Warping effect
    flipud=0.5,   # Vertical flip
    fliplr=0.5,   # Horizontal flip
    mosaic=0.5,   # Mosaic augmentation probability
    mixup=0.2,    # Image mixing
```

```
    copy_paste=0.2, # Object copying
    auto_augment='randaugment',  # Auto augmentation
    erasing=0.2,  # Random erasing
    dropout=0.5,
    project='/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/results',
    name='run08_yolov12m-cls'
)
```

More specific training parameters were tested on this run. The number of epochs was doubled compared to YOLOv5, but *patience* was activated, so the training would stop in case the learning curve plateaued and there is no more improvement. Several learning parameters, such as weight decay, were either activated or increased compared to their default value. While it can lead to longer training time, this is supposed to improve performance (Ultralytics 2023d). The *AdamW* optimiser adjusts the learning rate individually for the parameters during learning process and also provides decoupled weight decay, which should further improve model performance (KINGMA & BA 2017; Ultralytics n.d. c). Several image augmentation parameters such as variation of hue, saturation or flipping were also either activated, or slightly increased from their default values. This helps to generalise the learning process, as it diversifies the training data (Ultralytics n.d. c). An alternative to directly setting these parameters is the aforementioned HP tuning, where the different values are first tested on the training data, before the actual training starts. This was conducted in other model runs, but ultimately did not yield better results with this training dataset (see chapters *5.7.1* & *5.7.2*). After approx. 2 hours and 20 minutes, early stopping was triggered at epoch 66, which means the training had already achieved its best results at epoch 36.

Afterwards, the inference was executed on the four batches comprising the EA. The classification threshold was set at 0.25 and the images were looped over in batches of 64, which was supposed to increase inference speed. At the end, the results of this run were directly taken from the strings of model output and written in a CSV-file, instead of saving every image individually to Google Drive, which was done to improve performance and to save cloud storage space. Furthermore, a log function was added to improve process tracking.

Just like in run 03 with YOLOv5, the inference speed fluctuated remarkably between the different batches without correlation to their batch size. The inference duration on run 08 with YOLOv12 was between approx. 8 and 10 hours with **processing rates between 0.69 and 0.95 images per second**. Thus, the inference speed was much slower than on YOLOv5, which could have been partly due to using model size *m* instead of *s*. Another reason for the

slow processing rate could be the different results saving method, which might have inflated the workload, instead of reducing it. Since the model workings are mostly a black box, the causes could not be determined directly.

The last step was again the transformation from the CSV to a Geopackage, which included all the positively classified image tiles and the other parameters, such as confidence score, as fields. Just like with the previous inference step, a tracking log was introduced.

5.7.5 YOLOv8 run 04 – detection with RGBI

Since the results of YOLOv5 run 03 were excellent and the new labelled data was not achieving similar metrics, the original training dataset, without labels, was tested with different model versions and settings – YOLOv8 run 04 (see Annex *III.VI*) was one of them. In addition, the *dataset.yaml* was set to accept all four channels in order to also include the infrared band (see below). Therefore, the GeoTIFFs were not converted to PNGs in this run. Otherwise, the dataset creation was the same as in previous runs; an 80/20 split was set with a random seed.

```
# Set random seed for reproducibility
random.seed(42)
# Class information
CLASS_NAME = 'ctp'
NUM_CLASSES = 1
[…]
# 1. Create dataset.yaml
dataset_config = f"""
train: {DATASET_PATH}/images/train
val: {DATASET_PATH}/images/val

nc: {NUM_CLASSES}
names: ['{CLASS_NAME}']
channels: 4
"""
```

The majority of training parameters on this run were left at default, but the epochs were set at 200 again, with a patience of 30 like in YOLOv12 run 08 (see code below). Early stopping was triggered at 55 epochs, so it took only 25 epochs to achieve the best training results. The duration of the training was approx. 1 hour and 10 minutes.

```
# ========== CONFIGURATION ==========
MODEL_SIZE = 'm'

# Training parameters
EPOCHS = 200
BATCH_SIZE = 16
IMAGE_SIZE = 640
# ===================================


model = YOLO(f'yolov8{MODEL_SIZE}.pt')
model.train(
    data=DATASET_YAML_PATH,
    epochs=EPOCHS,
    imgsz=IMAGE_SIZE,
    batch=BATCH_SIZE,
    device=0,
    name=f'yolov8{MODEL_SIZE}_04-rgbi_train',
    exist_ok=True,
    patience=30
)
```

The inference was executed with a 0.10 confidence threshold and without saving individual images as results. Each image result is stored in a dictionary with its parameters (such as image name and confidence scores) saved as keys. At the end, all these dictionaries are compiled in a DataFrame and then written into a csv. The inference duration was between approx. 40 minutes to 1 hour with **processing rates between 6.9 and 10.95 images per second**, which means YOLOv8 run 04 had the fastest inference top speed out of the three runs.

After the inference was executed on all batches, the positive results were again transformed into vector data and saved in a Geopackage.

5.8 Alternative machine learning approach with Random Forest

In addition to the YOLO models, an alternative ML approach was tested: Random Forest (RF). RF classifiers learn to identify images using single pixels, patches, or segments. For the purpose of this project, patch-based and segment-based classifications were tested. Pixel-based classification was deemed unfeasible on this dataset as single pixels would not be able to represent the complexity of the object classes.

5.8.1 RF classifier with patches

For training with patches the image is split up into smaller patches that show the object class. This separation into smaller parts is also conducted on the images of the inference dataset. For the training dataset, positive image patches that contain only CTP, young CTP or old CTP were created (for single class and double class training, respectively); and for negative training samples, image patches that do not contain any CTC (see chapter *5.6*).

The selection of the training patch size depends on the dimensions of the object class as well as on the GSD of the dataset. It was important to capture the characteristic pattern of CTC plantations rows and driving aisles, so a certain size minimum had to be chosen to include multiple plantation rows or a driving aisle. On the other hand, large patches can cause a reduction in processing speed.

For this project, a patch size of 64 × 64 pixel was chosen. This size had been successfully applied on similar data before (SOTILLE et al. 2022). It provides a suitable representation of the CTC characteristics and it is practical for splitting the GeoTIFFs (640 × 640) evenly. A larger patch size of 128 × 128 was also tested.

The 64 × 64 (see Annex *II.XIII.III*) and 128 × 128 patches were clipped from the centre of training data GeoTIFFs previously used for training of the YOLO models. The patches were then manually inspected to ensure they represent the desired class. This was done for a single class model (*ctp* and *negative*) and also for a double class model (*young_ctp*, *old_ctp*, and *negative*). After the patches were inspected, a random sample of 1,000 images per class was selected for training (see Annex *III.VII*).

Initially, the training was conducted with default parameters (100 estimators) and no additional channels for the single class and double class models. Afterwards, the Normalised Difference Vegetation Index (NDVI), a common remote sensing index for observing vegetation, which is computed from the red and infrared colour channels (TUCKER & SELLERS 1986: 1408), was tested as a fifth channel during training (see code snippet below). This meant that the RF classifier would be conducted on 64 × 64 × 5 vectors (with just RGBI: 64 × 64 × 4).

```
Set random seed for reproducibility
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
```

```
[…]
PATCH_SIZE = 64
NUM_CHANNELS = 4
NUM_CHANNELS_WITH_NDVI = 5  # RGBI + NDVI
[…]
        # Calculate NDVI = (NIR - Red) / (NIR + Red)
        red = patch[:, :, 0].astype(np.float32)
        nir = patch[:, :, 3].astype(np.float32)
        denom = (nir + red)
        # Avoid divide by zero
        denom[denom == 0] = 1e-6
        ndvi = (nir - red) / denom

        # Append NDVI as new 5th channel
        patch_with_ndvi = np.dstack((patch, ndvi))

        # Flatten and store
        X_list.append(patch_with_ndvi.flatten())
        y_list.append(cls)
[…]
# Split train/test stratified by class
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=RANDOM_SEED, stratify=y_encoded)

# Train Random Forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=RANDOM_SEED)
rf.fit(X_train, y_train)
```

Furthermore, the RF classification was tested with different numbers of estimators, *N*, i.e. the number of trees *grown*. A range from 10 to 500 has been proven to work in other studies (BELGIU & DRĂGUŢ 2016: 25). For the RF patch classification setup, models with 100, 200, 300, 400, and 500 estimators were tested. The other main parameter for RF configuration is *m*, which is "the number of variables used to split each node" (AKAR & GÜNGÖR 2012: 107). Previous studies showed that the best results are achieved when *m* is the square root of the variables (ibid.). This default setting was also used in the model setup.

After training, the inference was computed on the EA sample of 3,000 GeoTIFFs. For this, they had to be transformed into the same size (64 × 64 and 128 × 128, respectively) that was used for training (see *3. Preprocess inference dataset* in Annex *III.VII*). Thus, the inference was conducted on approx. 300,000 patches. For every patch, the algorithm predicts a class (*ctp* or *negative*; *young_ctp*, *old_ctp*, or *negative*). In order to have a comparative evaluation to the YOLO model results, the inference on the patches was assessed on the original 3,000 GeoTIFFs, not the individual patches.

For each model iteration, the results were transformed to features in a Geopackage and evaluated at three thresholds: **at least one patch** per GeoTIFF tile is predicted positive (i.e. *ctp*;

young_ctp, or *old_ctp* respectively), at least **10 patches** per tile are predicted positive, or at least **20 patches** per tile are predicted positive. Finally, the precision, recall and F1 score were calculated for every model iteration at all three thresholds.

The best model on the 64 × 64 dataset was also tested on a 128 × 128 dataset with the same evaluation setup.

5.8.2 RF classifier with segmentation

For RF classification with segmentation, the image is separated with a segmentation algorithm (see also chapter *3.3.1.1*). Each image is divided into smaller parts, however, not along a grid of boundaries, but along edges of features in the image (see *Figure 36*) to create homogenous segments. The RF classification method with segmentation was only conducted in a single class setup (*ctp*, *negative*), because of the large manual workload and computational resources required for an iteration.

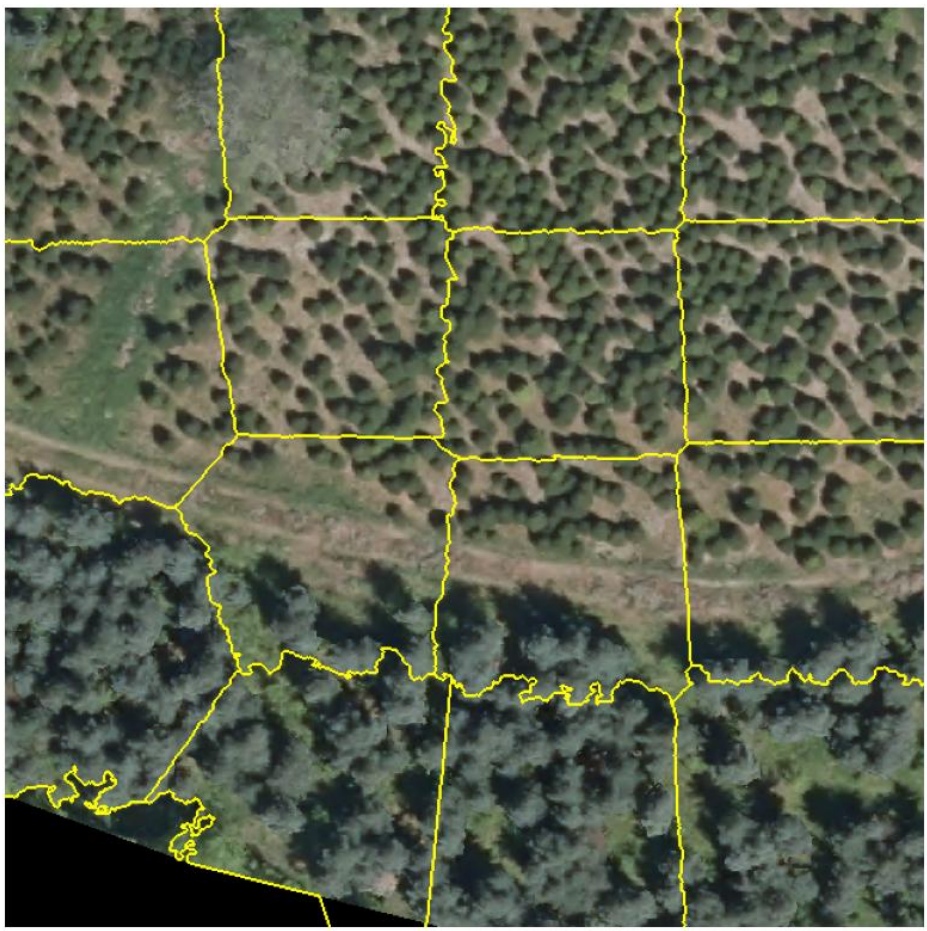


Figure 36. segment boundaries on positive training data GeoTIFF for RF classification

The positive training data was generated by segmentation of GeoTIFFs that already functioned as training data in YOLOv12 models. After trials with different parameters, the segment size was set at (maximum) 20, which in practice resulted in approx. 16 to 17 segments per GeoTIFF.

The segments (ca. 16,000) generated by the 1,000 positive training data images were then manually inspected to ensure they were assigned the correct label. After this inspection, approx. 12,500 correct *ctp* segments remained, which could be used as positive training data. The rest was to be used as negative training data. Another segmentation was computed on previously used negative training data images, which generated again approx. 16,000 segments. Out of the roughly 19,500 negative segments, 12,500 were randomly selected to match the number of positive training data segments.

Finally, the numerical spectral statistics such as median, mean, minimum or maximum value for each band (including NDVI) of the labelled segments were extracted from the segments and saved in a CSV-file (see *Extract vector features per segment* in Annex *III.VIII*). This was the input for the RF classification training. Before the training, a Hyperparameter search was conducted to test searching for the best training parameters (see code snippet below). Because of the lack of improvement in iterations with higher number of estimators for RF classifications with patches, the tested number of estimators were limited to 100 and 200.

```
# Hyperparameter grid for Random Forest
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [15, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5],
    'max_features': ['sqrt', 'log2']
}

# Initialize base Random Forest classifier
rf_base = RandomForestClassifier(random_state=42, n_jobs=-1)

# GridSearchCV for hyperparameter tuning
grid_search = GridSearchCV(
    rf_base,
    param_grid,
    cv=3,
    scoring='accuracy',
    n_jobs=-1
)
```

```
Best Parameters: {'max_depth': 30, 'max_features': 'sqrt', 'min_sam-
ples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 200}
```

The RF classification with segmentation also required a preprocessing of the inference dataset (640 × 640 GeoTIFFs) into segments of which numerical spectral statistics were extracted and saved as a CSV-file. The trained RF model then predicted the class for each segment. Just like with the patch inference, different thresholds per GeoTIFF (at least 1, 2, 3, 4, 5, 6, 7, or 8 positive segments per image) were tested to find the best performance.

5.8.3 RF classifier models' results

The alternative ML approach was first tested on the EA sample dataset (see Annex *II.XV.II*). The results of the RF classification with patches were not sufficient for operative use. None of the tested iterations (different numbers of estimators, inclusion of NDVI as a fifth channel, change in patch size to 128) yielded a substantial improvement. The single class (*ctp* and *negative*) runs' best iteration, *RF_1class_64_RGBI_500_10*, only achieved an F1 score of **0.063**. While the runs with two positive classes (*young_ctp* and *old_ctp*) achieved better results, they still scored far below YOLO results. The best performing model was *RF_2class_64_100_20* with a F1 of **0.132**.

With all patch classification iterations, the number of False Positives was very high, compared to YOLO models or also RF classification with segments, which resulted in a very low precision. The highest precision among RF patch classification runs, was the best performing model *RF_2class_64_100_20* with 0.072.

Regarding the operation speed, this method was much slower than YOLO. While the training took only 5 to 20 minutes, the inference on the EA sample dataset lasted between 10 minutes to 3 hours per iteration. As this dataset included only 3,000 images, the operative use on the entire EA dataset (approx. 100,000) or in other use scenarios would be inefficient.

Run	TP	PP	FP	Total predicted Positives	TN	FN	EA sample	Precision	Recall	F1
RF_Seg._6	68	10	27	105	2868	27	3000	0.716	0.716	0.716
v5_run03_025	78	18	31	127	2856	17	3000	0.716	0.821	0.765

Table 6. comparison of EA sample dataset inference results of model runs RF_Segmentation_6 and YOLOv5_run03_25

The RF classification with segmentation yielded much better results than the iterations with patches (see bottom of Annex *II.XV.II*). The best run, *RF_Segmentation_6* achieved a **F1 of 0.716,** a **precision of 0.716,** and a **recall of 0.716**. This is the **second best performance** of all model runs (including YOLO) on the EA sample dataset **after *v5_run03_025*** (see *Table 6*), however, the RF classification with segmentation required tremendous computation time to achieve it. While the inference was computed in a few minutes, the preprocessing of segments, in which the numerical spectral statistics are calculated and saved in a CSV-file, took approx. 3 hours for the training data and another 3 hours for the inference dataset. Preliminary tests suggested a preprocessing time of approx. 35 to 40 hours for the whole EA, thus this method lacks scalability compared to the YOLO models. Since this framework is developed for future use scenarios, the RF classification with segmentation was suspended, even though it achieved good inference results.

5.9 Evaluation of EA inference results

Each of the three models' predicted positives of the complete EA were evaluated by hand using the Geopackages containing the polygons of the image frames in QGIS, which were created at the end of every model run. The TrueDOP WMS from 2023 functioned as a basemap and the preprocessed nDSM (< 4m) worked as an additional filter.

The initial evaluation of v5 run 03 was carried out in autumn 2024 by Rüdiger Müller and Philipp Würth; the evaluations of v8 run 04 and v12 run 08 were completed later. They were based on the CTP characteristics, as well as the TPs and FPs of the initial v5 run 03 evaluation. In addition, the previously mentioned category of PP (potential positive) was introduced.

Run	TP	PP	FP	Total predicted Positives	TN	FN	1% sample	EA images	GT polygons (of 253)
v5_run03	**3560**	**95**	**1244**	**4899**	940	21	961	100978	**247**
v8_run04	3364	213	3163	6740	921	22	943	100978	191
v12_run08	4356	377	3269	8002	915	**15**	930	100978	245

Table 7. evaluation results of the models v5 run 03, v8 run 04, and v12 run 08

Because of the large number of predicted negatives, a 1% sample of each model was randomly selected and evaluated. This sample size varies, because the models produced different amounts of negatively predicted images (see *Table 7*). Furthermore, the ground truth (GT) polygons (legal CTPs) were also used as an additional metric (see Annex *II.XII*). The correct GT rate was determined by counting all the polygons that were covered by at least one positively predicted image by the model.

The **recall** (see *Figure 37*) is very high on all three models (**0.978**, **0.977**, and **0.984**), which is excellent, as it was important to ensure that CTPs were not missed. However, the correct ground truth rate shows a steep drop in v8 run 04 (0.755) compared to v5 run 03 (0.976) and v12 run 08 (0.968). Furthermore, both v8 run 04 (0.515) and v12 run 08 (0.571) have a much lower precision than v5 run 03 (0.741). This is further underpinned by the **F1 score**, which is by far the highest for **v5 run 03** (**0.843**) compared to **v8 run 04 (0.675)** and **v12 run 08 (0.723)**.

In conclusion **v5 run 03 is the best-performing model.** The only metric in which another model, v12 run 08, performs better, is recall. However, the difference (0.978 vs. 0.984) is so small, that it can be accepted, considering the better correct GT rate of v5 run 03 and the overall much higher precision.

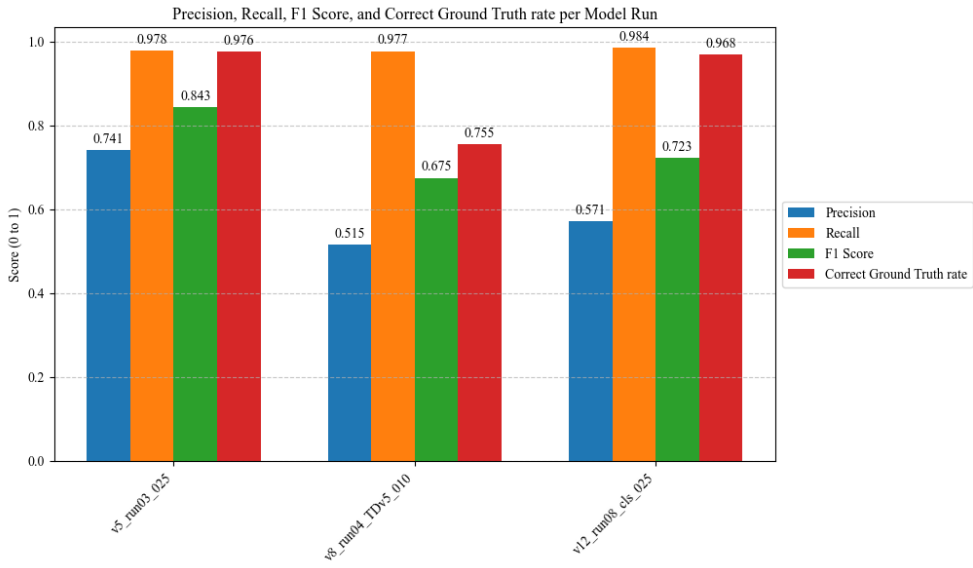


Figure 37. precision, recall, F1 score, and correct ground truth rate per YOLO model run on EA dataset

The spatial results for YOLOv5 run 03, YOLOv12 run 08, and YOLOv8 run 04 can be seen in Annexes *II.XVI.I*, *II.XVI.II*, and *II.XVI.III*, respectively.

When examining the spatial distribution of the predicted positives of the three models, it was striking that YOLOv8 has accumulations in urban areas (especially visible in the west of the EA, see Annex *II.XIV.III*) and also quarries (visible in the far east of the EA, see Annex *II.XIV.IV*), which did not occur with the other model results. YOLOv5 and YOLOv12 have a more similar distribution of FPs, but with YOLOv12 results having a higher accumulation in the same regions – e.g. woodland areas in the south (see Annex *II.XIV.V*), which contain a lot of Brushwood plantations or defunct CTPs.

Regarding the TP results of all three models, there is a concentration (see Annex *II.XIV.VI*) in the area south of Bestwig – a historical centre of CTC. Since CTPs are often marked by multiple image tiles, an aggregation of neighbouring TP can give an approximation of the number of plantations. This is only a rough estimate, as sometimes different plantations lie side by side, and would be counted as one through the aggregation or image tiles on the same plantations, but may not be connected. This was done with QGIS geoprocessing (*dissolve*). In YOLOv5 run 03, **approx**. **595 plantations were detected** in the EA; in YOLOv8 run 04, it was 501 plantations; in YOLOv12 run 08, there were 494 plantations identified.

5.10 Search for suspicious plantations

In the final step of the framework, the model results were crosschecked to search for plantations that are suspicious because of their location on woodland or damaged area. This was already conducted with the initial results of YOLOv5 run 03 in autumn 2024 to create a prioritised output dataset for the State Agency. The new results, from model runs with YOLOv8 and YOLOv12, were checked as well. It is important to note, that **the results that will be presented here are not instances of illegal CTPs**, as this can only be determined by the local authorities. There is a wide range of regulations that may result in exceptions and make a CTP legal even if it is on woodland or damaged areas. Therefore, these plantations will be referred to as *suspicious* not *illegal*.

To make sure no potential CTPs go undetected, the categories TP and PP of each model were merged before they were intersected with the woodland and damaged areas in QGIS. After the intersection, the results were checked by hand to conclude if the TP or PP image tile is actually on a CTP that is located on woodland and/or a damaged area, rather than caught in the intersection because part of the tile, that is not covered in CTC, is woodland

and/or damaged area. The output was a dataset for each model with image tiles that are on woodland and/or on clear cut woodland (because of damage to the conifers), which can help prioritise the search of local authorities. Furthermore, this step of the process tested if the involvement of datasets containing woodland and damaged areas may help with the search for suspicious plantations. The results are presented in *Table 8* and on maps in Annexes *II.XVII.I*, *II.XVII.II*, and *II.XVII.III*.

Run	TPs & PPs on woodland (n/total TPs & PPs in EA)	TPs & PPs on damaged areas	TPs & PPs on damaged areas (aggregated)
v5_run03	724/3655	32	12
v8_run04	755/3577	67	28
v12_run08	1260/4733	27	12

Table 8. results of suspicious plantations per model

YOLOv12 run 08 has the highest number and rate of TPs and PPs on woodland per total TPs and PPs (27%), whereas the rates of YOLOv5 run 03 (20%) and YOLOv8 run 04 (21%) are much lower. Regarding the plantations on damaged areas (clear cut), which is even more crucial when it comes to suspicious plantations, model YOLOv8 run 04 had the highest number – both by individual image tile and by aggregation. YOLOv5 run 03 had the second highest by number, but tied with the results of model YOLOv12 run 08 on aggregated plantations.

Out of the detected suspicious plantations, one was examined and confirmed as an allegedly illegal CTP on a field trip with Marc Messerschmidt (see *Figure 6* & Annex *II.XVII.I*). The legal status of the other suspicious plantations that were found is at present undeclared.

As previously highlighted, no direct claims are implied regarding the legal status of any plantations presented here.

6 Conclusion

This research project successfully demonstrated that both machine learning and deep learning approaches can be harnessed to detect Christmas tree plantations in aerial imagery with substantial accuracy and operational relevance. Two primary algorithmic strategies were investigated: the traditional Random Forest (RF) classifier from the machine learning domain, and the YOLO family of convolutional neural network (CNN) models representing state-of-the-art deep learning approaches.

The Random Forest classification was evaluated using both patch-based and segmentation-based workflows. While the patch-based RF approach struggled with low precision and overall performance, the segmentation-based RF workflow achieved much improved results, yielding an F1 score of 0.716, with precision and recall likewise at 0.716. This placed the segmentation-based RF performance just behind the top-performing YOLOv5 run 03 on the evaluation area sample dataset. However, despite its effectiveness in terms of inference accuracy, the RF segmentation pipeline was significantly constrained by processing inefficiencies – especially during data preprocessing for segment feature extraction – that limited its operational scalability. As a result, the focus of the framework shifted towards YOLO models, which offered a superior balance of detection accuracy and processing efficiency for large-scale, automated monitoring tasks.

Systematic evaluation of YOLOv5, YOLOv8, and YOLOv12, across a series of model configurations, established a robust and practical methodology for automated plantation monitoring, with YOLOv5 run 03 achieving optimal metrics with a F1 score of 84.32%, precision of 74.10%, and recall of 97.81%. In addition, the consistent high recall performance across all three final model runs (97.81%, 97.67%, 98.39%) meets the project requirements regarding false negative errors, as this represents greater operational risk than false positive detections. This performance consistency validates the robustness of YOLO architectures for Christmas tree plantation identification tasks.

Precision values varied significantly between architectures: YOLOv5 run 03 (74.10%), YOLOv8 run 08 (51.54%), and YOLOv12 run 04 (57.13%). The lower performance quality in newer architectures contradicts assumptions that architectural complexity correlates with improved detection accuracy. This finding indicates that established architectures may be

better optimised for specific computer vision tasks without requiring extensive hyperparameter tuning.

Processing efficiency statistics revealed substantial variations in inference speeds, ranging from YOLOv8's peak performance of 10.95 images per second to YOLOv12's reduced processing rate of 0.69 to 0.95 images per second. These processing differences appear independent of batch size configurations and may be due to varying approaches in output management. With the established processing rates of the best-performing model YOLOv5 (2.44 to 9.4 images per second), an implementation for regular monitoring cycles across larger areas such as the whole Hochsauerlandkreis, the Sauerland region, or even the entire state of North Rhine-Westphalia would be feasible.

The RGBI implementation in YOLOv8 produced increased false positive rates, particularly concentrated in urban environments, contradicting the research hypothesis that infrared band integration would enhance detection performance. This outcome suggests spectral confusion between built infrastructure materials and plantation vegetation when incorporating near-infrared wavelengths into the classification process.

Analysis indicates that RGB imagery provides sufficient spectral information for plantation identification, as the geometric and structural features of Christmas tree plantations offer adequate discriminative characteristics. The uniform canopy structure and consistent patterns of plantation rows – even of different growth stages – are effectively captured through standard three-channel aerial imagery without requiring additional spectral bands for accurate classification.

Classification on image tile level and detection without regular bounding box detection achieved superior performance metrics compared to object detection using training datasets with bounding box annotations. As the original unlabelled YOLOv5 dataset consistently outperformed manually labelled datasets, it is indicated that extensive manual annotation does not guarantee performance improvements. This finding suggests that dataset quality and representativeness may be more critical factors than labelling precision for this specific application domain.

This approach complied with the project purpose as object localisation was only required to identify plantation presence per tile rather than precise boundaries within the image tile.

Integration of normalised Digital Surface Models (nDSM) enabled effective height-based filtering to distinguish active Christmas tree plantations from mature forest stands exceeding 4-meter height thresholds. This multi-modal approach demonstrates the value of combining spectral imagery with elevation data for vegetation structure assessment.

The incorporation of damaged areas and official woodland datasets enabled identification of suspicious plantation activities, with field verification confirming at least one allegedly unauthorised plantation. However, because of the limited data available at present regarding findings in the field, more validation is required to ensure a more comprehensive assessment.

The primary research question – *Can orthophotos be analysed using ML to locate Christmas tree plantations?* – is confirmed. Machine learning approaches successfully identify Christmas tree plantations with high accuracy, achieving F1 scores exceeding 84%. This result closes the identified research gap in detecting Christmas tree plantation and Nordmann fir monocultures.

Regarding the specific working questions - *Does involving RGBI data instead of RGB improve model results?* – this research project showed a negative conclusion. RGBI implementation increased false positive detection rates without corresponding improvements in true positive identification, particularly in urban environments. Standard RGB imagery proves sufficient for accurate plantation detection without requiring additional spectral complexity.

Can nDSM and damaged areas help in finding suspicious plantations? – This question can be confirmed. The multi-modal approach successfully identified suspicious plantations with field verification, demonstrating practical utility for regulatory enforcement by the authorities. However, expanded validation procedures would strengthen confidence in this application.

Based on performance evaluation, YOLOv5 architecture is recommended for operational deployment, combining superior performance metrics with processing efficiency. RGB imagery provides sufficient spectral information for accurate detection without requiring multispectral processing complexity.

Integration of nDSM data should be maintained as standard protocol as a vegetation height filter between active plantations and high forest. The classification approach is preferred

over object detection methodologies with accurate bounding boxes for monitoring applications where plantation presence identification is prioritised over precise boundary delineation.

The State Agency for Forests and Wood NRW confirmed that project results met operational requirements and commissioned future monitoring implementation. This validation indicates successful transition from research prototype to operational deployment.

The automated monitoring framework addresses fundamental limitations of manual inspection, enabling systematic coverage across large areas with consistent detection. This enables enforcement of forestry regulations supporting ecosystem protection and wildlife conservation objectives. The validated framework provides technological infrastructure for evidence-based environmental management and policy enforcement at scale.

The integration of machine learning technologies in environmental monitoring represents technological advancement in sustainable forest management. Increasing pressures from climate change and intensifying land use patterns require automated monitoring tools for effective landscape management.

This research successfully established a scientifically validated framework for Christmas tree plantation monitoring that addresses both regulatory enforcement requirements and environmental objectives. State authority acceptance and planned operational implementation confirm the practical applicability and contribution to sustainable forest management practices in North Rhine-Westphalia.

7 List of references

AHLSWEDE, Steve, SCHULZ, Christian, GAVA, Christiano, HELBER, Patrick, BISCHKE, Benjamin, FÖRSTER, Michael, ARIAS, Florencia, HEES, Jörn, DEMIR, Begüm & KLEINSCHMIT, Birgit (2023): TreeSatAI Benchmark Archive: a multi-sensor, multi-label dataset for tree species classification in remote sensing. – *Earth System Science Data*, 15/2: pp. 681–695. doi: 10.5194/essd-15-681-2023.

AKAR, Özlem & GÜNGÖR, Oğuz (2012): Classification of multispectral images using Random Forest algorithm. – *Journal of Geodesy and Geoinformation*, 1/2: pp. 105–112. doi: 10.9733/jgg.241212.1.

ALI, Momina Liaqat & ZHANG, Zhou (2024): The YOLO Framework: A Comprehensive Review of Evolution, Applications, and Benchmarks in Object Detection. doi: 10.20944/preprints202410.1785.v1.

ALPAYDIN, Ethem (2016): Machine learning: the new AI. – Cambridge, Massachusetts London, England (The MIT Press).

ALZUBAIDI, Laith, BAI, Jinshuai, AL-SABAAWI, Aiman, SANTAMARÍA, Jose, ALBAHRI, A. S., AL-DABBAGH, Bashar Sami Nayyef, FADHEL, Mohammed A., MANOUFALI, Mohamed, ZHANG, Jinglan, AL-TIMEMY, Ali H., DUAN, Ye, ABDULLAH, Amjed, FARHAN, Laith, LU, Yi, GUPTA, Ashish, ALBU, Felix, ABBOSH, Amin & GU, Yuantong (2023): A survey on deep learning tools dealing with data scarcity: definitions, challenges, solutions, tips, and applications. – *Journal of Big Data*, 10/1: 46. doi: 10.1186/s40537-023-00727-2.

ALZUBAIDI, Laith, ZHANG, Jinglan, HUMAIDI, Amjad J., AL-DUJAILI, Ayad, DUAN, Ye, AL-SHAMMA, Omran, SANTAMARÍA, J., FADHEL, Mohammed A., AL-AMIDIE, Muthana & FARHAN, Laith (2021): Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. – *Journal of Big Data*, 8/1: 53. doi: 10.1186/s40537-021-00444-8.

BAYRAK, Onur Can, ERDEM, Fırat & UZAR, Melis (2023): Deep learning based aerial imagery classification for tree species identification. – *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLVIII-M-1–2023: pp. 471–476. doi: 10.5194/isprs-archives-XLVIII-M-1-2023-471-2023.

BELGIU, Mariana & DRĂGUŢ, Lucian (2016): Random forest in remote sensing: A review of applications and future directions. – *ISPRS Journal of Photogrammetry and Remote Sensing*, 114: pp. 24–31. doi: 10.1016/j.isprsjprs.2016.01.011.

Bezirksregierung Köln (2023a): Normalisiertes Digitales Oberflächenmodell NRW. – Retrieved January 30, 2025, from https://www.bezreg-koeln.nrw.de/geobasis-nrw/produkte-und-dienste/hoehenmodelle/digitale-oberflaechenmodelle/normalisiertes.

Bezirksregierung Köln (2023b): Digitale Orthophotos NRW. – Retrieved January 30, 2025, from https://www.bezreg-koeln.nrw.de/geobasis-nrw/produkte-und-dienste/luftbild-und-satellitenbildinformationen/aktuelle-luftbild-und-0.

Bezirksregierung Köln (2024a): Digitales Landschaftsmodell NRW. – Retrieved February 4, 2025, from https://www.bezreg-koeln.nrw.de/geobasis-nrw/produkte-und-dienste/landschaftsmodelle/aktuelle-landschaftsmodelle/digitales-basis.

Bezirksregierung Köln (2024b): Digitale Verwaltungsgrenzen NRW. – Retrieved January 31, 2025, from https://www.bezreg-koeln.nrw.de/geobasis-nrw/produkte-und-dienste/verwaltungskarten-und-grenzen/digitale-verwaltungsgrenzen.

BORK, Karin (2025): Behandlung von Rüsselkäferbefall in Kulturen. – *LWF Merkblatt*, 44. – Retrieved February 26, 2025, from https://www.lwf.bayern.de/service/publikationen/lwf_merkblatt/244293/index.php.

BRAND, Mark H. (undated): Abies procera Noble Fir. – (University of Connecticut Plant Database) *Plant Database*. – Retrieved September 17, 2025, from https://plantdatabase.uconn.edu/detail.php?pid=6.

BREIMAN, Leo (2001): Random Forests. – *Machine Learning*, 45/1: pp. 5–32. doi: 10.1023/A:1010933404324.

Bundesverband der Weihnachtsbaum- und Schnittgrünerzeuger (undated): FAQ. – Retrieved March 13, 2025, from https://bvwe.de/faq/.

CABI (2021): CABI International. Elatobium abietinum. – Wallingford, UK (CABI Head Office) *CABI Compendium*. – Retrieved February 26, 2025, from https://doi.org/10.1079/cabicompendium.30897.

CARMESIN, Jannis (2023): Der Weihnachtsbaumdetektiv. – Retrieved February 27, 2024, from https://www.zeit.de/gesellschaft/2023-12/illegale-weihnachtsbaumkulturen-wald-sauerland-nachrichtenpodcast.

CECH, Thomas L, PERNY, Bernhard & TALGØ, Venche (2009): Neue Ergebnisse zur Kabatina-Nadelschütte der Tannen (Kabatina abietis). – Retrieved February 25, 2024, from https://bfw.ac.at/400/pdf/fsaktuell_48_8.pdf.

CHARNIAK, Eugene (2024): AI & I: An Intellectual History of Artificial Intelligence. – (The MIT Press). doi: 10.7551/mitpress/15482.001.0001.

CHIANG, Chia-Yen, BARNES, Chloe, ANGELOV, Plamen & JIANG, Richard (2020): Deep Learning based Automated Forest Health Diagnosis from Aerial Images. – *IEEE Access*, 8: pp. 144064–144076. doi: 10.1109/ACCESS.2020.3012417.

CHOI, Jung Im & TIAN, Qing (2022): Adversarial Attack and Defense of YOLO Detectors in Autonomous Driving Scenarios. – (arXiv). doi: 10.48550/arXiv.2202.04781.

Commission of the European Communities (1988): Annual report / ESPRIT: European Strategic Programme for Research and Development in Information Technology ; 1987. – Luxembourg (Commission of the European Community).

DECK, Luca, SCHOEFFER, Jakob, DE-ARTEAGA, Maria & KÜHL, Niklas (2024): A Critical Survey on Fairness Benefits of Explainable AI. – pp. 1579–1595,. doi: 10.1145/3630106.3658990.

DELIPETREV, Blagoj (2020): AI Watch, historical evolution of artificial intelligence: analysis of the three main paradigm shifts in AI. – LU (Publications Office).

FINK, Andreas H., BRÜCHER, Tim, ERMERT, Volker, KRÜGER, Andreas & PINTO, Joaquim G. (2009): The European storm Kyrill in January 2007: synoptic evolution, meteorological impacts and some considerations with respect to climate change. – *Natural Hazards and Earth System Sciences*, 9/2: pp. 405–423. doi: 10.5194/nhess-9-405-2009.

Forest Research (undated): Caucasian fir, Nordmann fir (NMF). – (Forest Research Publications). – Retrieved September 17, 2025, from https://www.forestresearch.gov.uk/tools-and-resources/tree-species-database/131571-caucasian-fir-nordmann-fir-nmf/.

FRANKLIN, Steven E. & AHMED, Oumer S. (2018): Deciduous tree species classification using object-based analysis and machine learning with unmanned aerial vehicle multispectral data. – *International Journal of Remote Sensing*, 39/15–16: pp. 5236–5245. doi: 10.1080/01431161.2017.1363442.

GAILLY, Robin, PAQUET, Jean-Yves, TITEUX, Nicolas, CLAESSENS, Hugues & DUFRÊNE, Marc (2017): Effects of the conversion of intensive grasslands into Christmas tree plantations on bird assemblages. – *Agriculture, Ecosystems & Environment*, 247: pp. 91–97. doi: 10.1016/j.agee.2017.06.029.

GALLAGHER, James E. & OUGHTON, Edward J. (2025): Surveying You Only Look Once (YOLO) Multispectral Object Detection Advancements, Applications, and Challenges. – *IEEE Access*, 13: pp. 7366–7395. doi: 10.1109/ACCESS.2025.3526458.

GUO, Baicang, ZHANG, Hongyu, WANG, Huanhuan, LI, Xinwei & JIN, Lisheng (2024): Adaptive occlusion object detection algorithm based on OL-IoU. – *Scientific Reports*, 14/1: 27644. doi: 10.1038/s41598-024-78959-2.

HAGGE, Jonas, SEIBOLD, Sebastian & GRUPPE, Axel (2019): Beetle biodiversity in anthropogenic landscapes with a focus on spruce plantations, Christmas tree plantations and maize fields. – *Journal of Insect Conservation*, 23/3: pp. 565–572. doi: 10.1007/s10841-019-00146-w.

HE, Kaiming, GKIOXARI, Georgia, DOLLAR, Piotr & GIRSHICK, Ross (2020): Mask R-CNN. – *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42/2: pp. 386–397. doi: 10.1109/TPAMI.2018.2844175.

HE, Kaiming, ZHANG, Xiangyu, REN, Shaoqing & SUN, Jian (2015): Deep Residual Learning for Image Recognition. – (arXiv). doi: 10.48550/arXiv.1512.03385.

HLEG (2019): A definition of AI. Main capabilities and scientific disciplines. – B-1049 Brussels (European Commission).

HO, Tin Kam (1995): Random decision forests. – pp. 278–282, *3rd International Conference on Document Analysis and Recognition* Montreal, Que., Canada (IEEE Comput. Soc. Press). doi: 10.1109/ICDAR.1995.598994.

HOSANG, Jan, BENENSON, Rodrigo & SCHIELE, Bernt (2017): Learning Non-maximum Suppression. – pp. 6469–6477, *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* Honolulu, HI (IEEE). doi: 10.1109/CVPR.2017.685.

IT.NRW (2024): NRW: Die größte Anbaufläche für Weihnachtsbäume gibt es im Hochsauerlandkreis. – Retrieved March 13, 2025, from https://www.it.nrw/nrw-die-groesste-anbauflaeche-fuer-weihnachtsbaeume-gibt-es-im-hochsauerlandkreis-127023#:~:text=Wie%20das%20Statistische%20Landesamt%20weiter,B%C3%A4ume%20als%20ein%20Jahr%20zuvor.

JAGEL, Armin & DÖRKEN, Veit Martin (2019): Abies nordmanniana. Nordmann-Tanne (Pinaceae). der Tannenbaum. – *Jahrbuch des Bochumer Botanischen Vereins*, 10: pp. 189–198.

KÄMPFER, Steffen, LÖFFLER, Franz, BRÜGGESHEMKE, Jonas & FARTMANN, Thomas (2022): Untangling the role of a novel agroecosystem as a habitat for declining farmland birds. – *Annals of Applied Biology*, 181/3: pp. 367–378. doi: 10.1111/aab.12789.

KASSAW, Kaleb, LUZI, Francesco, COLLINS, Leslie M. & MALOF, Jordan M. (2024): Are Deep Learning Models Robust to Partial Object Occlusion in Visual Recognition Tasks? – (arXiv). doi: 10.48550/arXiv.2409.10775.

KINGMA, Diederik P. & BA, Jimmy (2017): Adam: A Method for Stochastic Optimization. – (arXiv). doi: 10.48550/arXiv.1412.6980.

KOTTHAPALLI, Manikanta, RAVIPATI, Deepika & BHATIA, Reshma (2025): YOLOv1 to YOLOv11: A Comprehensive Survey of Real-Time Object Detection Innovations and Challenges. – (arXiv). doi: 10.48550/arXiv.2508.02067.

Kreisverwaltung Warendorf (undated): Weihnachtsbaum- und Schmuckreisigkulturen. – *Serviceportal.Kreis-Warendorf*. – Retrieved March 24, 2025, from https://serviceportal.kreis-warendorf.de/detail/-/vr-bis-detail/dienstleistung/403/show.

Landesbetrieb Wald & Holz NRW (2019a): Grundsätzliche Betrachtung der Weihnachtsbaumnutzung auf Waldflächen als Vor- oder Nebennutzung durch Wald und Holz NRW. – (Landesbetrieb Wald & Holz NRW).

Landesbetrieb Wald & Holz NRW (2019b): Windwurfschadflächen Friederike NRW. – Retrieved January 31, 2025, from https://gdk.gdi-de.org/geonetwork/srv/api/records/e41d7917-bd9b-40c0-aa3a-4e9e7de1ba35.

Landesbetrieb Wald & HOLZ NRW (2024a): Legale Weihnachtsbaumkulturen.

Landesbetrieb Wald & Holz NRW (2024b): Vitalitätsabnahme Nadelwald. – Retrieved January 31, 2025, from https://www.opengeodata.nrw.de/produkte/umwelt_klima/wald_forst/fernerkundung/.

Landesbetrieb Wald & Holz NRW (undated): Orkan Kyrill und seine Folgen in Nordrhein-Westfalen. – *Wald & Holz NRW*. – Retrieved January 27, 2025, from https://www.wald-und-holz.nrw.de/en/wald-in-nrw/wald-und-klima/kyrill-und-seine-folgen-in-nrw.

Landtag NRW (2013a): Gesetzentwurf - Gesetz zur Änderung des Landesforstgesetzes. – Retrieved January 29, 2025, from https://www.landtag.nrw.de/portal/WWW/dokumentenarchiv/Dokument/MMD16-2097.pdf.

Landtag NRW (2013b): Änderung des Landesforstgesetzes vom 3.12.2013.

Landtag NRW (2019): Landesforstgesetz für das Land Nordrhein-Westfalen. – Retrieved January 3, 2025, from https://www.wald-und-holz.nrw.de/fileadmin/Publikationen/20190629_wuh_Landesforstgesetz_web.pdf

Landtag NRW (2020): Drucksache 17/12109. – Retrieved March 25, 2025, from https://www.landtag.nrw.de/portal/WWW/dokumentenarchiv/Dokument/MMD17-12109.pdf.

LEI, Mengqi, LI, Siqi, WU, Yihong, HU, Han, ZHOU, You, ZHENG, Xinhu, DING, Guiguang, DU, Shaoyi, WU, Zongze & GAO, Yue (2025): YOLOv13: Real-Time Object Detection with Hypergraph-Enhanced Adaptive Visual Perception. – (arXiv). doi: 10.48550/arXiv.2506.17733.

LOHRER, Thomas (2013): Tannennadelrost. – *ARBOFUX - Diagnosedatenbank für Gehölze*. Retrieved February 26, 2025, from https://www.arbofux.de/tannennadelrost.html.

LUGER, George F. (2025): Artificial Intelligence: Principles and Practice. – Cham (Springer Nature Switzerland). doi: 10.1007/978-3-031-57437-5.

MAURMANN, Karl Heinz (2015a): Weihnachtsbaumhochburg Sauerland. – *Geographische Kommission für Westfalen*. – Retrieved January 17, 2025, from https://www.westfalen-regional.de/de/weihnachtsbaumhochburg_sauerland/.

MAURMANN, Karl Heinz (2015b): Weihnachtsbaumkulturen aus ökonomischer und ökologischer Perspektive. – Retrieved January 20, 2025, from https://www.westfalen-regional.de/de/weihnachtsbaumkulturen_oekologie/.

MAXWELL, Aaron E., WARNER, Timothy A. & FANG, Fang (2018): Implementation of machine-learning classification in remote sensing: an applied review. – *International Journal of Remote Sensing*, 39/9: pp. 2784–2817. doi: 10.1080/01431161.2018.1433343.

MCCARTHY, John, MINSKY, Marvin L., ROCHESTER, Nathaniel & SHANNON, Claude E. (1955): A proposal for the Dartmouth summer research project on artificial intelligence. – Hanover, New Hampshire, USA.

MESSERSCHMIDT, Benno (2024): Detektion und rechtliche Einordnung von Weihnachtsbaum- und Schmuckreisigkulturen.

MESSERSCHMIDT, Marc, ANDERMAHR, David & KORTMANN, Greta (2024): Exkursionsführer. Ungenehmigte Waldumwandlungen am Beispiel von neu angelegten Weihnachtsbaumkulturen im Wald.

MIRAKI, Mojdeh, SOHRABI, Hormoz, FATEHI, Parviz & KNEUBUEHLER, Mathias (2024): Coupling UAV and satellite data for tree species identification to map the distribution of Caspian poplar. – *Landscape Ecology*, 39/2: 30. doi: 10.1007/s10980-024-01846-8.

MLV NRW (1994): Gesetz zur Sicherung des Naturhaushalts und zur Entwicklung der Landschaft (Landschaftsgesetz - LG). – Retrieved March 24, 2025, from https://recht.nrw.de/lmi/owa/br_bes_text?sg=1&menu=0&bes_id=5365&aufgehoben=J&anw_nr=2#NORM.

MLV NRW (2023): Oh Tannenbaum: Jeder dritte Weihnachtsbaum stammt aus Nordrhein-Westfalen. – Retrieved March 12, 2025, from https://www.mlv.nrw.de/wp-content/uploads/2023/12/mlv-01.12.2023.pdf.

MORE, Ghansham, PATIL, Omkar, MORE, Omkar, MORE, Mire, SURYAVANSHI, Samadhan & MALI, Manisha (2024): Comparison of Object Detection Algorithms CNN, YOLO and SSD. – *Int. J. Sci. R. Tech.,*, 11/1: pp. 137–144.

MOSKOVSKAYA, Elizaveta, CHEBOTAREVA, Olesya, EFIMOVA, Valeria & MURAVYOV, Sergey (2023): Predicting dataset size for neural network fine-tuning with a given quality in object detection task. – *Procedia Computer Science*, 229: pp. 158–167. doi: 10.1016/j.procs.2023.12.017.

NIERHAUS-WUNDERWALD, Dagmar (2000): Rostpilze an Fichten. – *Merkblatt für die Praxis*, 32. – Retrieved February 26, 2025, from https://www.wsl.ch/de/publikationen/rostpilze-an-fichten.

NIKOUEI, Mahya, BAROUTIAN, Bita, NABAVI, Shahabedin, TARAGHI, Fateme, AGHAEI, Atefe & MOGHADDAM, Mohsen Ebrahimi (2025): Small Object Detection: A Comprehensive Survey on Challenges, Techniques and Real-World Applications. – *Intelligent Systems with Applications*, 27: 200561. doi: https://doi.org/10.1016/j.iswa.2025.200561.

PATEL, Vishal M, GOPALAN, Raghuraman, LI, Ruonan & CHELLAPPA, Rama (2015): Visual Domain Adaptation: A survey of recent advances. – *IEEE Signal Processing Magazine*, 32/3: pp. 53–69. doi: 10.1109/MSP.2014.2347059.

PEFC (2017): PEFC-Standards für Weihnachtsbaumkulturen auf Waldflächen. – PEFC D. – Retrieved January 29, 2025, from

https://www.pefc.de/media/filer_public/47/b5/47b5c8c2-53be-4a72-b24b-f249cbc4ef24/pefc_d_1002-2_v2_pefc-standards-fuer-weihnachtsbaumkulturen.pdf.

RICHTER, Felix (2016): Wer verdient am Weihnachtsbaum? – *Statista*. – Retrieved March 14, 2025, from https://de.statista.com/infografik/7259/preiszusammensetzung-weihnachtsbaum/.

RIDDELL, Audrey P., FITZGERALD, Stephen A., QI, Chu & STRIMBU, Bogdan M. (2020): Classification Strategies for Unbalanced Binary Maps: Finding Ponderosa Pine (Pinus ponderosa) in the Willamette Valley. – *Remote Sensing*, 12/20: 3325. doi: 10.3390/rs12203325.

ROSENBLATT, Frank (1958): The perceptron: A probabilistic model for information storage and organization in the brain. – *Psychological Review*, 65/6: pp. 386–408. doi: 10.1037/h0042519.

RUSSAKOVSKY, Olga, DENG, Jia, SU, Hao, KRAUSE, Jonathan, SATHEESH, Sanjeev, MA, Sean, HUANG, Zhiheng, KARPATHY, Andrej, KHOSLA, Aditya, BERNSTEIN, Michael, BERG, Alexander C. & FEI-FEI, Li (2015): ImageNet Large Scale Visual Recognition Challenge. – *International Journal of Computer Vision*, 115/3: pp. 211–252. doi: 10.1007/s11263-015-0816-y.

RUSSELL, Stuart & NORVIG, Peter (2016): Artificial Intelligence. A Modern Apporach. 3rd Edition. – (Pearson Education Inc.).

Sauerland-Tourismus e.V. (2025): Sauerland-Tourismus zieht Bilanz für das Jahr 2024. – Retrieved March 11, 2025, from https://www.sauerland.com/de/service/journalisten-blogger/presseinformationen/sauerland-tourismus-zieht-bilanz-fuer-das-jahr-2024.

SCHRÖTER, Hansjochen, JOHN, Reinhold & PETERCORD, Ralf (2009): Die Stamm- und Triebläuse der Weißtanne. – *FVA Waldschutz-Info*, 3/2009.

SEJNOWSKI, Terrence J. (2018): The Deep Learning Revolution. – Cambridge, MA (The MIT Press).

SOTILLE, Maria E., BREMER, Ulisses F., VIEIRA, Gonçalo, VELHO, Luiz F., PETSCH, Carina, AUGER, Jeffrey D. & SIMÕES, Jefferson C. (2022): UAV-based classification of maritime Antarctic vegetation types using GEOBIA and random forest. – *Ecological Informatics*, 71: 101768. doi: 10.1016/j.ecoinf.2022.101768.

Sparkasse (2024): Baum, Gans und Glühwein. Das sind die Preise 2024. – Retrieved March 28, 2025, from https://www.sparkasse.de/aktuelles/weihnachten-preise.html#weihnachtsbaeume-2024-schoen-nachhaltig.

STANGE, Lisa (2016): Rechtliche Rahmenbedingungen und waldbauliche Empfehlungen für die Weiterbehandlung von Weihnachtsbaumkulturen im Regionalforstamt Oberes Sauerland. Schmallenberg (Regionalforstamt Oberes Sauerland).

STEVENS, Eli, ANTIGA, Luca & VIEHMANN, Thomas (2020): Deep learning with PyTorch. – Shelter Island, NY (Manning Publications Co).

STREITBERGER, Merle & FARTMANN, Thomas (2020): Effects of Christmas-tree plantations on phytodiversity: implications for conservation. – *New Forests*, 51/5: pp. 869–886. doi: 10.1007/s11056-019-09767-0.

STREITBERGER, Merle & FARTMANN, Thomas (2021): Phytodiversity in Christmas-tree plantations under different management regimes. – *Weed Research*, 61/3: pp. 178–187. doi: 10.1111/wre.12468.

SUN, Yibo, SUN, Zhe & CHEN, Weitong (2024): The evolution of object detection methods. – *Engineering Applications of Artificial Intelligence*, 133: 108458. doi: 10.1016/j.engappai.2024.108458.

TIAN, Yunjie, YE, Qixiang & DOERMANN, David (2025): YOLOv12: Attention-Centric Real-Time Object Detectors. – (arXiv). doi: 10.48550/arXiv.2502.12524.

TUCKER, C. J. & SELLERS, P. J. (1986): Satellite remote sensing of primary production. – *International Journal of Remote Sensing*, 7/11: pp. 1395–1416. doi: 10.1080/01431168608948944.

Ultralytics (2023a): Ultralytics YOLOv5. – *Ultralytics YOLO Documentation*. – Retrieved August 25, 2025, from https://docs.ultralytics.com/de/models/yolov5/.

Ultralytics (2023b): Explore Ultralytics YOLOv8. – *Ultralytics YOLO Documentation*. – Retrieved August 26, 2025, from https://docs.ultralytics.com/models/yolov8/.

Ultralytics (2023c): Pose Estimation. – *Ultralytics YOLO Documentation*. – Retrieved August 26, 2025, from https://docs.ultralytics.com/tasks/pose/.

Ultralytics (2023d): Model Training with Ultralytics YOLO. – *Ultralytics YOLO Documentation*. – Retrieved April 9, 2025, from https://docs.ultralytics.com/modes/train/.

Ultralytics (2023e): Ultralytics YOLO Hyperparameter Tuning Guide. – *Ultralytics YOLO Documentation*. – Retrieved April 9, 2025, from https://docs.ultralytics.com/guides/hyperparameter-tuning/.

Ultralytics (2023f): Test-Time Augmentation (TTA). – *Ultralytics YOLO Documentation*. – Retrieved April 9, 2025, from https://docs.ultralytics.com/yolov5/tutorials/test_time_augmentation/.

Ultralytics (2024a): Oriented Bounding Boxes Object Detection. – *Ultralytics YOLO Documentation*. – Retrieved August 25, 2025, from https://docs.ultralytics.com/de/models/yolov5/.

Ultralytics (2024b): YOLOv9: A Leap Forward in Object Detection Technology. – *Ultralytics YOLO Documentation*. – Retrieved August 26, 2025, from https://docs.ultralytics.com/models/yolov9/.

Ultralytics (2024c): Ultralytics YOLO11. – *Ultralytics YOLO Documentation*. – Retrieved August 26, 2025, from https://docs.ultralytics.com/models/yolo11/.

Ultralytics (2025a): YOLO12: Attention-Centric Object Detection. – *Ultralytics YOLO Documentation*. – Retrieved August 25, 2025, from https://docs.ultralytics.com/models/yolo12/.

Ultralytics (2025b): Reference for split.py. – *Ultralytics YOLO Documentation*. – Retrieved September 4, 2025, from https://docs.ultralytics.com/reference/data/split/.

Ultralytics (undated): Batch Size. – *Ultralytics Glossary*. – Retrieved September 5, 2025, from https://www.ultralytics.com/glossary/batch-size.

Ultralytics (undated): Epoch. – *Ultralytics Glossary*. – Retrieved September 5, 2025, from https://www.ultralytics.com/glossary/epoch.

Ultralytics (undated): Adam Optimizer. – *Ultralytics Glossary*. – Retrieved September 6, 2025, from https://www.ultralytics.com/glossary/adam-optimizer.

VASWANI, Ashish, SHAZEER, Noam, PARMAR, Niki, USZKOREIT, Jakob, JONES, Llion, GOMEZ, Aidan N, KAISER, Łukasz & POLOSUKHIN, Illia (2017): Attention is All you Need. – pp. 6000–6010, *NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems*, Long Beach, California, USA.

WANG, Chien-Yao, BOCHKOVSKIY, Alexey & LIAO, Hong-Yuan Mark (2023): YOLOv7: Trainable Bag-of-Freebies Sets New State-of-the-Art for Real-Time Object Detectors. – pp. 7464–7475, *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* Vancouver, BC, Canada (IEEE). doi: 10.1109/CVPR52729.2023.00721.

WANG, Mei & DENG, Weihong (2018): Deep Visual Domain Adaptation: A Survey. – *Neurocomputing*, 312: pp. 135–153. doi: 10.1016/j.neucom.2018.05.083.

ZÄPERNICK, Marion (2006): Geballtes Wissen rund um Weihnachtsbäume. – *LWF aktuell*, 55: pp. 12–13.

Zentrale Kompetenzstelle für Geoinformation (2025): Verwaltungsgrenzen Hessen. – Retrieved September 17, 2025, from https://www.geoportal.hessen.de/mapbender/php/mod_showMetadata.php?resource=layer&layout=tabs&redirectToMetadataUrl=1&id=59669.

ZHAO, Xia, WANG, Limin, ZHANG, Yufei, HAN, Xuming, DEVECI, Muhammet & PARMAR, Milan (2024): A review of convolutional neural networks in computer vision. – *Artificial Intelligence Review*, 57/4. doi: 10.1007/s10462-024-10721-6.

8 Annex

I Christmas tree cultivation background data

I.I Production data of a Christmas tree cultivation cycle (MESSERSCHMIDT 2024: 15-17)

Kennwerte	Einheit	Kulturjahr									
		1	2	3	4	5	6	7	8	9	10
Bestand											
Anfangsbestand	Stk.	6.000	6.000	6.000	6.000	6.000	6.000	5.400	4.600	3.300	1.800
Verkaufte Bäume	Stk.	0	0	0	0	0	500	700	1.200	1.200	600
Verluste	Stk.	0	400	0	0	0	100	100	100	300	1.200
Nachpflanzung	Stk.	0	400	0	0	0	0	0	0	0	0
Endbestand	Stk.	6.000	6.000	6.000	6.000	6.000	5.400	4.600	3.300	1.800	0
Verkaufsmenge											
Bäume, geschlagen	Stk.						500	700	1.200	1.200	600
Größensortierung Bäume geschlagen											
100-125 cm	%						40				
125-150 cm	%						60	20			
150-175 cm	%							50	30	20	
175-200 cm	%							30	50	40	60
200-225 cm	%								10	30	30
225-250 cm	%								10	10	10

Leistungs-/Kostenart	Kulturjahr									
	1	2	3	4	5	6	7	8	9	10
	€/ha									
Bäume, geschlagen	0	0	0	0	0	3.118	7.149	15.322	16.477	8.516
Bäume, im Topf	0	0	0	0	0	0	0	0	0	0
Schnittgrün, im Bund	0	0	0	0	0	0	0	0	0	0
Summe Leistungen	**0**	**0**	**0**	**0**	**0**	**3.118**	**7.149**	**15.322**	**16.477**	**8.516**
Kulturanlage	2.040	136	0	0	0	0	0	0	0	0
Schnitt/Pflege	0	0	0	0	0	0	0	0	0	0
Düngemittel	81	106	187	327	327	327	327	327	327	46
Pflanzenschutzmittel	225	225	225	325	546	450	450	450	450	100
Ernte	0	0	0	0	0	183	257	440	440	220
Sonstiges Material	1.458	0	0	0	0	0	0	0	0	0
Pacht	700	700	700	700	700	700	700	700	700	700
Zinskosten	135	35	33	41	47	50	52	58	58	32
Summe Direktkosten	**4.639**	**1.202**	**1.145**	**1.393**	**1.620**	**1.710**	**1.786**	**1.975**	**1.975**	**1.098**
Lohnarbeit und Dienstle	15	0	15	0	15	0	15	0	15	0
Variable Maschinenkost	375	99	125	122	204	317	379	525	517	825
Zinskosten	12	3	4	4	7	10	12	16	16	25
Summe variabler Arbei	**402**	**102**	**144**	**126**	**226**	**327**	**406**	**541**	**548**	**850**
Lohnkosten	1.234	753	709	1.850	1.528	1.883	2.065	2.478	2.522	1.761
Fix Maschinenkosten	419	109	242	137	228	242	281	368	369	682
Summe fixe Arbeitserled	1.653	862	951	1.987	1.756	2.125	2.346	2.846	2.891	2.443
Summe Einzelkosten	**6.694**	**2.166**	**2.240**	**3.506**	**3.602**	**4.162**	**4.538**	**5.362**	**5.414**	**4.391**

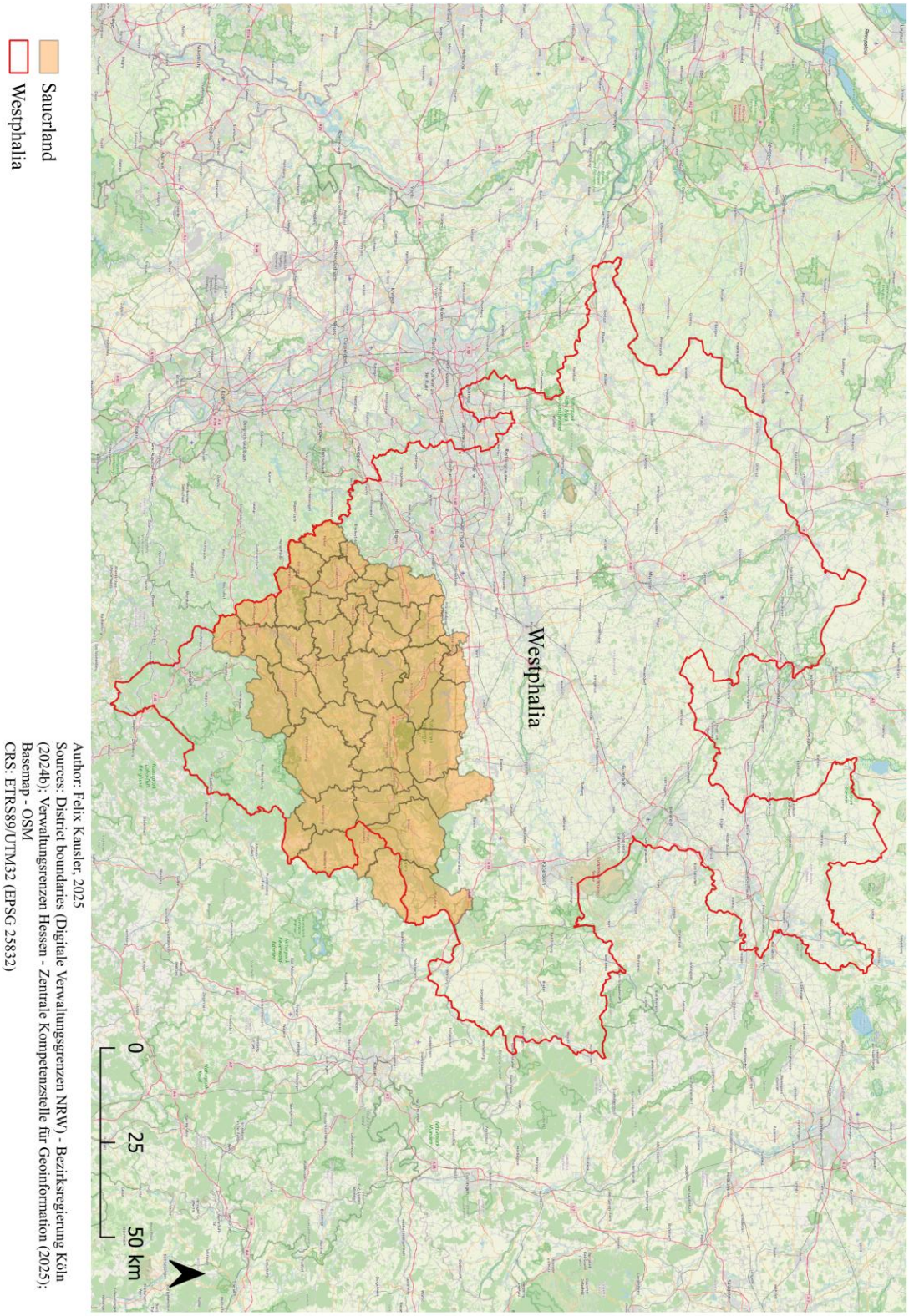
Leistungs-/Kostenart	Kulturjahr 1	1-2	1-3	1-4	1-5	1-6	1-7	1-8	1-9	1-10	Annuität €/ha
	Betrag in €/(ha - Zeitraum)										
Leistung	**0**	**0**	**0**	**0**	**0**	**3.118**	**10.361**	**25.994**	**43.251**	**53.065**	**4.629**
Direktkosten	4.639	5.980	7.304	8.916	10.803	12.837	15.008	17.433	19.931	21.627	1.887
Direktkostenfreie Leistung	**-4.639**	**-5.980**	**-7.304**	**-8.916**	**-10.803**	**-9.719**	**-4.647**	**8.561**	**23.320**	**31.438**	**2.742**
Variable Arbeitserledigungskosten	401	515	674	820	1.070	1.428	1.876	2.473	3.094	4.037	352
Deckungsbeitrag	**-5.040**	**-6.495**	**-7.978**	**-9.736**	**-11.873**	**-11.147**	**-6.523**	**6.088**	**20.226**	**27.401**	**2.390**
Fixe Arbeitserledigungskosten	1.653	2.565	3.593	5.688	7.614	9.967	12.612	15.836	19.202	22.221	1.938
Einzelkostenfreie Leistung	**-6.693**	**-9.060**	**-11.571**	**-15.424**	**-19.487**	**-21.114**	**-19.135**	**-9.748**	**1.024**	**5.180**	**452**

I.II Legal classification of Christmas tree plantations based on § 1 LFoG (STANGE 2016: 20; translated)

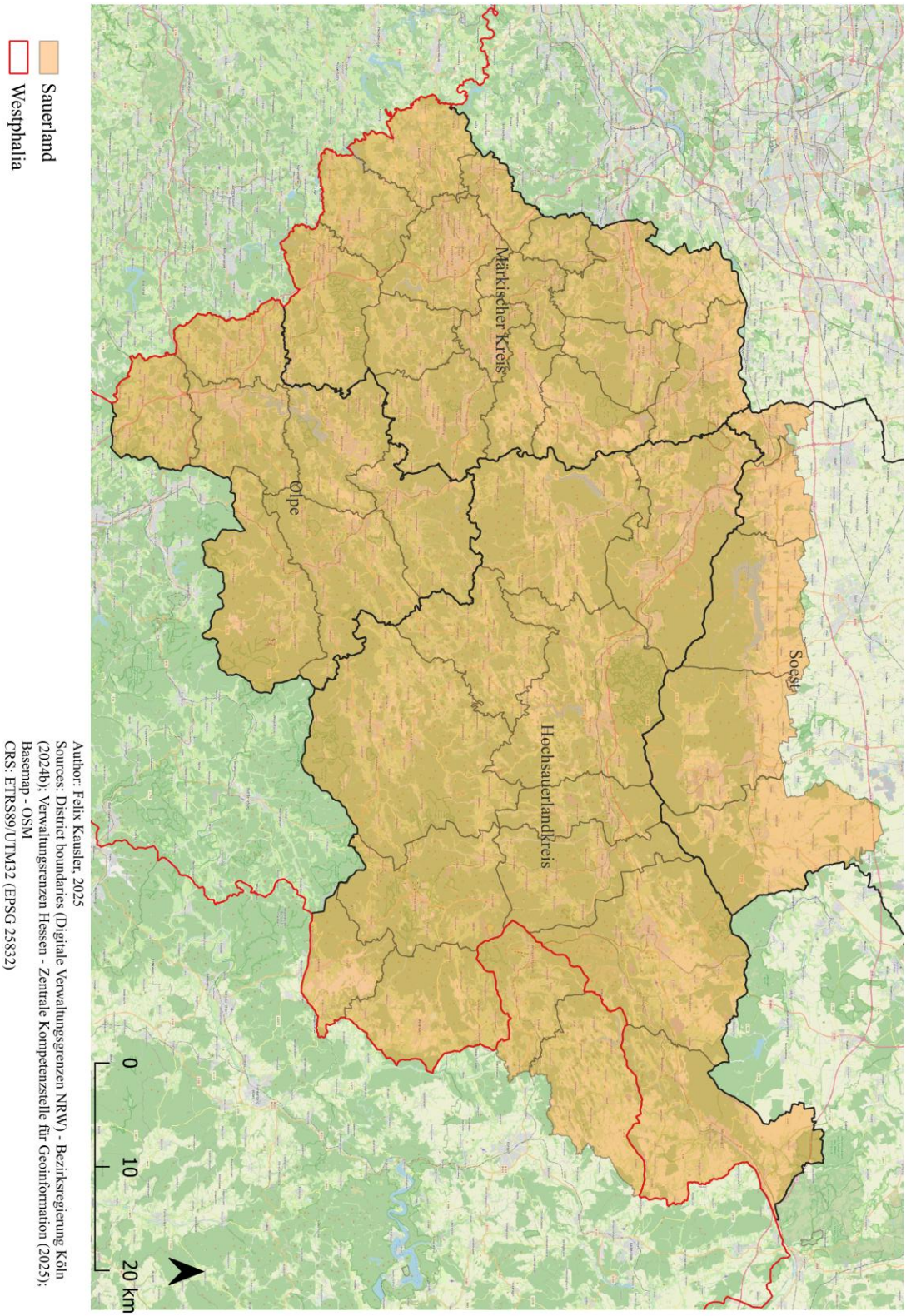
CTPs	Forest by law	Explanation	Owner/operator	Contract
up to 2 ha	yes	• not in close proximation to other CTPs	• **owner** cultivates himself, area is not subleased • reported to the forest authority	no
under power lines	yes		• reported to the forest authority	no
until 31/12/2028, that are forests by law	yes	• continued management as CTP, • from 2029 only as high forest, possibly with secondary use, i.e. with management restrictions • conversion to high forest (*let trees grow through*)	• decision needs to be made: what should happen after 2028?	possible
with continued management **after 2028**	yes		• conclude contract with forest authority **by 12.12.2016**	yes
without continued management **after 2028**	yes	• conversion to high forest	• report measures to the forest authority, which may refuse them or issue ancillary provisions! Proper forest management §1b *LFoG*	no
with continued management without contract **from 1/1/2029**	yes	• area remains forest by law as long as conversion is not approved	• requires **conversion approval** from the forest authority in accordance with § 39 *LFoG*	no
planted **in 1980 – 1987**, areas in the open landscape with afforestation assessment as CTP	no	• were established/authorised during this period as initial afforestation for use as CTP • are forest if they grow through • were/are exempt from forestry law	• continued cultivation is permitted	no
established **after 2013**	yes	• is an **unauthorised conversion**	• Application for conversion pursuant to § 39 *LFoG* must be submitted	no
areas in the open landscape from 1987	no	• established with and without approval • responsibility of the Lower landscape authority	• responsibility of the Lower landscape authority	no

II Maps and Datasets

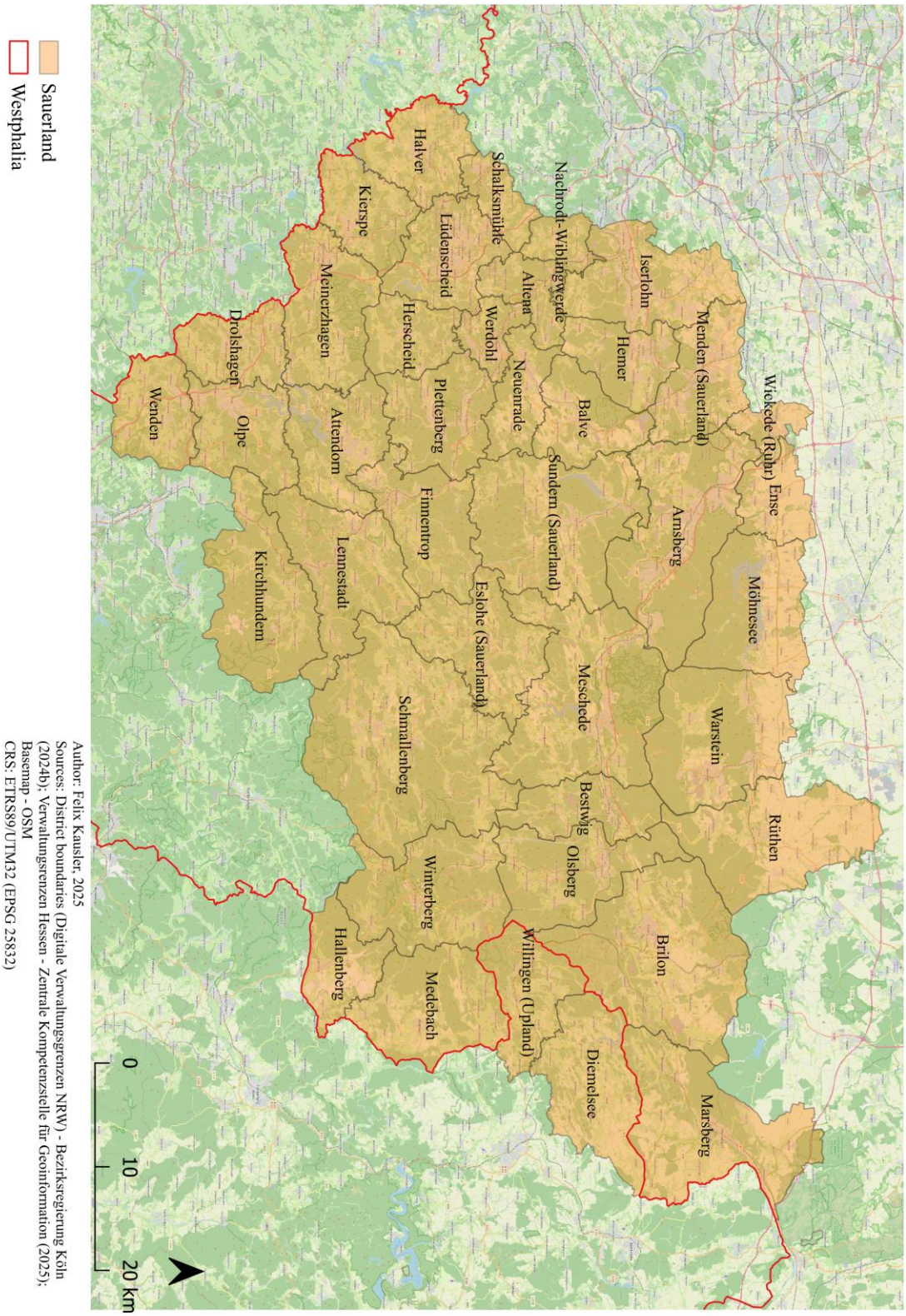
II.I Westphalia and the Sauerland region



II.II Districts of the Sauerland (NRW)



II.III Municipalities of the Sauerland



II.IV Landcover in Bestwig-Heringhausen, NRW 2013 (MAURMANN 2015b)

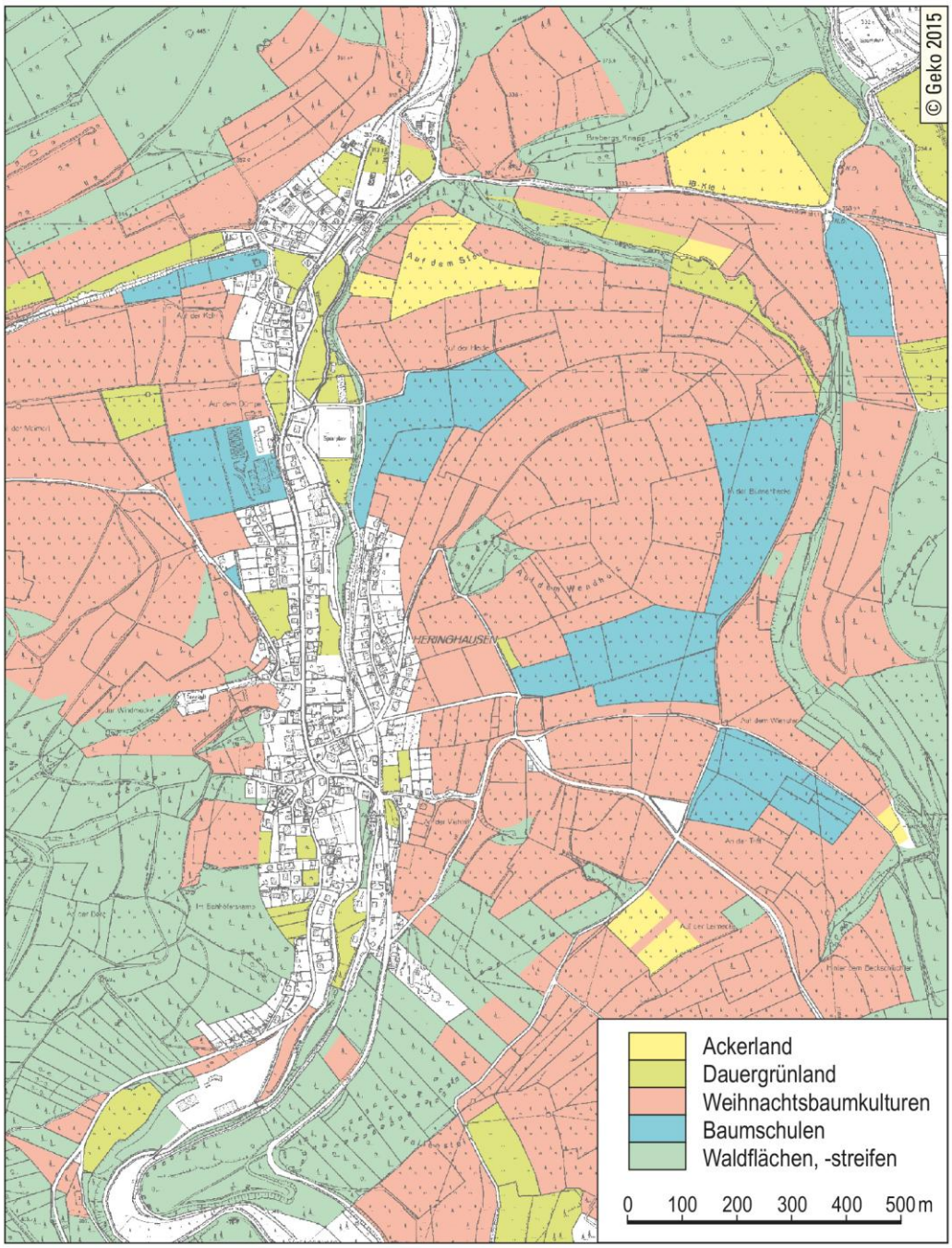
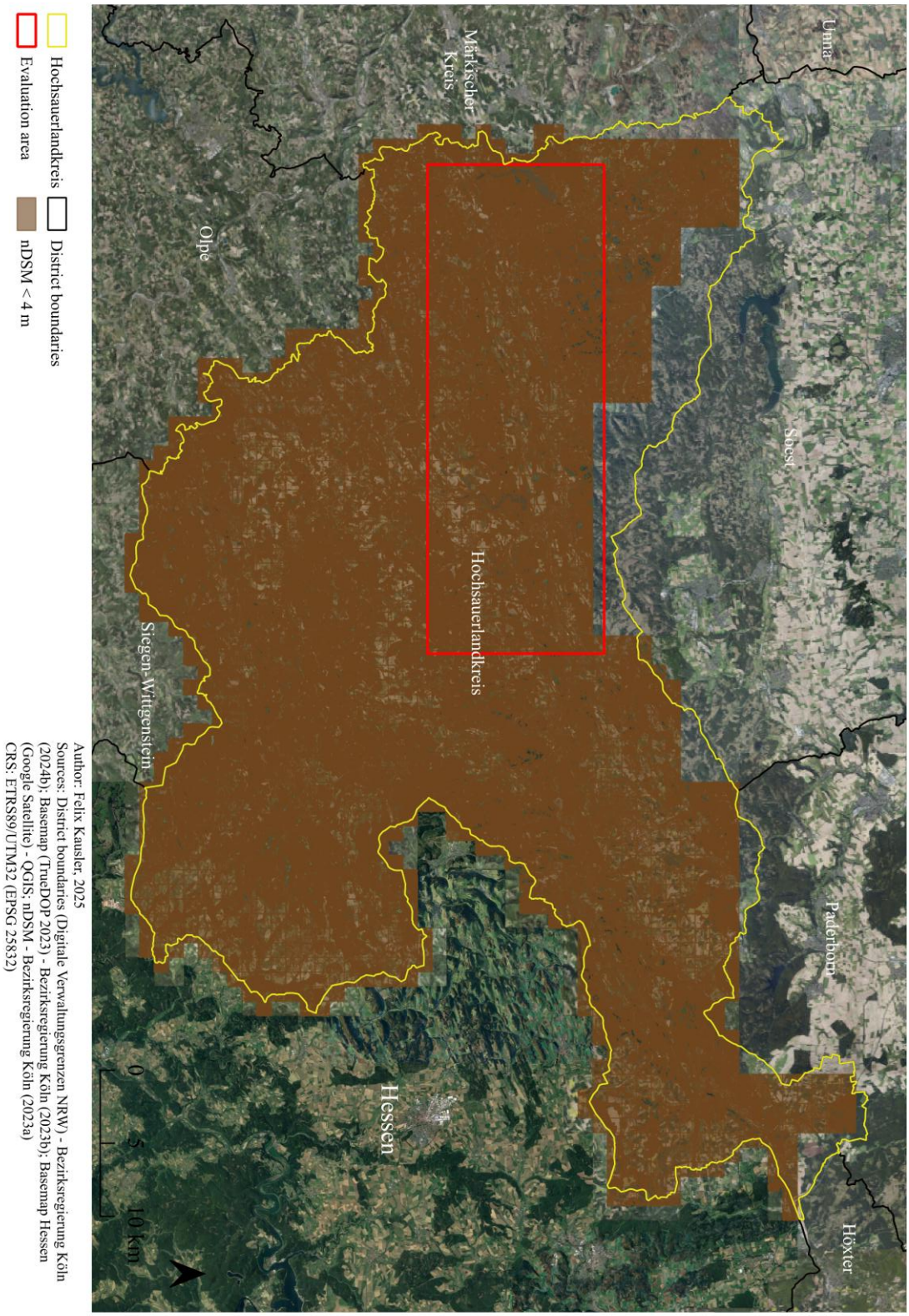
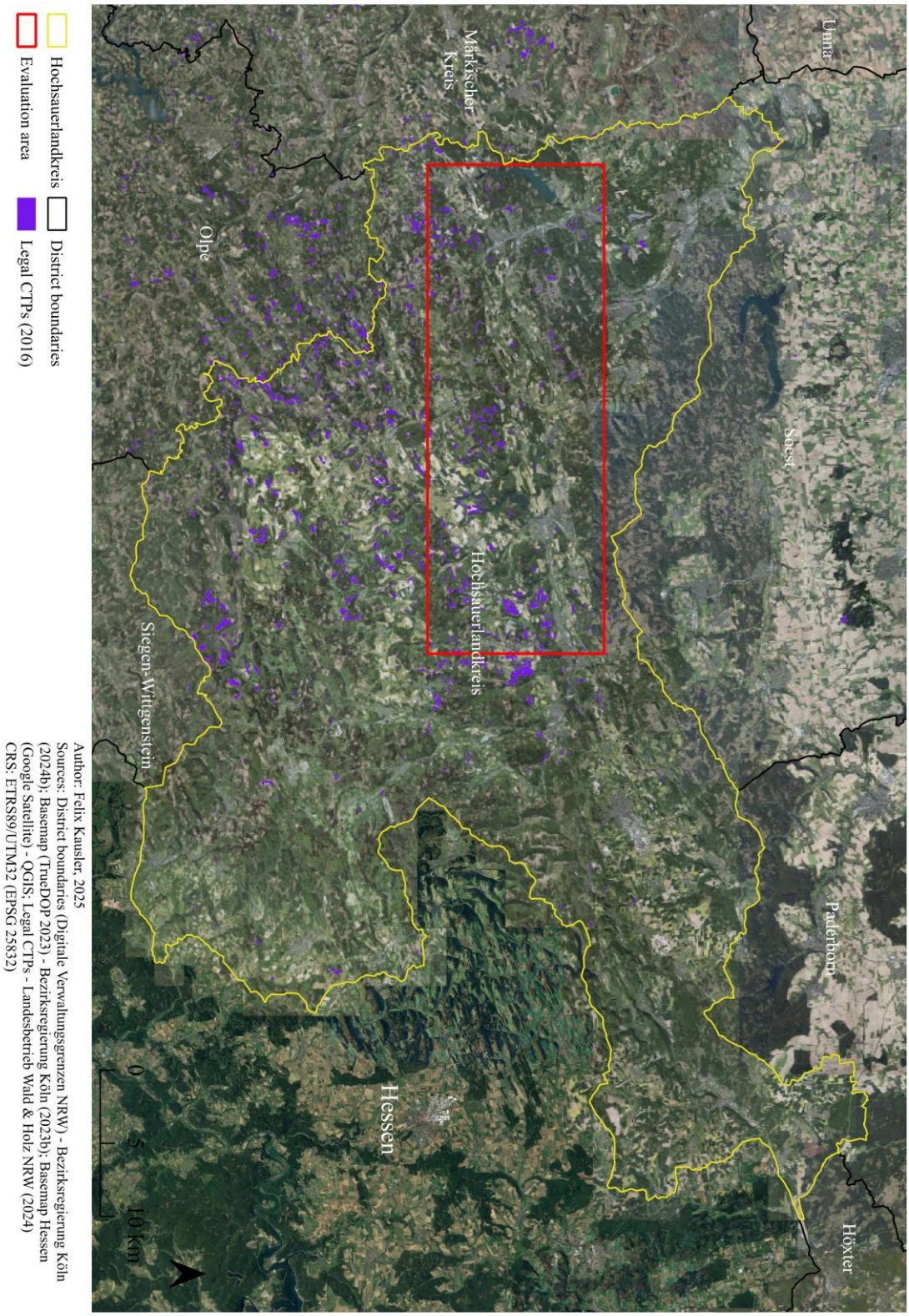


Abb. 1: Bodennutzung in Bestwig-Heringhausen 2013 (Quelle: eigene Kartierung, Kartengrundlage: Geobasisdaten der Kommunen und des Landes NRW, © Geobasis NRW)

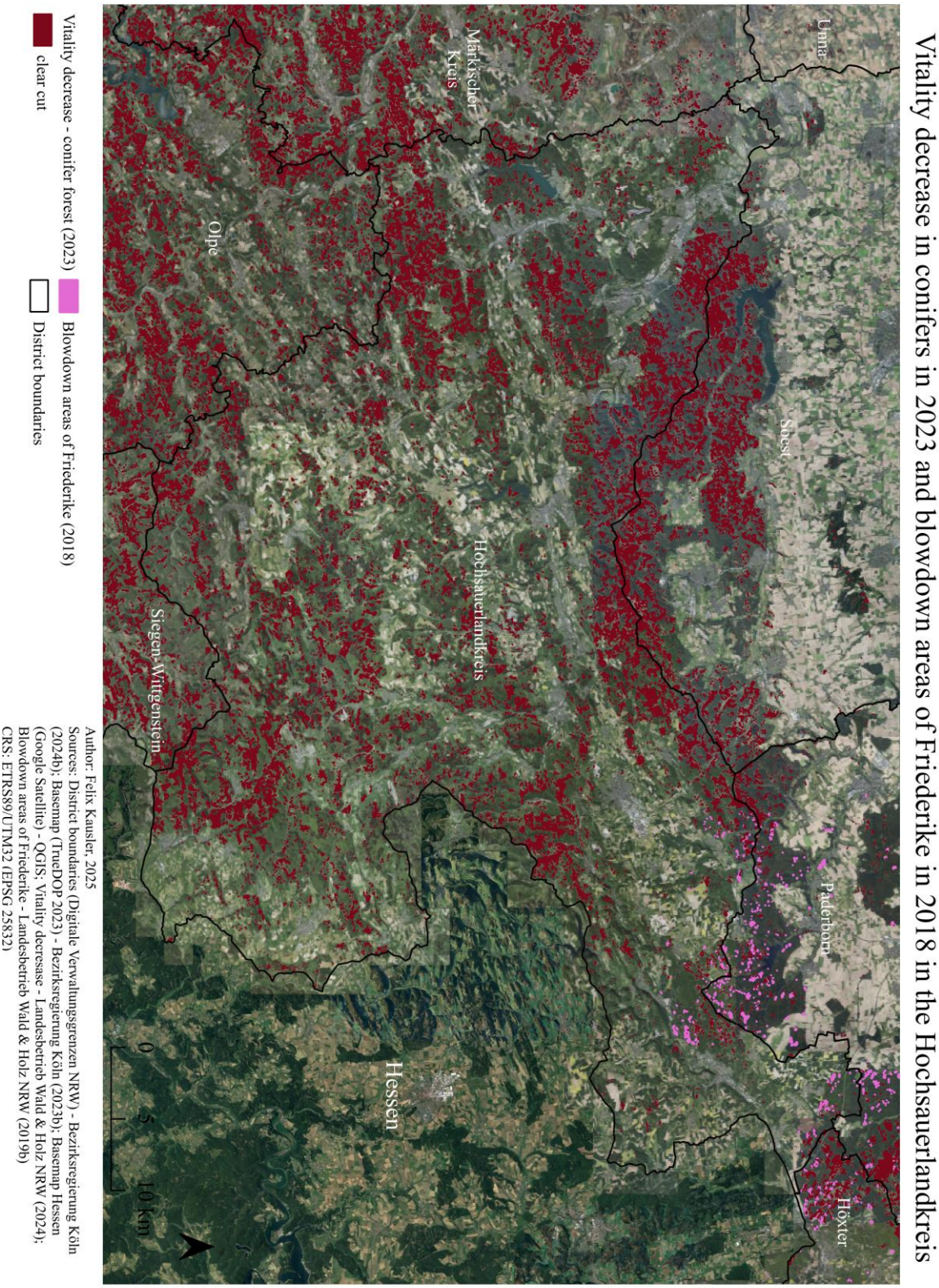
II.V nDSM in the Hochsauerlandkreis



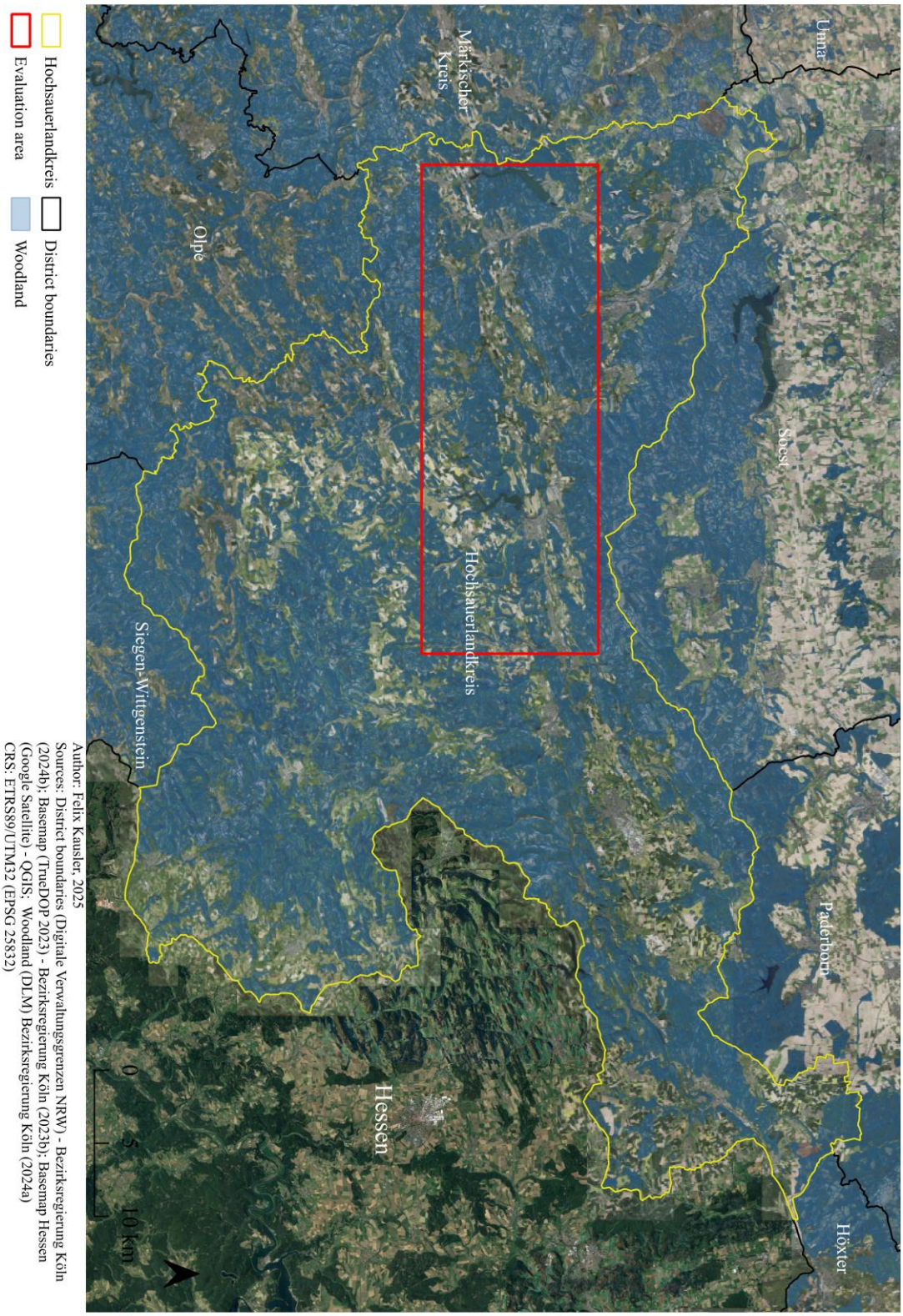
II.VI Legal Christmas tree plantations in the Hochsauerlandkreis and neighbouring areas



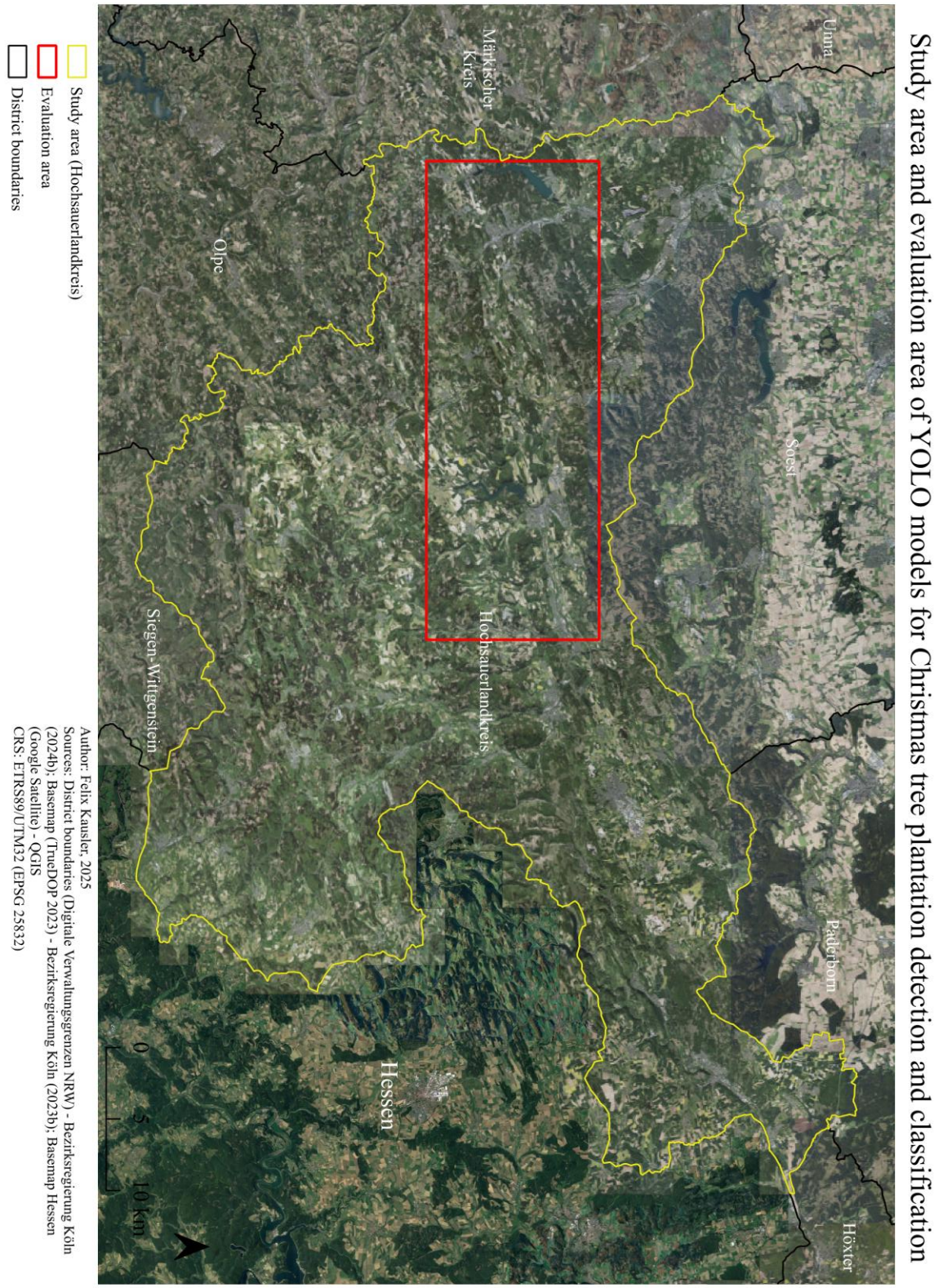
II.VII Damaged areas in the Hochsauerlandkreis and neighbouring areas



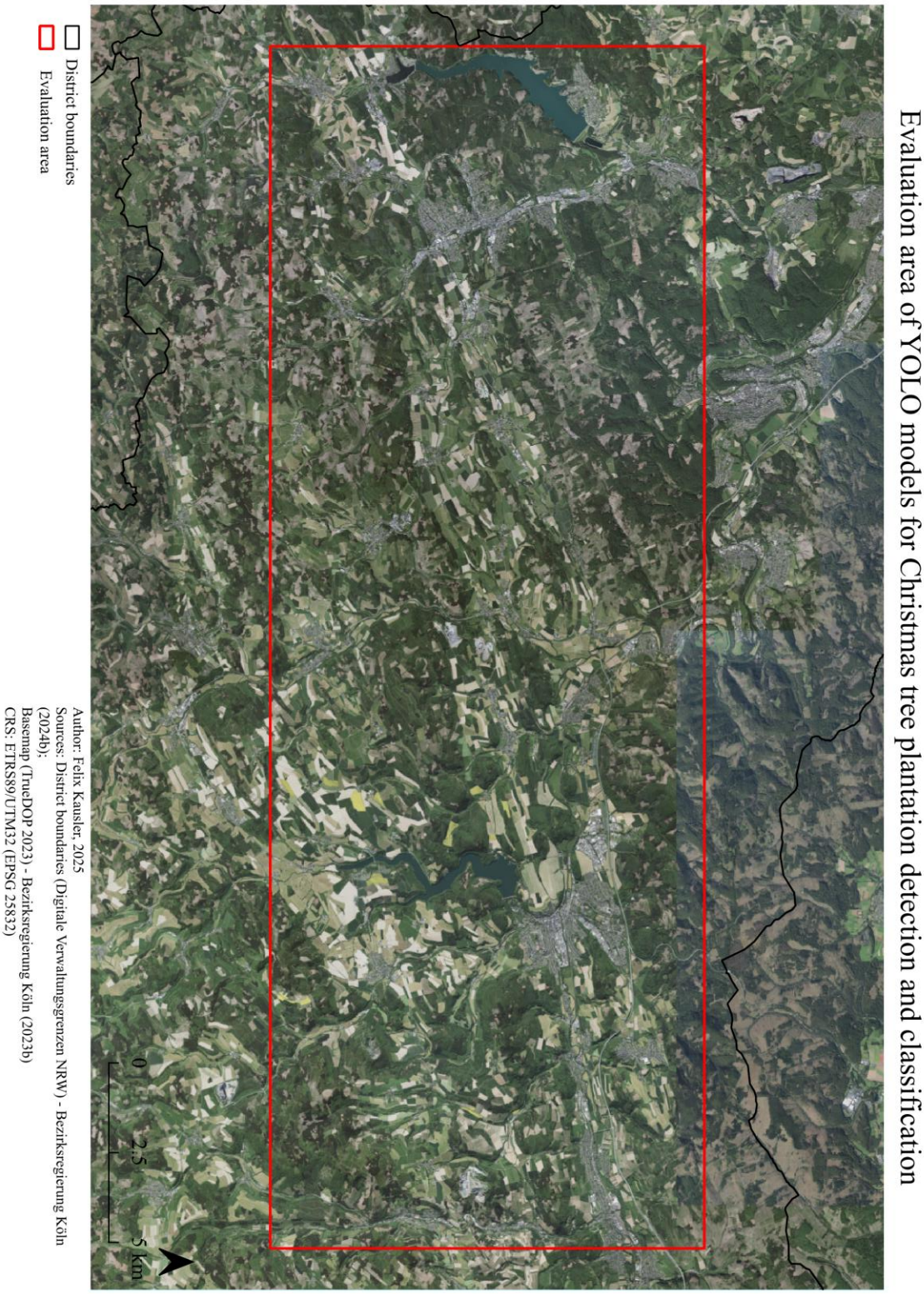
II.VIII Official woodland areas in the Hochsauerlandkreis and neighbouring areas



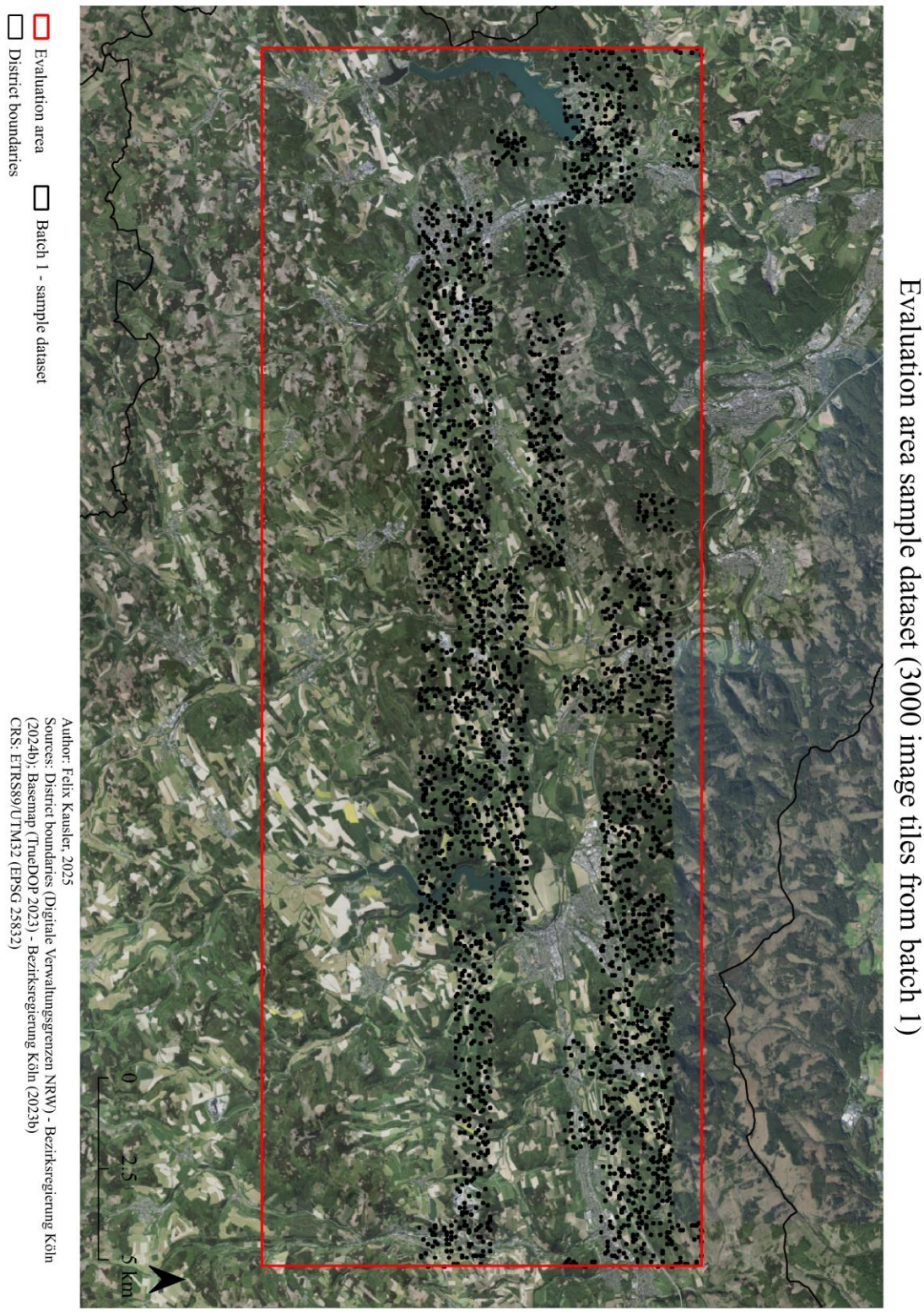
II.IX Study Area



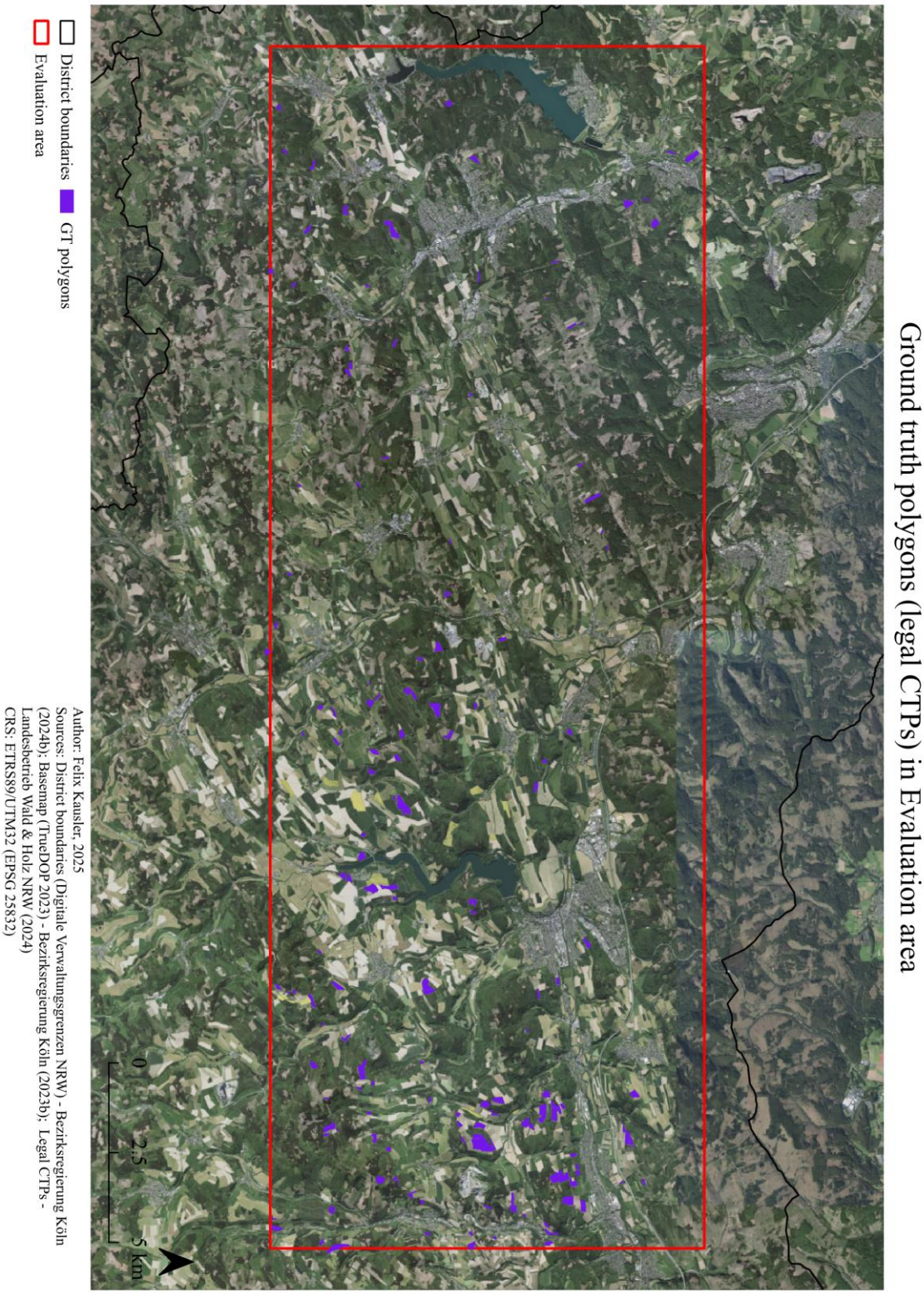
II.X Evaluation area



II.XI Evaluation area sample dataset

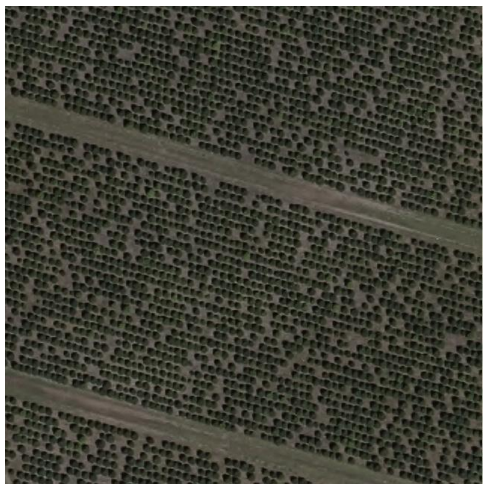


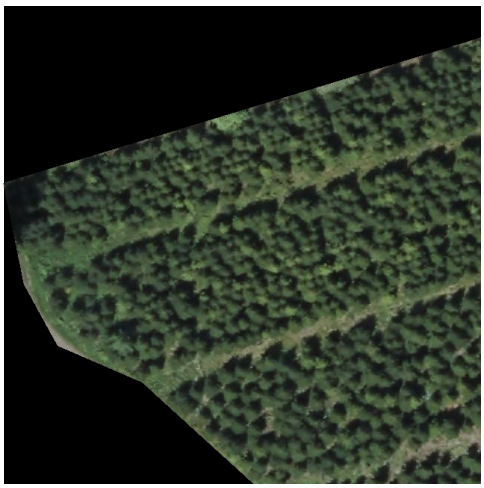
II.XII Ground truth polygons in Evaluation area



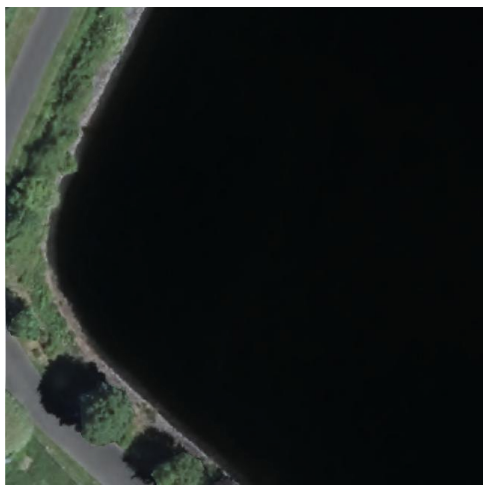
II.XIII Training data examples

II.XIII.I Positive training data examples from YOLO

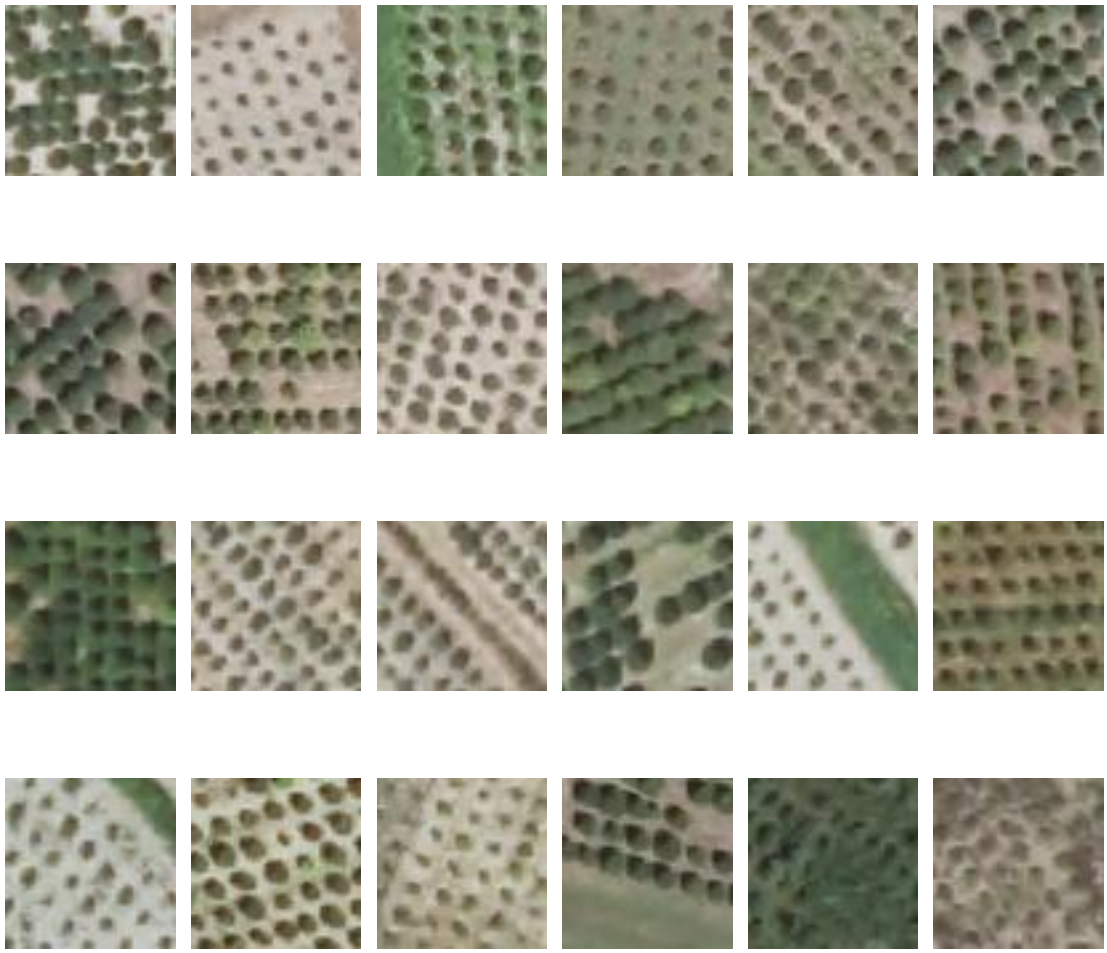




II.XIII.II Negative training data examples from YOLO



II.XIII.III Positive training data examples from RF patch classification (64 × 64)

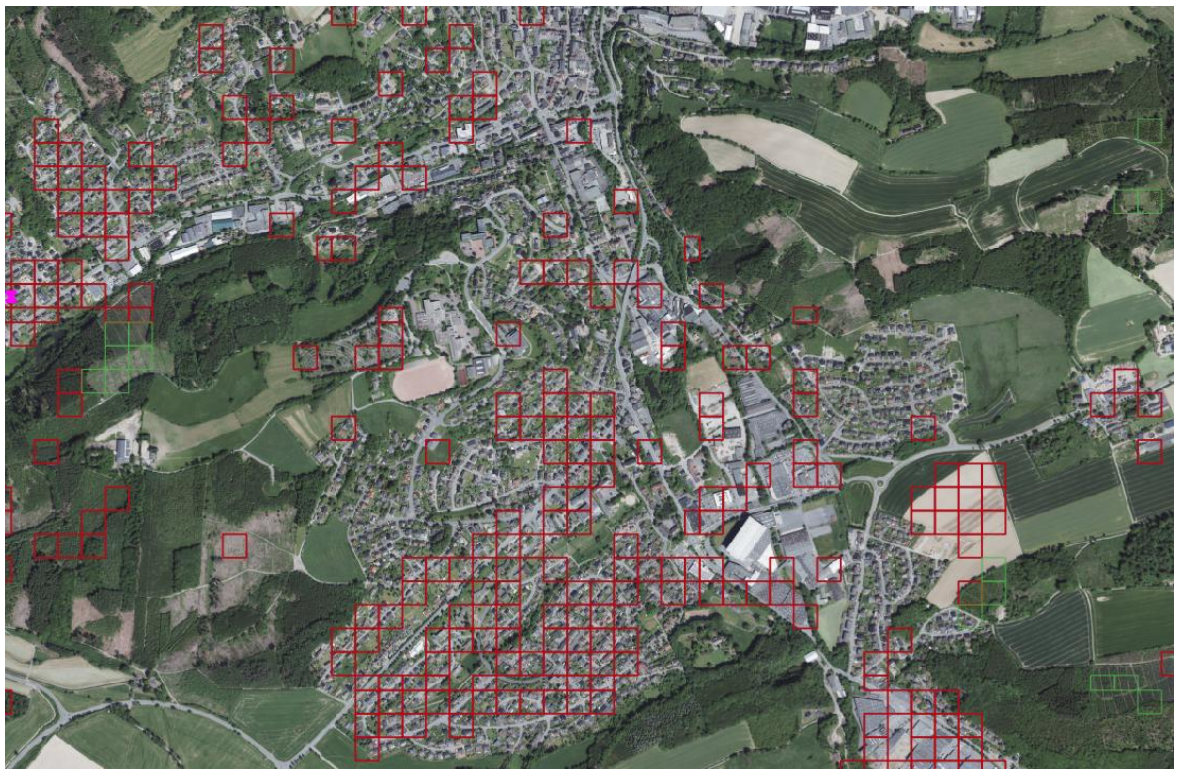


II.XIV Inference data examples

II.XIV.I Inference with detection overlaps

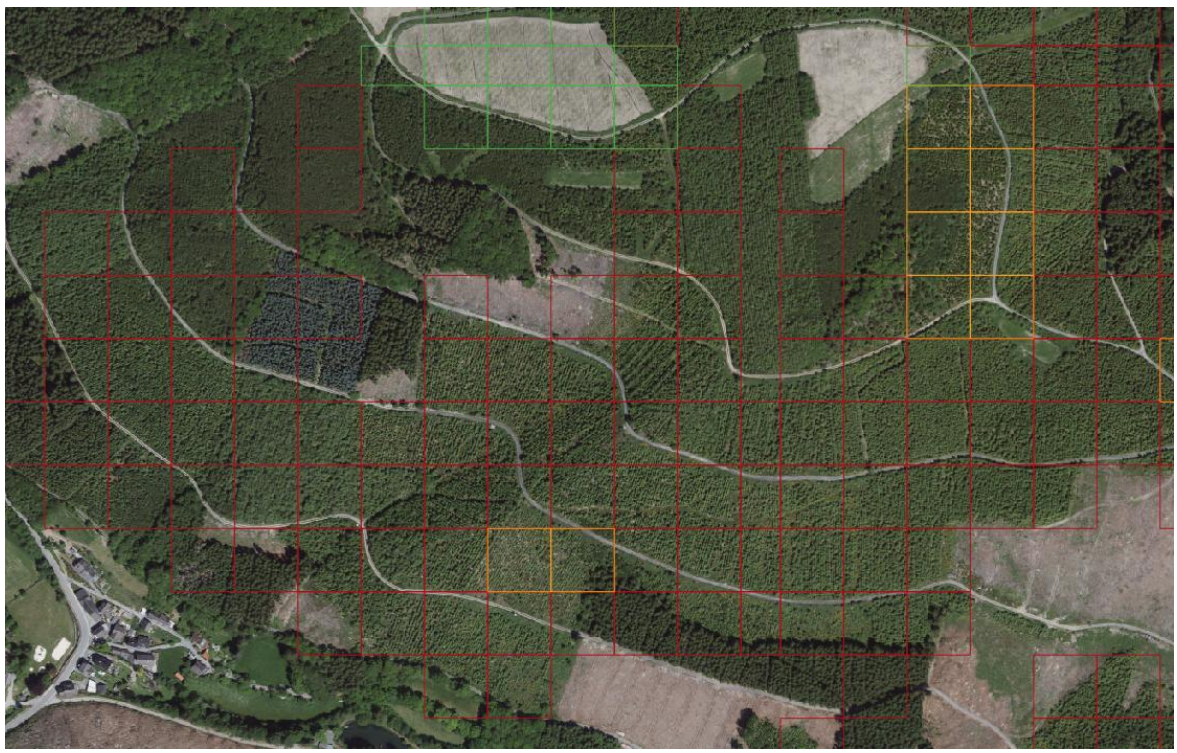
II.XIV.II Potential positive example

II.XIV.III YOLOv8 run False Positives in urban areas



II.XIV.IV YOLOv8 run 04 False Positives in quarry

II.XIV.V YOLOv12 run 08 False Positives in the south of the EA



II.XIV.VI CTP accumulation south of Meschede/west of Heringhausen (with TPs from YOLOv5 run 03)

II.XV Evaluation area sample dataset results

II.XV.I YOLO model results

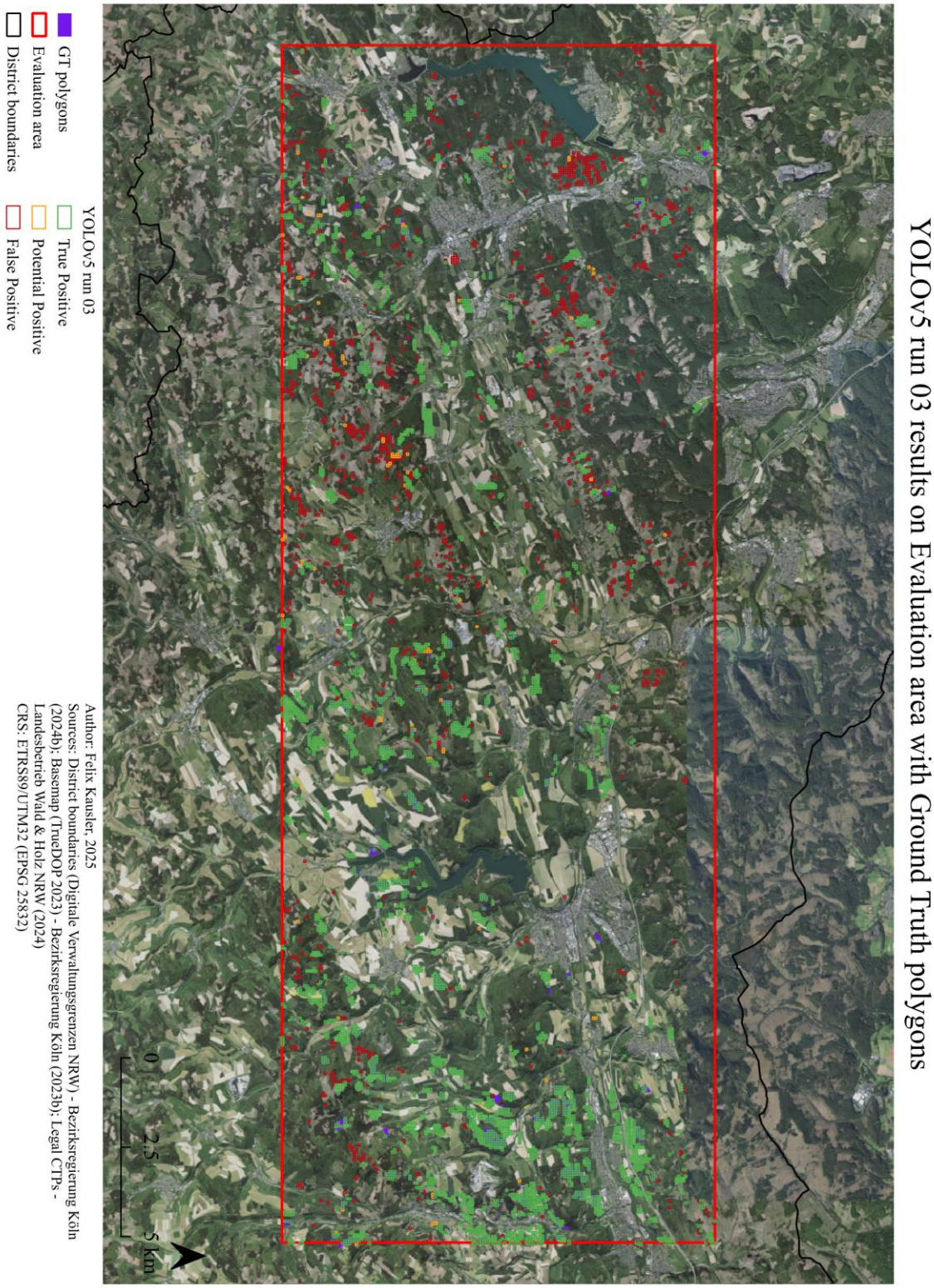
Run	Task	Channels	Training Dataset	Classes	Model Size	Conf_Threshold	HP/TTA	TP (of 95)	Pot. Positives (of 83)	FP	TN	FN	Total predicted Positives
v5_run03_025	**detect**	**RGB**	**v5_unlabelled**	**1**	**s**	**0.25**	**no**	78		**31**	**2856**	**17**	**127**
v8_run02_010	detect	RGBI	v12_labelled	1	m	0.10	no	42	8	7	2890	**53**	57
v8_run02_005	detect	RGBI	v12_labelled	1	m	0.05	no	50	10	11	2884	**45**	71
v8_run03_HP-TTA_010	detect	RGBI	v12_labelled	1	m	0.10	HP & TTA	12	3	39	2863	**83**	54
v8_run04_TDv5_005	detect	RGBI	v5_unlabelled	1	m	0.05	no	73	26	145	2734	22	244
v8_run04_TDv5_010	**detect**	**RGBI**	**v5_unlabelled**	**1**	**m**	**0.10**	**no**	69	**22**	**85**	**2798**	**26**	**176**
v8_run05_TDv5_HP-TTA_010	detect	RGBI	v5_unlabelled	1	m	0.10	HP & TTA	82	55	1524	1326	**13**	1661
v8_run05_TDv5_HP-TTA_020	detect	RGBI	v5_unlabelled	1	m	0.20	HP & TTA	74	44	1218	1643	**21**	1336
v8_run06_TDv5_HP_010	detect	RGBI	v5_unlabelled	1	m	0.10	HP	80	64	1656	1185	**15**	1800
v8_run06_TDv5_HP_020	detect	RGBI	v5_unlabelled	1	m	0.20	HP	73	51	1321	1533	**22**	1445
v8_run07_cls_TDv5_HP_050	*classify*	RGBI	v5_unlabelled	1	m	0.50	HP	53	17	116	2772	**42**	186
Run	**Task**	**Channels**	**Training Dataset**	**Classes**	**Model Size**	**Conf_Threshold**	**HP/TTA**	**TP (of 95)**	**Pot. Positives (of 83)**	**FP**	**TN**	**FN**	**Total predicted Positives**
v12_run02_005	detect	RGB	v12_labelled	1	m	0.05	no	55	26	193	2686	40	274
v12_run03_2cl_005	detect	RGB	v12_labelled	2	m	0.05	no	52	13	39	2853	43	104
v12_run04_2cl_TTA_005	detect	RGB	v12_labelled	2	m	0.05	TTA	60	16	61	2828	35	137
v12_run05_TDv5_010	detect	RGB	v5_unlabelled	1	m	0.10	no	70	24	273	2608	25	367
v12_run05_TDv5_020	detect	RGB	v5_unlabelled	1	m	0.20	no	66	17	166	2722	29	249
v12_run06_TDv5_HP-TTA_010	detect	RGB	v5_unlabelled	1	m	0.10	HP & TTA	76	45	751	2109	19	872
v12_run07_TDv5_HP_010	detect	RGB	v5_unlabelled	1	m	0.10	**HP**	75	37	380	2488	20	492
v12_run07_TDv5_HP_020	detect	RGB	v5_unlabelled	1	m	0.20	**HP**	67	32	280	2593	28	379
v12_run08_cls_025	***classify***	**RGB**	**v12_unlabelled**	**1**	**m**	**0.25**	**no**	**74**	**29**	**66**	**2810**	**21**	**169**
v12_run08_cls_010	*classify*	RGB	v12_unlabelled	1	m	0.10	no	78	42	183	2680	17	303
v12_run09_cls_TDv5_010	*classify*	RGB	v5_unlabelled	1	m	0.10	no	79	28	154	2723	16	261

II.XV.II Random Forest model results

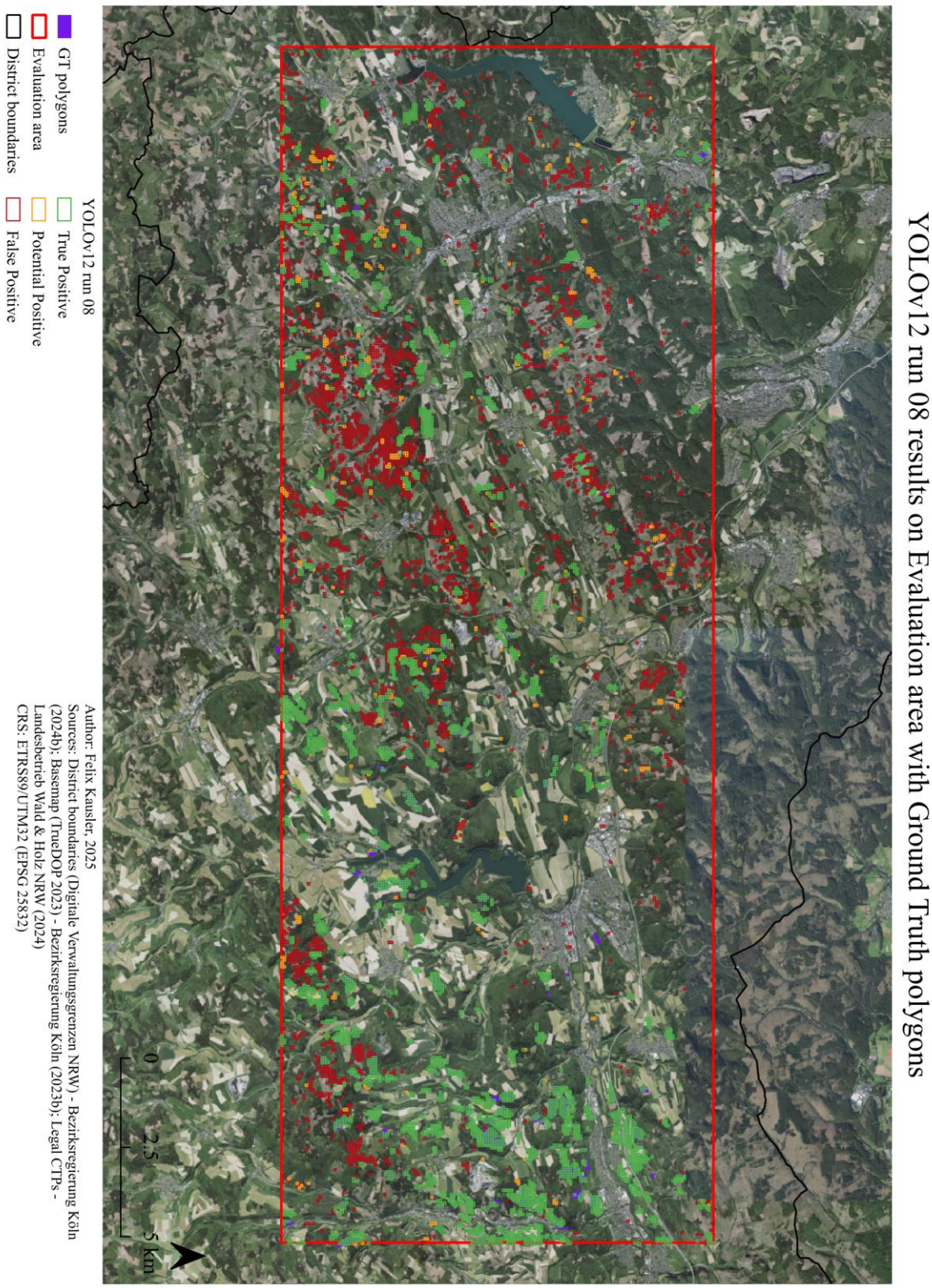
Run	Task	Channels	Training Dataset	Classes	Estimators	Positive_Threshold	HP	TP (of 95)	Pot. Positives (of 83)	FP	TN	FN	Total predicted Positives	Precision	Recall	F1
RF_1class_64_RGBI_100_1	classify	4	v12_64-patches	1	100	1	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_64_RGBI_100_10	classify	4	v12_64-patches	1	100	10	no	91	82	2814	9	4	2987	0.0319	0.9579	0.0618
RF_1class_64_RGBI_100_20	classify	4	v12_64-patches	1	100	20	no	87	81	2801	23	8	2969	0.0306	0.9158	0.0589
RF_1class_NDVI_64_100_1	classify	5	v12_64-patches	1	100	1	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_NDVI_64_100_10	classify	5	v12_64-patches	1	100	10	no	90	83	2814	8	5	2987	0.0316	0.9474	0.0613
RF_1class_NDVI_64_100_20	classify	5	v12_64-patches	1	100	20	no	86	81	2801	23	9	2968	0.0296	0.9053	0.0575
RF_1class_64_RGBI_200_1	classify	4	v12_64-patches	1	200	1	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_64_RGBI_200_10	classify	4	v12_64-patches	1	200	10	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_64_RGBI_200_20	classify	4	v12_64-patches	1	200	20	no	86	81	2801	23	9	2968	0.0296	0.9053	0.0575
RF_1class_64_RGBI_300_1	classify	4	v12_64-patches	1	300	1	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_64_RGBI_300_10	classify	4	v12_64-patches	1	300	10	no	91	83	2814	8	4	2988	0.0316	0.9579	0.0613
RF_1class_64_RGBI_300_20	classify	4	v12_64-patches	1	300	20	no	86	81	2801	23	9	2968	0.0296	0.9053	0.0575
RF_1class_64_RGBI_400_1	classify	4	v12_64-patches	1	400	1	no	95	83	2822	0	0	3000	0.0326	1	0.0631
RF_1class_64_RGBI_400_10	classify	4	v12_64-patches	1	400	10	no	91	83	2814	8	4	2988	0.0316	0.9579	0.0613
RF_1class_64_RGBI_400_20	classify	4	v12_64-patches	1	400	20	no	86	81	2801	23	9	2968	0.0296	0.9053	0.0575
RF_1class_64_RGBI_500_1	classify	4	v12_64-patches	1	500	1	no	94	83	2821	1	1	2998	0.0326	0.9895	0.0631
RF_1class_64_RGBI_500_10	**classify**	**4**	**v12_64-patches**	**1**	**500**	**10**	**no**	**95**	**83**	**2809**	**13**	**0**	**2987**	**0.0327**	**1**	**0.0632**
RF_1class_64_RGBI_500_20	classify	4	v12_64-patches	1	500	20	no	86	81	2800	24	9	2967	0.0297	0.9053	0.0576
Run	**Task**	**Channels**	**Training Dataset**	**Classes**	**Estimators**	**Positive_Threshold**	**HP**	**TP (of 95)**	**Pot. Positives (of 83)**	**FP**	**TN**	**FN**	**Total predicted Positives**	**Precision**	**Recall**	**F1**
RF_2class_64_100_1	classify	4	v12_64-patches	2	100	1	no	94	81	2428	396	1	2603	0.0379	0.9895	0.0729
RF_2class_64_100_10	classify	4	v12_64-patches	2	100	10	no	85	73	1485	1347	10	1643	0.0548	0.8947	0.1030
RF_2class_64_100_20	**classify**	**4**	**v12_64-patches**	**2**	**100**	**20**	**no**	**76**	**60**	**969**	**1876**	**19**	**1105**	**0.0721**	**0.8000**	**0.1315**
RF_2class_NDVI_64_100_1	classify	5	v12_64-patches	2	100	1	no	95	82	2381	442	0	2558	0.0383	1	0.0738
RF_2class_NDVI_64_100_10	classify	5	v12_64-patches	2	100	10	no	81	72	1484	1349	14	1637	0.0520	0.8526	0.0982
RF_2class_NDVI_64_100_20	classify	5	v12_64-patches	2	100	20	no	73	55	952	1898	22	1080	0.0710	0.7684	0.1276
RF_2class_64_RGBI_200_1	classify	4	v12_64-patches	2	200	1	no	94	83	2399	24	1	2576	0.0373	0.9895	0.0709
RF_2class_64_RGBI_200_10	classify	4	v12_64-patches	2	200	10	no	81	73	1478	423	14	1632	0.0523	0.8526	0.0983
RF_2class_64_RGBI_200_20	classify	4	v12_64-patches	2	200	20	no	72	56	995	1354	23	1123	0.0678	0.7579	0.1236
RF_2class_64_RGBI_300_1	classify	4	v12_64-patches	2	300	1	no	94	82	2389	1854	1	2565	0.0378	0.9895	0.0728
RF_2class_64_RGBI_300_10	classify	4	v12_64-patches	2	300	10	no	81	73	1484	434	14	1638	0.0520	0.8526	0.0982
RF_2class_64_RGBI_300_20	classify	4	v12_64-patches	2	300	20	no	71	55	1000	1348	24	1126	0.0663	0.7474	0.1202
RF_2class_64_RGBI_400_1	classify	4	v12_64-patches	2	400	1	no	94	82	2377	1850	1	2553	0.0377	0.9895	0.0726
RF_2class_64_RGBI_400_10	classify	4	v12_64-patches	2	400	10	no	81	73	1485	446	14	1639	0.0521	0.8526	0.0984
RF_2class_64_RGBI_400_20	classify	4	v12_64-patches	2	400	20	no	71	55	1006	1347	24	1132	0.0663	0.7474	0.1202
RF_2class_64_RGBI_500_1	classify	4	v12_64-patches	2	500	1	no	94	82	2375	1844	1	2551	0.0378	0.9895	0.0728
RF_2class_64_RGBI_500_10	classify	4	v12_64-patches	2	500	10	no	82	71	1477	448	13	1630	0.0522	0.8611	0.0997
RF_2class_64_RGBI_500_20	classify	4	v12_64-patches	2	500	20	no	70	57	1015	1357	25	1142	0.0642	0.7368	0.1147
RF_2class_128_100_1	classify	4	v12_128-patches	2	100	1	no	93	83	2815	7	2	2991	0.0324	0.9787	0.0627
RF_2class_128_100_10	classify	4	v12_128-patches	2	100	10	no	74	74	2695	136	21	2843	0.0263	0.7789	0.0507
RF_2class_128_100_20	classify	4	v12_128-patches	2	100	20	no	30	49	1966	890	65	2045	0.0150	0.3158	0.0293
Run	**Task**	**Channels**	**Training Dataset**	**Classes**	**Estimators**	**Positive_Threshold**	**HP**	**TP (of 95)**	**Pot. Positives (of 83)**	**FP**	**TN**	**FN**	**Total predicted Positives**	**Precision**	**Recall**	**F1**
RF_Segmentation_1	classify	5	v12_segments	1	200	1	yes	91	67	933	1905	4	1091	0.0856	0.9263	0.1567
RF_Segmentation_2	classify	5	v12_segments	1	200	2	yes	87	57	451	2397	8	595	0.1581	0.9053	0.2692
RF_Segmentation_3	classify	5	v12_segments	1	200	3	yes	84	48	215	2642	11	347	0.2677	0.8737	0.4099
RF_Segmentation_4	classify	5	v12_segments	1	200	4	yes	82	38	99	2768	13	219	0.4084	0.8211	0.5455
RF_Segmentation_5	classify	5	v12_segments	1	200	5	yes	78	24	49	2832	17	151	0.5522	0.7789	0.6463
RF_Segmentation_6	**classify**	**5**	**v12_segments**	**1**	**200**	**6**	**yes**	**75**	**18**	**12**	**2875**	**20**	**105**	**0.7158**	**0.7158**	**0.7158**
RF_Segmentation_7	classify	5	v12_segments	1	200	7	yes	68	12	10	2883	27	90	0.7750	0.6526	0.7086
RF_Segmentation_8	classify	5	v12_segments	1	200	8	yes	60	11	9	2885	35	80	0.7297	0.5684	0.6391

II.XVI EA inference results

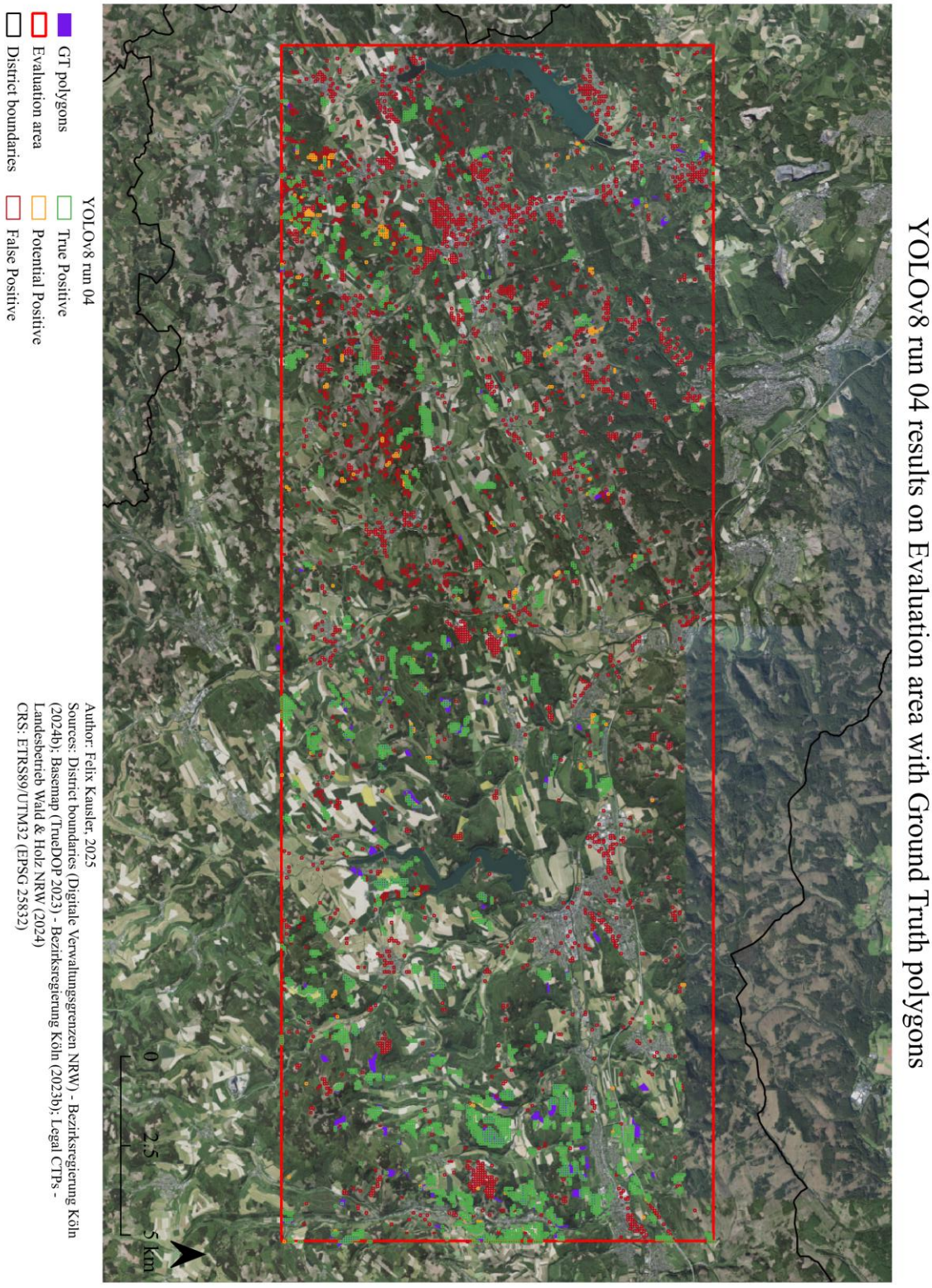
II.XVI.I YOLOv5 EA inference



II.XVI.II YOLOv12 EA inference

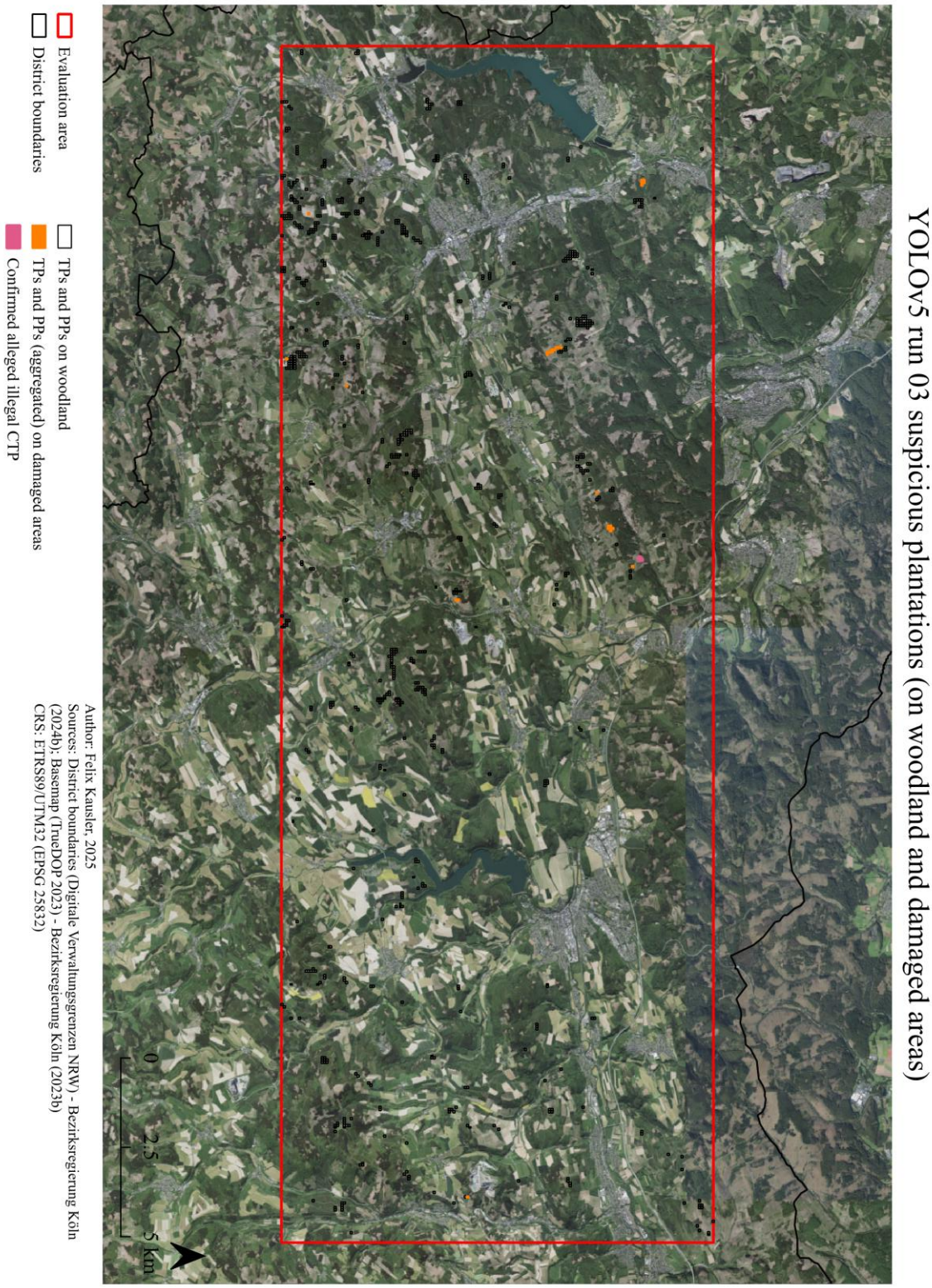


II.XVI.III YOLOv8 EA inference

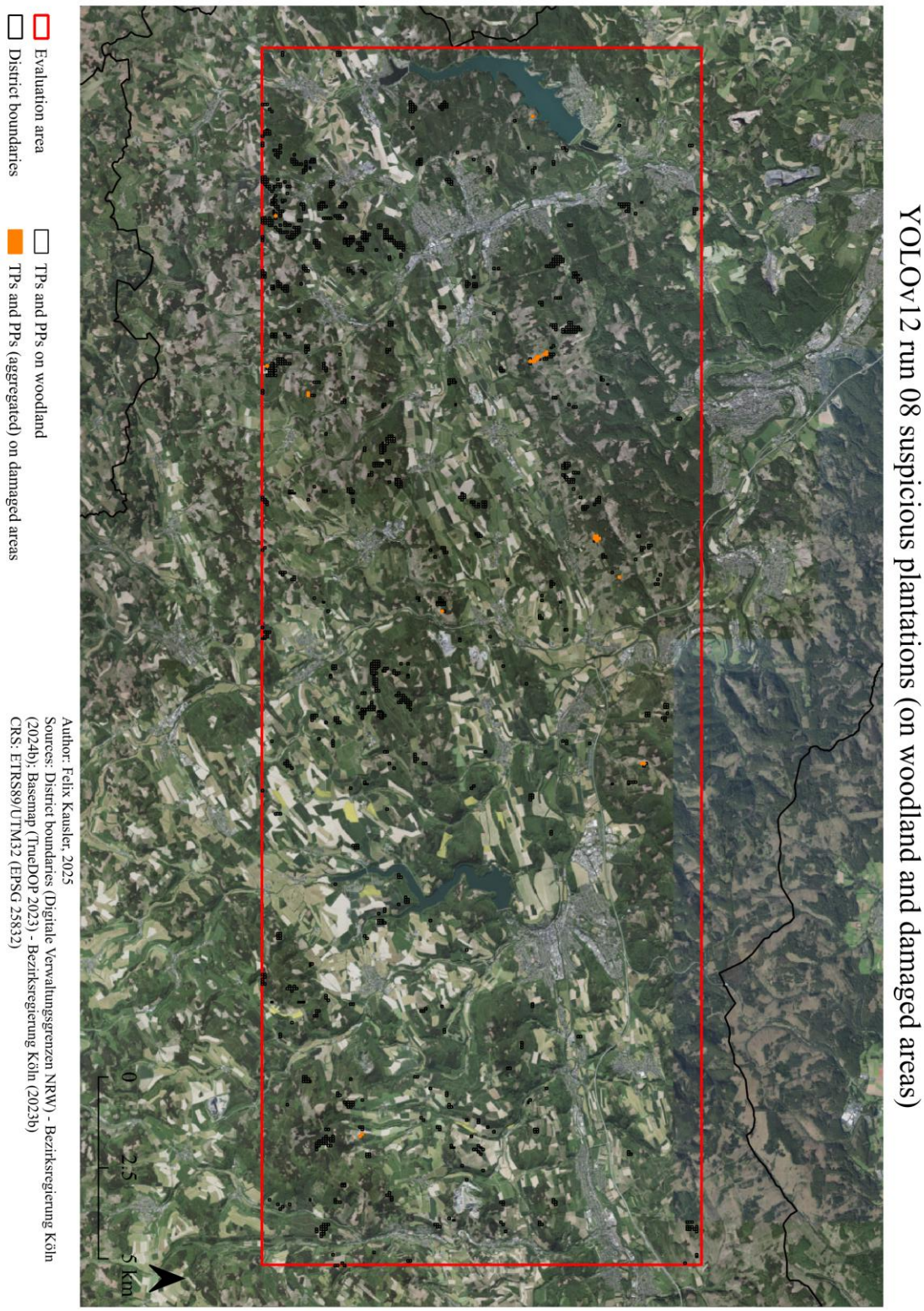


II.XVII Suspicious plantations – woodland and damaged areas

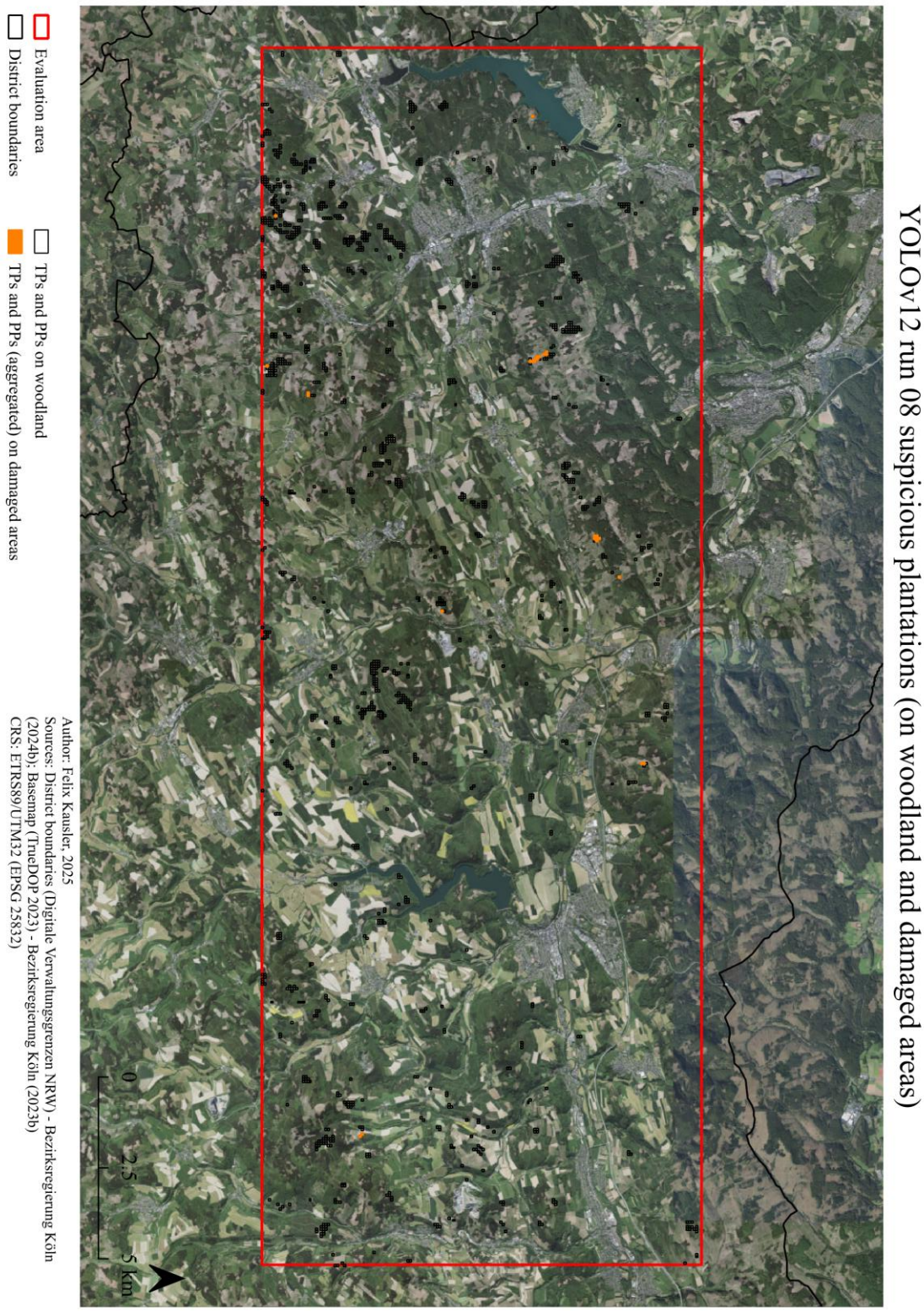
II.XVII.I YOLOv5 results



II.XVII.II YOLOv12 results



II.XVII.III YOLOv8 results



III Python code

The LLM interface Perplexity was used for the conceptualisation and troubleshooting of the following chapters of Python code. The code was executed in Visual Studio Code or Google Colab.

III.I nDSM Processing (Visual Studio Code)

```
#------------------------
#Libraries
#---------

import os
import glob
import rasterio
from rasterio.merge import merge
from rasterio.enums import Resampling
import fiona
import numpy as np
import pandas as pd
import geopandas as gpd
import rasterio
from shapely.geometry import box
from rasterio.transform import Affine
import numpy as np
from rasterio.windows import Window
from rasterio.transform import from_bounds
from rasterio.features import shapes
import networkx as nx


#-------------------------------------------------------------------------------------------
# 1.Raster calculation_height thresholds: data >= 0.3, data <= 4
#-------------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\HSK_raw"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_raster_calc\HSK"

def raster_calculation(input_folder, output_folder, nodata_value=-9999):
    # Find all GeoTIFF files in the input folder
    search_criteria = "*.tif"
    q = os.path.join(input_folder, search_criteria)
    tif_filenames = glob.glob(q)

    # Ensure output folder exists
    os.makedirs(output_folder, exist_ok=True)

    for input_file in tif_filenames:
        with rasterio.open(input_file) as src:
            # Read the first band
            data = src.read(1)

            # Create a mask based on the condition
            mask = np.logical_and(data >= 0.3, data <= 4) * 1

            # Set nodata values
            mask[data == src.nodata] = nodata_value
```

```
            # Update metadata
            out_meta = src.meta.copy()
            out_meta.update({
                "driver": "GTiff",
                "count": 1,
                "nodata": nodata_value
            })

            # Define the output file path
            base_name = os.path.basename(input_file)
            output_file = os.path.join(output_folder, base_name)

            # Write the mask to the output file
            with rasterio.open(output_file, "w", **out_meta) as dest:
                dest.write(mask, 1)

        print(f"Raster calculation complete for {input_file}")


raster_calculation(input_folder, output_folder)
print("All raster calculations complete")


#-------------------------------------------------------------------------------------------
# 2. Raster calculation_mask
#-------------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\HSK_raw"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_wbk_mask\raster_calc\HSK"

def assign_value_to_raster(input_raster_path, output_raster_path, value=1):
    # Open the input raster file
    with rasterio.open(input_raster_path) as src:
        # Read the metadata of the input raster
        metadata = src.meta.copy()

        # Create a new array with the same shape as the input raster and fill it with the specified
value
        new_data = np.full(src.shape, value, dtype=src.dtypes[0])

        # Update the metadata to reflect the new data type
        metadata.update(dtype=new_data.dtype)

        # Write the new array to the output raster file
        with rasterio.open(output_raster_path, 'w', **metadata) as dst:
            dst.write(new_data, 1)

        print(f"Raster calculation complete. File saved at: {output_raster_path}")

def process_folder(input_folder, output_folder, value=1):
    # Ensure the output folder exists
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Iterate over all files in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith('.tif'):  # Process only .tif files
            input_raster_path = os.path.join(input_folder, filename)
            output_raster_path = os.path.join(output_folder, filename)

            # Perform the raster calculation
            assign_value_to_raster(input_raster_path, output_raster_path, value)


process_folder(input_folder, output_folder, value=1)
print("All raster calculations complete")

#-------------------------------------------------------------------------------------------
#Polygonise_Raster 1:
# Polygonise the raster with height tresholds
```

```
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_raster_calc\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_polygonise_raster\HSK"

def polygonize_raster(raster_file, output_vector_file, crs="EPSG:25832"):
    # Open the raster file
    with rasterio.open(raster_file) as src:
        image = src.read(1)  # Read the first band
        mask = image != src.nodata  # Mask of valid data

        # Generate polygons
        results = (
            {'properties': {'value': v}, 'geometry': s}
            for i, (s, v) in enumerate(shapes(image, mask=mask, transform=src.transform))
        )

        # Convert to GeoDataFrame
        geoms = list(results)
        gdf = gpd.GeoDataFrame.from_features(geoms)
        gdf.crs = crs  # Set CRS

        # Save to file
        gdf.to_file(output_vector_file, driver="GPKG")

def polygonize_folder(input_folder, output_folder, crs="EPSG:25832"):
    # Ensure output folder exists
    os.makedirs(output_folder, exist_ok=True)

    # Find all GeoTIFF files in the input folder
    search_criteria = "*.tif"
    q = os.path.join(input_folder, search_criteria)
    tif_filenames = glob.glob(q)

    for tif_file in tif_filenames:
        base_name = os.path.basename(tif_file)
        output_file = os.path.join(output_folder, base_name.replace(".tif", "_pol.gpkg"))
        polygonize_raster(tif_file, output_file, crs)
        print(f"Polygonization complete for {tif_file}")


polygonize_folder(input_folder, output_folder)
print("All raster polygonizations complete")


#-----------------------------------------------------------------------------------------
#Polygonise_Raster 2:
# Polygonise the raster_mask
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_wbk_mask\raster_calc\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_wbk_mask\mask_polygonise\HSK"

def polygonize_raster(raster_file, output_vector_file, crs="EPSG:25832"):
    # Open the raster file
    with rasterio.open(raster_file) as src:
        image = src.read(1)  # Read the first band
        mask = image != src.nodata  # Mask of valid data

        # Generate polygons
        results = (
            {'properties': {'value': v}, 'geometry': s}
            for i, (s, v) in enumerate(shapes(image, mask=mask, transform=src.transform))
        )

        # Convert to GeoDataFrame
        geoms = list(results)
        gdf = gpd.GeoDataFrame.from_features(geoms)
        gdf.crs = crs  # Set CRS
```

```
        # Save to file
        gdf.to_file(output_vector_file, driver="GPKG")

def polygonize_folder(input_folder, output_folder, crs="EPSG:25832"):
    # Ensure output folder exists
    os.makedirs(output_folder, exist_ok=True)

    # Find all GeoTIFF files in the input folder
    search_criteria = "*.tif"
    q = os.path.join(input_folder, search_criteria)
    tif_filenames = glob.glob(q)

    for tif_file in tif_filenames:
        base_name = os.path.basename(tif_file)
        output_file = os.path.join(output_folder, base_name.replace(".tif", "_pol.gpkg"))
        polygonize_raster(tif_file, output_file, crs)
        print(f"Polygonization complete for {tif_file}")

polygonize_folder(input_folder, output_folder)
print("All raster polygonizations complete")

#-------------------------------------------------------------------------------------------
#Clip CTP
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\Trainingsdaten\d_wbk_ID.gpkg"
mask_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_wbk_mask\mask_polygonise\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\wbk_clip\HSK"

def clip_vector_with_vectors(input_folder, mask_folder, output_folder):
    # Find all mask vector files in the mask folder
    search_criteria = "*.gpkg"
    q = os.path.join(mask_folder, search_criteria)
    mask_filenames = glob.glob(q)

    # Ensure output folder exists
    os.makedirs(output_folder, exist_ok=True)

    # Load the main vector file (GeoPackage)
    vector_gdf = gpd.read_file(input_folder)

    for mask_file in mask_filenames:
        mask_gdf = gpd.read_file(mask_file)

        # Clip the vector data with the mask vector
        clipped = gpd.overlay(vector_gdf, mask_gdf, how='intersection')

        # Define the output file path
        base_name = os.path.splitext(os.path.basename(mask_file))[0]
        output_file = os.path.join(output_folder, f"{base_name}.gpkg")

        # Save the clipped vector data
        clipped.to_file(output_file, driver="GPKG")

        print(f"Clipping complete for {mask_file}")


clip_vector_with_vectors(input_folder, mask_folder, output_folder)
print("All clipping operations complete")


#-------------------------------------------------------------------------------------------
# 5. Overlay_Intersection
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\wbk_clip\HSK"
overlay_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\nDSM_polygonise_raster\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\overlay_intersection\HSK"
```

```
def is_polygon_layer(gdf):
    return gdf.geom_type.unique()[0] == 'Polygon'

def overlay_intersection(input_layer_path, overlay_layer_path, output_layer_path):
    # Load vector layers
    print(f"Loading vector layers for {input_layer_path} and {overlay_layer_path}...")

    input_layer = gpd.read_file(input_layer_path)

    overlay_layer = gpd.read_file(overlay_layer_path)

    # Ensure both layers have the same CRS
    print("Ensuring both layers have the same CRS...")
    if input_layer.crs != overlay_layer.crs:
        overlay_layer = overlay_layer.to_crs(input_layer.crs)

    # Check if input_layer is empty
    if input_layer.empty:
        print(f"Input layer {input_layer_path} is empty. Skipping...")
        return

    # Perform overlay intersection
    print(f"Performing overlay intersection for {input_layer_path} and {overlay_layer_path}...")
    intersection = gpd.overlay(input_layer, overlay_layer, how='intersection')

    # Check if intersection is empty after filtering
    if intersection.empty:
        print(f"No polygons in the intersection result for {input_layer_path} and {over-
lay_layer_path}. Skipping...")
        return

    # Save the intersection result to a new file
    print(f"Saving the intersection result to {output_layer_path}...")
    intersection.to_file(output_layer_path, driver='GPKG')
    print(f"Intersection complete. File saved at: {output_layer_path}")

def process_folder(input_folder, overlay_folder, output_folder):
    # Ensure the output folder exists
    os.makedirs(output_folder, exist_ok=True)

    # Iterate over each input file in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith(".gpkg"):
            input_layer_path = os.path.join(input_folder, filename)
            overlay_layer_path = os.path.join(overlay_folder, filename)
            output_layer_path = os.path.join(output_folder, filename)

            # Process the input and overlay layers
            overlay_intersection(input_layer_path, overlay_layer_path, output_layer_path)


# Process the entire folder
process_folder(input_folder, overlay_folder, output_folder)


#-----------------------------------------------------------------------------------------
# 6. Merge separated CTP polygons
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\overlay_intersection\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\overlay_intersection_merge\HSK"


def merge_touching_polygons(gdf, id_col="WBK_ID"):
    merged_gdfs = []
    for wbk_id, group in gdf.groupby(id_col):
        # Build graph: nodes are indices, edges where polygons touch
        G = nx.Graph()
        indices = list(group.index)
```

```
        G.add_nodes_from(indices)
        for i, poly1 in group.geometry.iteritems():
            for j, poly2 in group.geometry.iteritems():
                if i < j and poly1.touches(poly2):
                    G.add_edge(i, j)
        # For each connected component, merge all polygons
        for component in nx.connected_components(G):
            polys = group.loc[list(component)].geometry
            merged_geom = unary_union(polys)
            merged_gdfs.append({
                id_col: wbk_id,
                "geometry": merged_geom
            })
    return gpd.GeoDataFrame(merged_gdfs, crs=gdf.crs)



#-------------------------------------------------------------------------------------
# 7. Coverage - threshold 25%_ new folder with selected CTP polygons
#-----------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\overlay_intersection_merge\HSK"
output_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\wbk_coverage\HSK\HSK-25"
coverage_threshold = 25  # Percentage

def calculate_coverage_and_dissolve(gpkg_path, output_path, coverage_threshold=25):
    # Load the GeoPackage
    print(f"Loading GeoPackage: {gpkg_path}")
    gdf = gpd.read_file(gpkg_path)

    # Calculate the area of each polygon
    print("Calculating polygon areas...")
    gdf['area'] = gdf.geometry.area

    # Filter polygons where value_2 == 1
    print("Filtering polygons where value_2 == 1...")
    value_2_gdf = gdf[gdf['value_2'] == 1]

    # Group by WBK_ID and calculate total and value_2 areas
    print("Grouping by WBK_ID and calculating total and value_2 areas...")
    total_area = gdf.groupby('WBK_ID')['area'].sum().reset_index(name='total_area')
    value_2_area = value_2_gdf.groupby('WBK_ID')['area'].sum().reset_index(name='value_2_area')

    # Merge total_area and value_2_area on WBK_ID
    print("Merging total and value_2 areas...")
    merged_areas = total_area.merge(value_2_area, on='WBK_ID', how='left').fillna(0)

    # Calculate coverage percentage
    print("Calculating coverage percentage...")
    merged_areas['coverage_percent'] = (merged_areas['value_2_area'] / merged_areas['total_area'])
* 100

    # Filter WBK_ID groups with coverage greater than the threshold
    print(f"Filtering WBK_ID groups with coverage > {coverage_threshold}%...")
    selected_WBK_ID = merged_areas[merged_areas['coverage_percent'] > coverage_threshold]['WBK_ID']

    # Select polygons with the selected WBK_ID
    print("Selecting polygons with the selected WBK_ID...")
    filtered_gdf = gdf[gdf['WBK_ID'].isin(selected_WBK_ID)]

    # Dissolve by WBK_ID
    print("Dissolving polygons by WBK_ID...")
    dissolved_gdf = filtered_gdf.dissolve(by='WBK_ID')

    # Save the dissolved polygons to a new file
    print(f"Saving the dissolved polygons to {output_path}...")
    dissolved_gdf.to_file(output_path, driver='GPKG')
    print("Processing complete. File saved at:", output_path)
    print("-----------------------------------------")
```

```
def process_folder(input_folder, output_folder, coverage_threshold=10):
    # Ensure the output folder exists
    os.makedirs(output_folder, exist_ok=True)

    # Iterate over each GeoPackage file in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith(".gpkg"):
            gpkg_path = os.path.join(input_folder, filename)
            output_path = os.path.join(output_folder, f"{os.path.splitext(filename)[0]}_cover-
age_{coverage_threshold}.gpkg")

            # Process each GeoPackage file
            calculate_coverage_and_dissolve(gpkg_path, output_path, coverage_threshold)


process_folder(input_folder, output_folder, coverage_threshold)


#-----------------------------------------------------------------------------------------
# 8. Merge of all selected CTP polygons (> 25%)
#-----------------------------------------------

# Folder connections
input_folder = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\wbk_coverage\HSK\HSK-25"
output_gpkg_path = r"C:\Users\felix\Daten\MA\Daten\PreProcessing\wbk_coverage\HSK\HSK-25.gpkg"

def merge_gpkg_layers(input_folder, output_gpkg_path):
    all_layers = []

    # Iterate over each file in the input folder
    for filename in os.listdir(input_folder):
        if filename.endswith(".gpkg"):
            input_layer_path = os.path.join(input_folder, filename)
            # Get the list of layer names in the GPKG
            layer_names = gpd.io.file.fiona.listlayers(input_layer_path)
            for layer in layer_names:
                # Read the layer into a GeoDataFrame
                gdf = gpd.read_file(input_layer_path, layer=layer)
                all_layers.append(gdf)

    # Concatenate all non-empty GeoDataFrames
    if all_layers:
        combined_gdf = gpd.GeoDataFrame(pd.concat(all_layers, ignore_index=True))

        # Save the combined GeoDataFrame to a new GPKG
        combined_gdf.to_file(output_gpkg_path, driver="GPKG")
        print(f"Combined GeoDataFrame saved to {output_gpkg_path}")
    else:
        print("No non-empty GeoDataFrames found to merge.")


merge_gpkg_layers(input_folder, output_gpkg_path)
```

III.II Generating training dataset (Google Colab)

III.II.I Positive training dataset

```
#------------------------
#Installations
#---------

!pip install geopandas
!pip install rasterio
!pip install os
!pip install shapely
!pip install numpy

#------------------------
#Libraries
#---------


from google.colab import drive
import os
import rasterio
from shapely.geometry import box
from tqdm import tqdm
import numpy as np
import time
import pickle
import geopandas as gpd
import rasterio
from rasterio.mask import mask
from rasterio.errors import WindowError
from rasterio.windows import Window
from PIL import Image
import matplotlib.pyplot as plt

# Mount Google Drive
drive.mount('/content/drive')


#-------------------------------------------------------------------------------------------
# 1. Processing of J2W to dictionary
#-------------------------------------------


# Folder connections
jp2_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/op_raw/op_HSK_raw'
j2w_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos op_raw/op_HSK_raw/dop_j2w'
output_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_clip'

# Ensure output folder exists
os.makedirs(output_folder, exist_ok=True)

def read_j2w(j2w_path):
    try:
        with open(j2w_path, 'r') as f:
            lines = f.read().splitlines()
        return [float(line.strip()) for line in lines]
    except Exception as e:
        print(f"Error reading {j2w_path}: {str(e)}")
        return None

# Create a dictionary of JP2 files with their bounding boxes
jp2_dict = {}
start_time = time.time()
j2w_files = [f for f in os.listdir(j2w_folder) if f.endswith('.j2w')]
print(f"Total number of J2W files to process: {len(j2w_files)}")

for j2w_file in tqdm(j2w_files, desc="Processing J2W files"):
    jp2_file = j2w_file.replace('.j2w', '.jp2')
```

```
    j2w_path = os.path.join(j2w_folder, j2w_file)

    j2w_data = read_j2w(j2w_path)
    if j2w_data is None:
        print(f"Skipping {j2w_file} due to read error")
        continue

    # Calculate bounding box from J2W data
    try:
        with rasterio.open(os.path.join(jp2_folder, jp2_file)) as src:
            width, height = src.width, src.height
    except Exception as e:
        print(f"Error opening {jp2_file}: {str(e)}")
        continue

    left = j2w_data[4]
    top = j2w_data[5]
    right = left + width * j2w_data[0]
    bottom = top + height * j2w_data[3]

    jp2_dict[jp2_file] = box(left, bottom, right, top)

    if (len(jp2_dict) % 100 == 0) or (len(jp2_dict) == len(j2w_files)):
        elapsed_time = time.time() - start_time
        print(f"Processed {len(jp2_dict)} out of {len(j2w_files)} files. Elapsed time:
{elapsed_time:.2f} seconds")

print(f"Finished processing {len(jp2_dict)} J2W files")

# Save the jp2_dict to a file so it can be used in the next part
with open(os.path.join(output_folder, 'jp2_dict.pkl'), 'wb') as f:
    pickle.dump(jp2_dict, f)

print("J2W processing complete. Results saved for use in polygon processing.")

#-----------------------------------------------------------------------------------------
# 2. Polygon clipping
#-----------------------------------------

# Folder connections
geopackage_path = '/content/drive/MyDrive/WeiBauDetect/Daten/Trainingsdaten/HSK-25_ID.gpkg'
jp2_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/op_raw_2/op_HSK-all_raw'
jp2_dict_path = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_
clip/jp2_dict/jp2_dict.pkl'
output_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_clip/
TIFFs_clipped'

# Load the jp2_dict from the file saved in Part 1
with open(jp2_dict_path, 'rb') as f:
    jp2_dict = pickle.load(f)

print(f"Loaded dictionary with {len(jp2_dict)} JP2 files")

# Read the geopackage
gdf = gpd.read_file(geopackage_path)

def merge_arrays(arrays):
    max_height = max(arr.shape[1] for arr in arrays)
    max_width = max(arr.shape[2] for arr in arrays)
    num_bands = arrays[0].shape[0]
    merged = np.zeros((num_bands, max_height, max_width), dtype=arrays[0].dtype)
    for arr in arrays:
        h, w = arr.shape[1], arr.shape[2]
        merged[:, :h, :w] = np.maximum(merged[:, :h, :w], arr)
    return merged

# Process polygons
for idx, row in tqdm(gdf.iterrows(), total=len(gdf), desc="Processing polygons"):
    polygon = row['geometry']
    id_HSK = row['ID_HSK']
```

```
    # Find intersecting JP2 files
    intersecting_jp2s = [jp2_file for jp2_file, jp2_box in jp2_dict.items() if jp2_box.inter-
sects(polygon)]

    if not intersecting_jp2s:
        print(f"No intersecting JP2 found for polygon with ID_HSK: {id_HSK}")
        continue

    # Clip and merge
    clipped_arrays = []
    for jp2_file in intersecting_jp2s:
        jp2_path = os.path.join(jp2_folder, jp2_file)
        try:
            with rasterio.open(jp2_path) as src:
                try:
                    out_image, out_transform = mask(src, [polygon], crop=True)
                    clipped_arrays.append(out_image)
                except ValueError as e:
                    print(f"Polygon with ID_HSK: {id_HSK} does not overlap with {jp2_file}")
                    continue
        except rasterio.errors.RasterioIOError as e:
            print(f"Error opening {jp2_file}: {str(e)}")
            continue

    if not clipped_arrays:
        print(f"No valid clipped data for polygon with ID_HSK: {id_HSK}")
        continue

    # Merge if multiple JP2s
    if len(clipped_arrays) > 1:
        merged_array = merge_arrays(clipped_arrays)
    else:
        merged_array = clipped_arrays[0]

    # Save the result
    output_filename = f"WBK_{id_HSK}_clip.tif"
    output_path = os.path.join(output_folder, output_filename)

    with rasterio.open(os.path.join(jp2_folder, intersecting_jp2s[0])) as src:
        kwargs = src.meta.copy()
        kwargs.update({
            'height': merged_array.shape[1],
            'width': merged_array.shape[2],
            'transform': out_transform
        })

        with rasterio.open(output_path, 'w', **kwargs) as dst:
            dst.write(merged_array)
            dst.colorinterp = [
                rasterio.enums.ColorInterp.red,
                rasterio.enums.ColorInterp.green,
                rasterio.enums.ColorInterp.blue,
                rasterio.enums.ColorInterp.undefined
            ]

    print(f"Saved clipped polygon with ID_HSK: {id_HSK} to {output_filename}")

print("Polygon processing complete!")

"""# 3. Separation of clippings in 640x640 tiles; null-values < 50%"""


#-------------------------------------------------------------------------------------
# 3. Separation of clippings in 640x640 tiles; null-values < 50%
#-----------------------------------------

# Folder connections
source_folder_path = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_clip/
TIFFs_clipped'
output_folder_path = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_TD_pos-
tiles_pool'
```

```
# Function to get file path
def get_file_path(folder_path, filename):
    return os.path.join(folder_path, filename)

# Function to check if a segment has at most 50% zeros
def is_valid_segment(segment):
    total_pixels = segment.shape[0] * segment.shape[1]
    zero_pixels = np.sum(np.all(segment == 0, axis=-1))
    return zero_pixels <= total_pixels * 0.5

# Function to create segments
def create_segments(image, size=640):
    height, width, _ = image.shape
    segments = []
    for y in range(0, height, size):
        for x in range(0, width, size):
            segment = image[y:y+size, x:x+size]
            if segment.shape[0] == size and segment.shape[1] == size:
                if is_valid_segment(segment):
                    segments.append((segment, x, y))
    return segments

# Function to display sample segments
def display_sample_segments(segments, title):
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))
    fig.suptitle(title, fontsize=16)
    for i, ax in enumerate(axes.flat):
        if i < len(segments):
            ax.imshow(segments[i][0])
            ax.axis('off')
            ax.set_title(f'Segment {i+1}')
        else:
            ax.axis('off')
    plt.tight_layout()
    plt.show()

# Get list of GeoTIFF files
geotiff_files = [f for f in os.listdir(source_folder_path) if f.endswith('.tif')]

# Process each GeoTIFF
for idx, geotiff_file in enumerate(tqdm(geotiff_files, desc="Processing GeoTIFFs")):
    image_path = get_file_path(source_folder_path, geotiff_file)
    try:
        with rasterio.open(image_path) as src:
            img_array = src.read().transpose(1, 2, 0)
            transform = src.transform
            crs = src.crs
    except rasterio.errors.RasterioIOError as e:
        print(f"Error opening the file {geotiff_file}: {e}")
        continue

    # Create segments
    segments = create_segments(img_array)

    print(f"Total number of valid 640x640 segments for {geotiff_file}: {len(segments)}")

    # Display sample segments every 100 GeoTIFFs
    if idx % 100 == 0:
        display_sample_segments(segments[:6], f"Sample Segments from {geotiff_file}")

    # Get the base name of the source file (without extension and 'clip')
    source_base_name = os.path.splitext(geotiff_file)[0].replace('_clip', '')

    # Save segments as GeoTIFFs
    for i, (segment, x, y) in enumerate(segments):
        output_filename = f'{source_base_name}_seg-{i+1}.tif'
        output_path = os.path.join(output_folder_path, output_filename)

        # Update the transform for the segment
```

```
        segment_transform = rasterio.Affine(transform.a, transform.b, transform.c + transform.a *
x,
                                            transform.d, transform.e, transform.f + transform.e *
y)

        # Write the GeoTIFF
        with rasterio.open(output_path, 'w', driver='GTiff',
                           height=segment.shape[0], width=segment.shape[1],
                           count=segment.shape[2], dtype=segment.dtype,
                           crs=crs, transform=segment_transform) as dst:
            for j in range(segment.shape[2]):
                dst.write(segment[:,:,j], j+1)

            # Set color interpretation for all bands
            dst.colorinterp = [
                rasterio.enums.ColorInterp.red,
                rasterio.enums.ColorInterp.green,
                rasterio.enums.ColorInterp.blue,
                rasterio.enums.ColorInterp.undefined
            ]

    print(f"Segments for {geotiff_file} saved to: {output_folder_path}")

print("Processing complete!")
```

III.II.II Negative training dataset

```
#------------------------
#Installations
#---------

!pip install rasterio
!pip install shapely
!pip install tqdm

#------------------------
#Libraries
#---------

import os
import random
import rasterio
from rasterio.windows import Window
from google.colab import drive
from tqdm import tqdm
import pickle
from shapely.geometry import box
import logging
import time


# Mount Google Drive
drive.mount('/content/drive')


# Set up logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Folder connections
jp2_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/op_raw/op_HSK_raw'
jp2_dict_path = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_
clip/jp2_dict/jp2_dict.pkl'
output_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/Processing/HSK_TD_neg-
tiles_pool'

# Create output folder if it doesn't exist
os.makedirs(output_folder, exist_ok=True)

# Load the jp2_dict
with open(jp2_dict_path, 'rb') as f:
    jp2_dict = pickle.load(f)

def get_random_window(jp2_dict, size=640, max_attempts=100):
    all_bounds = [polygon.bounds for polygon in jp2_dict.values()]
    total_bounds = (
        min(b[0] for b in all_bounds),
        min(b[1] for b in all_bounds),
        max(b[2] for b in all_bounds),
        max(b[3] for b in all_bounds)
    )

    for attempt in range(max_attempts):
        x = random.uniform(total_bounds[0], total_bounds[2] - size)
        y = random.uniform(total_bounds[1], total_bounds[3] - size)
        window_box = box(x, y, x + size, y + size)

        intersecting_jp2s = [jp2 for jp2, bounds in jp2_dict.items() if window_box.inter-
sects(bounds)]
        if intersecting_jp2s:
            jp2_file = random.choice(intersecting_jp2s)
            return window_box, jp2_file

    return None, None

def process_single_clip(clip_number, max_retries=10):
```

```
    for attempt in range(max_retries):
        start_time = time.time()
        try:
            window_box, jp2_file = get_random_window(jp2_dict)
            if window_box is None:
                logger.warning(f"Failed to find a valid window for clip {clip_number}, attempt {at-
tempt+1}")
                continue

            jp2_path = os.path.join(jp2_folder, jp2_file)

            with rasterio.open(jp2_path) as src:
                window = rasterio.windows.from_bounds(*window_box.bounds, transform=src.transform)
                window = Window(window.col_off, window.row_off, 640, 640)
                data = src.read(window=window)

                if data.shape[1] != 640 or data.shape[2] != 640:
                    logger.warning(f"Attempt {attempt+1} for clip {clip_number}: Incorrect shape
{data.shape}. Retrying...")
                    continue

                transform = src.window_transform(window)
                output_filename = f'non-WBK_rand-clip_{clip_number}.tif'
                output_path = os.path.join(output_folder, output_filename)

                profile = src.profile
                profile.update(
                    driver='GTiff',
                    height=640, width=640,
                    transform=transform,
                    photometric='rgb',
                    nodata=None
                )

                with rasterio.open(output_path, 'w', **profile) as dst:
                    dst.write(data)
                    dst.colorinterp = [rasterio.enums.ColorInterp.red,
                                       rasterio.enums.ColorInterp.green,
                                       rasterio.enums.ColorInterp.blue,
                                       rasterio.enums.ColorInterp.undefined]
                    for j in range(1, dst.count + 1):
                        dst.set_band_description(j, '')

            logger.info(f"Processed clip {clip_number} in {time.time() - start_time:.2f} seconds")
            return True
        except Exception as e:
            logger.error(f"Error processing clip {clip_number}, attempt {attempt+1}: {str(e)}")

    logger.error(f"Failed to process clip {clip_number} after {max_retries} attempts")
    return False

# Generate 1500 random clippings
total_clips = 1500
clip_number = 1
pbar = tqdm(total=total_clips, desc="Generating random clippings")

while clip_number <= total_clips:
    if process_single_clip(clip_number):
        pbar.update(1)
        clip_number += 1

    if clip_number % 10 == 0:
        logger.info(f"Completed {clip_number} clips")

pbar.close()
logger.info("Processing complete!")
```

III.III Evaluation metrics calculation (example for v8 model runs on evaluation area sample test; Visual Studio Code)

```
#------------------------
#Libraries
#---------

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# Data: Model runs and their TP, FP, FN values
data = {
    "Model run": [
        "v8m_run02_010",
        "v8m_run02_005",
        "v8m_run03_HP-TTA _010",
        "v8m_run04_TDv5_005",
        "v8m_run04_TDv5_010",
        "v8m_run05_TDv5_HP-TTA_010",
        "v8m_run05_TDv5_HP-TTA_020",
        "v8m_run06_TDv5_HP_010",
        "v8m_run06_TDv5_HP_020",
        "v8m_run07_cls_TDv5_HP"
    ],
    "TP": [42, 50, 12, 73, 69, 82, 74, 80, 73, 53],
    "FP": [7, 11, 39, 145, 85, 1524, 1218, 1656, 1321, 116],
    "FN": [53, 45, 83, 22, 26, 13, 21, 15, 22, 42],
}

df = pd.DataFrame(data)

# Calculate Precision, Recall, and F1 score
df["Precision"] = df["TP"] / (df["TP"] + df["FP"])
df["Recall"] = df["TP"] / (df["TP"] + df["FN"])
df["F1"] = 2 * (df["Precision"] * df["Recall"]) / (df["Precision"] + df["Recall"])

# Print results for each model run
for idx, row in df.iterrows():
    print(f"Model: {row['Model run']}")
    print(f"  Precision: {row['Precision']:.3f}")
    print(f"  Recall: {row['Recall']:.3f}")
    print(f"  F1 Score: {row['F1']:.3f}")
    print("-" * 30)

# Visualisation
x = np.arange(len(df))
width = 0.25

fig, ax = plt.subplots(figsize=(12, 6))
rects1 = ax.bar(x - width, df["Precision"], width, label="Precision")
rects2 = ax.bar(x, df["Recall"], width, label="Recall")
rects3 = ax.bar(x + width, df["F1"], width, label="F1 Score")

ax.set_ylabel("Score")
ax.set_title("Precision, Recall, F1 Score per v8 Model Run")
ax.set_xticks(x)
ax.set_xticklabels(df["Model run"], rotation=45, ha="right")
ax.legend()
ax.grid(axis="y", linestyle="--", alpha=0.7)

# Function to display bar values
def autolabel(rects):
    for rect in rects:
        height = rect.get_height()
        ax.annotate(f"{height:.2f}",
                    xy=(rect.get_x() + rect.get_width() / 2, height),
                    xytext=(0, 3),  # offset above bar
                    textcoords="offset points",
```

```
                    ha="center", va="bottom")

autolabel(rects1)
autolabel(rects2)
autolabel(rects3)

plt.tight_layout()
plt.show()
```

III.IV YOLOv5 detection (v5_run03_025; Google Colab)

```
#------------------------
#Installations
#---------
# Clone YOLOv5 repository
!git clone https://github.com/ultralytics/yolov5
# %cd yolov5
!pip install -r requirements.txt

# Install additional dependencies for GeoTIFF handling, progress tracking, and geoprocessing
!pip install rasterio tqdm geopandas

#------------------------
#Libraries
#---------
import os
import shutil
import rasterio
import numpy as np
import random
from tqdm import tqdm
from google.colab import drive
import torch
from osgeo import gdal
from PIL import Image, ImageDraw, ImageFont
import csv
from datetime import datetime
import gc
import geopandas as gpd
import pandas as pd

# Mount Google Drive
drive.mount('/content/drive')

print("Setup complete. Starting dataset preparation...")


#-------------------------------------------------------------------------------------------
# 1. YOLOv5 Dataset setup
#-------------------------------------------

# Set up dataset directories
dataset_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/dataset_05'
os.makedirs(dataset_dir, exist_ok=True)
os.makedirs(f'{dataset_dir}/images/train', exist_ok=True)
os.makedirs(f'{dataset_dir}/images/val', exist_ok=True)
os.makedirs(f'{dataset_dir}/labels/train', exist_ok=True)
os.makedirs(f'{dataset_dir}/labels/val', exist_ok=True)

print("Dataset directories created.")

# Function to preprocess GeoTIFF images
def preprocess_geotiff(input_path, output_path):
    with rasterio.open(input_path) as src:
        image = src.read()

    # Select 3 bands (assuming 4-band input)
    rgb_image = image[:3]

    # Normalize and convert to 8-bit
    rgb_image = (rgb_image / rgb_image.max() * 255).astype(np.uint8)

    # Transpose to (height, width, channels)
    rgb_image = np.transpose(rgb_image, (1, 2, 0))

    # Save as PNG
    Image.fromarray(rgb_image).save(output_path)

# Preprocess and organize dataset
```

```
positive_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/Trainingsdaten/positiv'
negative_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/negative'

def process_images(input_dir, is_positive):
    files = [f for f in os.listdir(input_dir) if f.endswith('.tif')]
    total_files = len(files)
    print(f"Processing {'positive' if is_positive else 'negative'} images: {total_files} files
found")

    for i, filename in enumerate(tqdm(files, desc=f"{'Positive' if is_positive else 'Negative'} im-
ages")):
        input_path = os.path.join(input_dir, filename)

        # Choose train or val set (80% train, 20% val)
        is_train = random.random() < 0.8

        # Set output directory
        output_dir = f'{dataset_dir}/images/{"train" if is_train else "val"}'

        # Set output filename
        output_filename = f'{"positive" if is_positive else "negative"}_{i:04d}.png'
        output_path = os.path.join(output_dir, output_filename)

        # Preprocess and save image
        preprocess_geotiff(input_path, output_path)

        # Create label file for positive samples
        if is_positive:
            label_dir = f'{dataset_dir}/labels/{"train" if is_train else "val"}'
            label_filename = f'positive_{i:04d}.txt'
            label_path = os.path.join(label_dir, label_filename)


            with open(label_path, 'w') as f:
                f.write('0 0.5 0.5 1.0 1.0\n')

# Process positive and negative images
process_images(positive_dir, is_positive=True)
process_images(negative_dir, is_positive=False)

print("Image processing complete. Creating dataset.yaml file...")

# Create dataset.yaml file
yaml_content = f"""
train: {dataset_dir}/images/train
val: {dataset_dir}/images/val

nc: 1
names: ['positive']
"""

with open('dataset.yaml', 'w') as f:
    f.write(yaml_content)

print("dataset.yaml file created. Starting YOLOv5 training...")


#-------------------------------------------------------------------------------------
# 2. YOLOv5 model training
#-----------------------------------------

# Train YOLOv5 model
!python train.py --img 640 --batch 16 --epochs 100 --data dataset.yaml --weights yolov5s.pt

print("Training complete. Running validation...")

# Run validation
!python val.py --weights runs/train/exp/weights/best.pt --data dataset.yaml --img 640

print("Validation complete. Running inference on test image...")
```

```
# Run inference on a test image
!python detect.py --weights runs/train/exp/weights/best.pt --source /content/drive/MyDrive/WeiBau-
Detect/Daten/yolov5/run_02/Trainingsdaten/positive_samples/WBK_1025_seg-13.tif --img 640

print("Inference complete. Saving trained model...")

# Create the directory if it doesn't exist
os.makedirs('/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/results/', exist_ok=True)

# Save the trained model
model_path = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/results/trained_model_r05.pt'
!cp runs/train/exp/weights/best.pt {model_path}

# Verify the model file exists
import os
if os.path.exists(model_path):
    print(f"Model successfully saved at: {model_path}")
else:
    print(f"Error: Model not found at: {model_path}")
    print("Please check the training logs for any errors.")

# Load the saved model to ensure it's valid
import torch
try:
    loaded_model = torch.load(model_path)
    print("Model loaded successfully. The saved model is valid.")
except Exception as e:
    print(f"Error loading the saved model: {str(e)}")
    print("Please check the training logs for any errors.")

print("Process complete!")


#-------------------------------------------------------------------------------------------
# 3. YOLOv5 inference
#-------------------------------------------

def load_custom_model(model_path):
    model = torch.hub.load('ultralytics/yolov5', 'custom', path=model_path)
    for m in model.modules():
        if hasattr(m, '_apply'):
            m._apply = lambda fn: m
    return model

# Load custom model
model_path = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/results/trained_model_r03.pt'
model = load_custom_model(model_path)

# Paths to files in Google Drive
image_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/EA_640/batch_1'
results_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/results/EA/batch_1'
positive_folder = os.path.join(results_folder, 'positive')
potential_folder = os.path.join(results_folder, 'potential')
negative_folder = os.path.join(results_folder, 'negative')

# Ensure these folders exist
os.makedirs(positive_folder, exist_ok=True)
os.makedirs(potential, exist_ok=True)
os.makedirs(negative_folder, exist_ok=True)

# Create a unique CSV file name using current timestamp
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
csv_file_path = os.path.join(results_folder, f'batch_1-confidence_scores_{timestamp}.csv')

def process_geotiff(input_path, output_path, model):
    ds = gdal.Open(input_path)
    if ds is None:
        print(f"Could not open {input_path}")
        return 0

    img_array = ds.ReadAsArray()
```

```
    img_array = np.transpose(img_array, (1, 2, 0))
    height, width = img_array.shape[:2]

    with torch.amp.autocast(device_type='cuda', dtype=torch.float16):
        results = model(img_array)

    pil_image = Image.fromarray(img_array.astype('uint8'))
    draw = ImageDraw.Draw(pil_image)

    try:
        font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans-Bold.ttf", 12)
    except IOError:
        font = ImageFont.load_default()

    max_conf = 0
    for *xyxy, conf, cls in results.xyxy[0]:
        max_conf = max(max_conf, conf)
        x1, y1, x2, y2 = map(int, xyxy)

        x1, y1, x2, y2 = [max(0, min(val, width-1 if i % 2 == 0 else height-1)) for i, val in enu-
merate([x1, y1, x2, y2])]

        if conf >= 0.5:
            color = "green"
        elif conf >= 0.25:
            color = "orange"
        else:
            color = "red"

        draw.rectangle([x1, y1, x2, y2], outline=color, width=2)

        label = f'{conf:.2f}'
        bbox = draw.textbbox((0, 0), label, font=font)
        text_width, text_height = bbox[2] - bbox[0], bbox[3] - bbox[1]

        text_x = max(x1, min(width - text_width, x2 - text_width))
        text_y = max(0, y1 - text_height - 2)

        bg_x1, bg_y1 = max(0, text_x - 2), max(0, text_y - 2)
        bg_x2 = min(width - 1, text_x + text_width + 2)
        bg_y2 = min(height - 1, text_y + text_height + 2)

        draw.rectangle([bg_x1, bg_y1, bg_x2, bg_y2], fill=color)
        draw.text((text_x, text_y), label, fill="white", font=font)

    if max_conf < 0.3:
        draw.rectangle([0, 0, width - 1, height - 1], outline="red", width=2)

    img_array = np.array(pil_image)

    driver = gdal.GetDriverByName('GTiff')
    out_ds = driver.Create(output_path, ds.RasterXSize, ds.RasterYSize, ds.RasterCount,
ds.GetRasterBand(1).DataType)

    out_ds.SetGeoTransform(ds.GetGeoTransform())
    out_ds.SetProjection(ds.GetProjection())

    for i in range(ds.RasterCount):
        out_ds.GetRasterBand(i+1).WriteArray(img_array[:,:,i])

    ds = None
    out_ds = None

    return max_conf

def get_all_geotiffs(folder):
    geotiffs = []
    for root, dirs, files in os.walk(folder):
        for file in files:
            if file.lower().endswith('.tif'):
                geotiffs.append(os.path.join(root, file))
```

```
    return geotiffs

image_files = get_all_geotiffs(image_folder)

# Open new CSV file for writing
with open(csv_file_path, 'w', newline='') as csvfile:
    csv_writer = csv.writer(csvfile)
    csv_writer.writerow(['Image Name', 'Confidence Score', 'Classification'])

    for image_path in tqdm(image_files, desc="Processing images"):
        rel_path = os.path.relpath(image_path, image_folder)
        subfolder = os.path.dirname(rel_path)
        image_file = os.path.basename(rel_path)

        temp_output = 'temp_output.tif'
        max_confidence_score = process_geotiff(image_path, temp_output, model)

        if max_confidence_score >= 0.5:
            output_folder = os.path.join(positive_folder, subfolder)
            prefix = 'pos_'
            classification = 'Positive'
        elif max_confidence_score >= 0.25:
            output_folder = os.path.join(potential_folder, subfolder)
            prefix = 'potential_'
            classification = 'Potential'
        else:
            output_folder = os.path.join(negative_folder, subfolder)
            prefix = 'neg_'
            classification = 'Negative'

        os.makedirs(output_folder, exist_ok=True)
        output_path = os.path.join(output_folder, f'{prefix}{image_file}')
        shutil.copy2(temp_output, output_path)
        os.remove(temp_output)

        # Write to CSV
        csv_writer.writerow([image_file, f'{max_confidence_score:.4f}', classification])



        print(f"Processed {image_file} - Saved to {os.path.relpath(output_folder, results_folder)}
(max conf: {max_confidence_score:.2f})")

print("All images processed.")
print(f"Confidence scores saved to {csv_file_path}")
# Cleanup steps
print("Cleaning up...")

# Clear GPU cache if using GPU
if torch.cuda.is_available():
    torch.cuda.empty_cache()

# Clear RAM
gc.collect()

# Remove any remaining temporary files
if os.path.exists('temp_output.tif'):
    os.remove('temp_output.tif')


#-------------------------------------------------------------------------------------
# 4. YOLOv5 results processing: CSV to Geopackage
#-------------------------------------------

# Paths to files in Google Drive
geopackage_path = '/content/drive/MyDrive/WeiBauDetect/Daten/vector_data/EA_640/grid_batch-1.gpkg'
csv_path = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/results/im-
age_tests/EG_gesamt/batch_1/test01/batch_1-confidence_scores_20241016_184915.csv'
output_geopackage_path = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_03/re-
sults/EA/EA_v5_batch-1.gpkg'
```

```
# Read the GeoPackage
gdf = gpd.read_file(geopackage_path)

# Read the CSV file
df = pd.read_csv(csv_path)

# Merge the GeoDataFrame with the CSV data
merged_gdf = gdf.merge(df, left_on='filename', right_on='Image Name', how='left')

# Create new fields for "Confidence Score" and "Classification"
merged_gdf['Confidence Score'] = merged_gdf['Confidence Score']
merged_gdf['Classification'] = merged_gdf['Classification']

# Drop the extra 'image name' column
merged_gdf = merged_gdf.drop(columns=['Image Name'])

# Write the updated GeoDataFrame to a new GeoPackage
merged_gdf.to_file(output_geopackage_path, driver='GPKG')

print(f"Updated GeoPackage saved to: {output_geopackage_path}")
```

III.V YOLOv12 classification (v12_run08_cls_025; Google Colab)

```
#------------------------
#Installations
#---------
!pip -q install --upgrade ultralytics==8.3.139 rasterio==1.3.9 pillow==10.3.0 tqdm==4.66.5

# Install additional dependencies for GeoTIFF handling and geoprocessing
!pip -q install geopandas==0.14.4 shapely==2.0.4 fiona==1.9.6 pyproj==3.6.1 pandas==2.2.2

#------------------------
#Libraries
#---------
import os
from glob import glob
from sklearn.model_selection import train_test_split
from ultralytics import YOLO
from PIL import Image
import csv
from datetime import datetime
from hashlib import md5
import geopandas as gpd
import numpy as np
import pandas as pd
from google.colab import drive
from hashlib import md5
from tqdm import tqdm
import rasterio
from rasterio.errors import RasterioIOError
import csv, time, re
from tqdm import tqdm
from shapely.geometry import box

# Mount Google Drive
drive.mount('/content/drive')

print("Setup complete. Starting dataset preparation...")

#-----------------------------------------------------------------------------------------
# 1. YOLOv12 Dataset setup
#-----------------------------------------

# === DATASET CONFIGURATION ===
dataset_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/dataset'
positive_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/training_data/positive'
negative_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/training_data/negative'

CLASS_NAMES = ['ctp', 'neg']
TEST_RATIO = 0.2
RANDOM_SEED = 42
# ==========================

# Set up dataset directories (YOLOv12 classification format)
os.makedirs(f'{dataset_dir}/train/positive', exist_ok=True)
os.makedirs(f'{dataset_dir}/train/negative', exist_ok=True)
os.makedirs(f'{dataset_dir}/val/positive', exist_ok=True)
os.makedirs(f'{dataset_dir}/val/negative', exist_ok=True)

print("YOLOv12 classification directories created.")


def deduplicate_files(file_list):
    """Remove duplicate files using MD5 hashing"""
    hashes = set()
    unique_files = []
    for file_path in file_list:
        try:
            with open(file_path, 'rb') as f:
                file_hash = md5(f.read()).hexdigest()
            if file_hash not in hashes:
```

```
                hashes.add(file_hash)
                unique_files.append(file_path)
        except Exception as e:
            print(f"Error processing {file_path}: {str(e)}")
    return unique_files

def preprocess_geotiff(input_path, output_path):
    """Convert and optimize geotiff images"""
    with rasterio.open(input_path) as src:
        image = src.read()

    # Convert to 3-channel RGB
    rgb_image = image[:3] if image.shape[0] >= 3 else np.stack([image[0]]*3)
    rgb_image = (rgb_image / rgb_image.max() * 255).astype(np.uint8)
    rgb_image = np.transpose(rgb_image, (1, 2, 0))

    Image.fromarray(rgb_image).save(output_path, optimize=True, compress_level=3)

def process_images(input_dir, class_name, seed=42):
    """Process images with deterministic 80/20 split and deduplication"""
    # Get and deduplicate files
    file_paths = [os.path.join(input_dir, f)
                  for f in os.listdir(input_dir) if f.endswith('.tif')]
    unique_files = deduplicate_files(file_paths)

    print(f"Processing {class_name} images: {len(unique_files)} unique files found")

    # Deterministic split (80/20)
    train_files, val_files = train_test_split(
        unique_files,
        test_size=TEST_RATIO
        random_state= RANDOM_SEED,
        shuffle=True
    )

    # Process training images
    for file_path in tqdm(train_files, desc=f"Training {class_name}"):
        filename = os.path.basename(file_path)
        output_path = f'{dataset_dir}/train/{class_name}/{os.path.splitext(filename)[0]}.png'
        preprocess_geotiff(file_path, output_path)

    # Process validation images
    for file_path in tqdm(val_files, desc=f"Validation {class_name}"):
        filename = os.path.basename(file_path)
        output_path = f'{dataset_dir}/val/{class_name}/{os.path.splitext(filename)[0]}.png'
        preprocess_geotiff(file_path, output_path)

# Process images with deterministic splitting
process_images(positive_dir, 'positive')
process_images(negative_dir, 'negative')

print("Image processing complete. Creating YOLOv12 dataset.yaml...")

# YOLOv12 classification YAML configuration
yaml_content = f"""
train: {dataset_dir}/train
val: {dataset_dir}/val

nc: 2
names: {CLASS_NAMES}
"""

with open(f'{dataset_dir}/dataset.yaml', 'w') as f:
    f.write(yaml_content)

print("YAML configuration created.")
```

```
#-------------------------------------------------------------------------------------------
# 2. YOLOv12 cls model training
#-------------------------------------------


# Load model
model = YOLO('/content/yolov12/ultralytics/cfg/models/v12/yolov12m_cls.yaml', task='classify')

# Enhanced training configuration
model.train(
    data='/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/dataset',  # Verify exact
YAML path
    epochs=200,
    imgsz=640,
    batch=16,
    patience=30,  # Early stopping after 30 epochs without improvement
    optimizer='AdamW',
    lr0=0.001,
    weight_decay=0.05,  # Increased L2 regularization
    hsv_h=0.3,  # Stronger hue augmentation
    hsv_s=0.8,   # Increased saturation variation
    hsv_v=0.5,   # Wider brightness range
    degrees=45,   # More rotation augmentation
    translate=0.2,# Position shifting
    scale=0.5,    # Zoom augmentation
    shear=15,     # Perspective distortion
    perspective=0.001,  # Warping effect
    flipud=0.5,   # Vertical flip
    fliplr=0.5,   # Horizontal flip
    mosaic=0.5,   # Mosaic augmentation probability
    mixup=0.2,    # Image mixing
    copy_paste=0.2, # Object copying
    auto_augment='randaugment',  # Auto augmentation
    erasing=0.2,  # Random erasing
    dropout=0.5,
    project='/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/results',
    name='run08_yolov12m-cls'
)

#-------------------------------------------------------------------------------------------
# 3. YOLOv12 inference
#-------------------------------------------

# === INFERENCE CONFIGURATION ===
MODEL_PATH = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/results/run08_yolov12m-
cls/weights/best.pt'
ROOT_IMAGES_DIR = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/EA_640/batch_1'
RECURSIVE = True
CONFIDENCE_THRESHOLD = 0.25

PROJECT_DIR = "/content/runs/classify"
RUN_NAME = f"exp_{timestamp}"

OUT_DIR = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/cls/run_08/results/EA/batch_1'
CSV_FILE_PATH = os.path.join(OUT_DIR, 'v12m_cls_b1.csv')
TRACK_LOG_PATH = os.path.join(OUT_DIR, f'v12m_cls_b1_{timestamp}.log')
# ==========================

os.makedirs(OUT_DIR, exist_ok=True)

# --- DISCOVER IMAGES ---
patterns = ["*.tif", "*.tiff", "*.TIF", "*.TIFF"]
if RECURSIVE:
    image_paths = []
    for pattern in patterns:
        image_paths.extend(glob.glob(os.path.join(ROOT_IMAGES_DIR, "**", pattern), recursive=True))
else:
    image_paths = []
    for pattern in patterns:
        image_paths.extend(glob.glob(os.path.join(ROOT_IMAGES_DIR, pattern), recursive=False))
```

```
# Dedup preserve order
seen = set()
all_image_files = []
for p in image_paths:
    if p not in seen:
        seen.add(p)
        all_image_files.append(p)

total_count = len(all_image_files)
existing_image_files = [p for p in all_image_files if os.path.exists(p)]
missing_paths = [p for p in all_image_files if not os.path.exists(p)]
missing_count = len(missing_paths)
existing_count = len(existing_image_files)

print(f"[TRACK] Total discovered: {total_count}")
print(f"[TRACK] Existing: {existing_count}")
print(f"[TRACK] Missing: {missing_count}")
if not existing_image_files:
    raise RuntimeError("No existing image files found to process under ROOT_IMAGES_DIR.")

# --- MODEL ---
model = YOLO(MODEL_PATH, task='classify')

# Regex to parsing
neg_re = re.compile(r'negative\s+([0-9]*\.?[0-9]+)')
pos_re = re.compile(r'positive\s+([0-9]*\.?[0-9]+)')

# --- INFERENCE + TRACK ---
inference_started = datetime.now()
t0 = time.time()
print(f"[TRACK] Inference start: {inference_started.isoformat()}")

rows = []
header = ['Image Name', 'Negative Score', 'Positive Score', 'Classification']

parsed_count = 0
fallback_count = 0
errors = []
per_image_log = []

# Parse each Results item
# Ultralytics predict returns a list of Results, one per input image. [1][2]
batch_size = 64
for i in range(0, existing_count, batch_size):
    batch = existing_image_files[i:i+batch_size]
    try:
        results = model.predict(batch, project=PROJECT_DIR, name=RUN_NAME, save=False, ver-
bose=True)
    except Exception as e:
        # On fatal batch error, fallback each image as negative
        errors.append((f"batch_{i}", f"Predict error: {repr(e)}"))
        for img_path in batch:
            rows.append([os.path.basename(img_path), f"{1.0:.4f}", f"{0.0:.4f}", "negative"])
            fallback_count += 1
        continue

    # Parse each result with verbose()/summary() [2]
    for img_path, r in zip(batch, results):
        neg_score = None
        pos_score = None
        parsed_ok = False
        try:
            v = r.verbose()  # e.g., "negative 0.97, positive 0.03" [2]
            mneg = neg_re.search(v)
            mpos = pos_re.search(v)
            if mneg and mpos:
                neg_score = float(mneg.group(1))
                pos_score = float(mpos.group(1))
                parsed_ok = True
            else:
                # Fallback: use summary() top class/conf and infer complement [2]
```

```
                sm = r.summary()  # list with dict having 'class' and 'conf' [2]
                if isinstance(sm, list) and sm:
                    top = sm
                    cls_idx = int(top.get('class', 0))
                    conf = float(top.get('conf', 0.0))
                    if cls_idx == 1:
                        pos_score = conf
                        neg_score = 1.0 - conf
                    else:
                        neg_score = conf
                        pos_score = 1.0 - conf
                    parsed_ok = True
        except Exception as e:
            errors.append((img_path, f"Parse error: {repr(e)}"))

        if not parsed_ok:
            neg_score, pos_score = 1.0, 0.0
            fallback_count += 1
        else:
            parsed_count += 1

        classification = 'positive' if pos_score > CONFIDENCE_THRESHOLD else 'negative'
        rows.append([os.path.basename(img_path), f"{neg_score:.4f}", f"{pos_score:.4f}", classifi-
cation])
        per_image_log.append(
            f"{os.path.basename(img_path)} | parsed_ok={parsed_ok} | neg={neg_score:.4f} |
pos={pos_score:.4f} | cls={classification}"
        )

        if (len(rows) % 100) == 0:
            print(f"[TRACK] Parsed {len(rows)}/{existing_count} images")

inference_ended = datetime.now()
t1 = time.time()
elapsed = t1 - t0
ips = existing_count / elapsed if elapsed > 0 else 0.0
print(f"[TRACK] Inference end:   {inference_ended.isoformat()}")
print(f"[TRACK] Inference elapsed: {elapsed:.2f}s, ~{ips:.2f} images/s")

# --- WRITE CSV ---
os.makedirs(os.path.dirname(CSV_FILE_PATH), exist_ok=True)
with open(CSV_FILE_PATH, 'w', newline='') as f:
    w = csv.writer(f)
    w.writerow(header)
    w.writerows(rows)

# --- WRITE TRACKING LOG ---
with open(TRACK_LOG_PATH, 'w') as logf:
    logf.write("=== YOLO Classification In-Memory Parsing Tracking ===\n")
    logf.write(f"Timestamp: {timestamp}\n")
    logf.write(f"Model: {MODEL_PATH}\n")
    logf.write(f"Root images dir: {ROOT_IMAGES_DIR}\n")
    logf.write(f"Project run: {os.path.join(PROJECT_DIR, RUN_NAME)}\n")
    logf.write("\n-- Discovery --\n")
    logf.write(f"Total discovered images: {total_count}\n")
    logf.write(f"Existing images: {existing_count}\n")
    logf.write(f"Missing images: {missing_count}\n")
    if total_count > 0:
        logf.write(f"Existing ratio: {existing_count/total_count:.3f}\n")
    logf.write("\n-- Inference --\n")
    logf.write(f"Start: {inference_started.isoformat()}\n")
    logf.write(f"End:   {inference_ended.isoformat()}\n")
    logf.write(f"Elapsed seconds: {elapsed:.3f}\n")
    logf.write(f"Images/sec (approx): {ips:.3f}\n")
    logf.write("\n-- Parsing --\n")
    logf.write(f"Parsed OK: {parsed_count}\n")
    logf.write(f"Fallback defaults used: {fallback_count}\n")
    if errors:
        logf.write("\n-- Errors --\n")
        for (path, msg) in errors:
            logf.write(f"{os.path.basename(path)}: {msg}\n")
```

```
    logf.write("\n-- Per-Image Status --\n")
    for line in per_image_log:
        logf.write(line + "\n")

print("Processing complete!")
print(f"CSV report: {CSV_FILE_PATH}")
print(f"Tracking log: {TRACK_LOG_PATH}")


#-------------------------------------------------------------------------------------
# 4. YOLOv12 results processing: CSV to Geopackage
#-----------------------------------------

# Config
CSV_PATH = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/classification/run_03/fi-
nal_batches_results/v12m_cls_b1-final.csv'  # path to the CSV with columns: Image Name, Negative
Score, Positive Score, Classification
ROOT_IMAGES_DIR = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov5/run_02/Orthophotos/Evaluier-
ungsgebiet_640_intersect/batch_1'  # folder with original GeoTIFFs
RECURSIVE = True  # scan subfolders
# Output GeoPackage
OUT_GPKG = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/classification/run_03/fi-
nal_batches_results/12m_cls_b1-final.gpkg'
OUT_LAYER = 'positive_tiles'
TRACK_LOG = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/classification/run_03/fi-
nal_batches_results/b1_test_txt-verbose_csv-to-geo.log'


# Load CSV
df = pd.read_csv(CSV_PATH)
print("Columns in CSV:", list(df.columns))
print(f"Loaded {len(df)} rows from {CSV_PATH}")

# Expected columns
required_cols = ['Image Name', 'Negative Score', 'Positive Score', 'Classification']
missing_cols = [c for c in required_cols if c not in df.columns]
if missing_cols:
    raise SystemExit(f"CSV missing required columns: {missing_cols}")

# Discover GeoTIFFs and map by basename
patterns = ['*.tif', '*.tiff', '*.TIF', '*.TIFF']
if RECURSIVE:
    image_paths = []
    for pat in patterns:
        image_paths.extend(glob.glob(os.path.join(ROOT_IMAGES_DIR, '**', pat), recursive=True))
else:
    image_paths = []
    for pat in patterns:
        image_paths.extend(glob.glob(os.path.join(ROOT_IMAGES_DIR, pat), recursive=False))

# Deduplicate and map basename -> full path (last wins if collisions)
seen = set()
ordered = []
for p in image_paths:
    if p not in seen:
        seen.add(p)
        ordered.append(p)
name_to_path = {os.path.basename(p): p for p in ordered}

print(f"Discovered {len(ordered)} GeoTIFF(s) under ROOT_IMAGES_DIR.")

# Filter positive rows
df['Classification'] = df['Classification'].astype(str).str.lower()
df_pos = df[df['Classification'] == 'positive'].copy()
print(f"Positive rows in CSV: {len(df_pos)}")

# Resolve full paths
def resolve_path(name):
    fn = str(name)
    p = name_to_path.get(fn)
    return p if (p and os.path.exists(p)) else None
```

```
df_pos['resolved_path'] = df_pos['Image Name'].apply(resolve_path)
missing_paths = df_pos['resolved_path'].isna().sum()
if missing_paths > 0:
    print(f"[WARN] {missing_paths} positive rows do not resolve to files in ROOT_IMAGES_DIR and
will be skipped.")

df_pos = df_pos[df_pos['resolved_path'].notna()].copy()
if len(df_pos) == 0:
    raise SystemExit("No positive rows with resolvable GeoTIFF paths. Check filenames and
ROOT_IMAGES_DIR.")

# Build polygon footprints from raster bounds
def polygon_from_raster_bounds(path):
    with rasterio.open(path) as src:
        b = src.bounds  # left, bottom, right, top
        poly = box(b.left, b.bottom, b.right, b.top)
        return poly, src.crs

geoms = []
crs_list = []
attrs = []
skipped_open = 0
skipped_no_crs = 0

print("Extracting positive footprints...")
for _, row in tqdm(df_pos.iterrows(), total=len(df_pos)):
    path = row['resolved_path']
    try:
        poly, crs = polygon_from_raster_bounds(path)
    except RasterioIOError as e:
        skipped_open += 1
        continue
    except Exception as e:
        skipped_open += 1
        continue

    if crs is None:
        skipped_no_crs += 1
        continue

    geoms.append(poly)
    crs_list.append(crs)
    attrs.append({
        'Image Name': row['Image Name'],
        'Negative Score': float(row['Negative Score']),
        'Positive Score': float(row['Positive Score']),
        'Classification': row['Classification']
    })

if not geoms:
    raise SystemExit("No valid geometries extracted (failed to open or missing CRS).")

print(f"Valid geometries: {len(geoms)} | Skipped open errors: {skipped_open} | Skipped no CRS:
{skipped_no_crs}")

# Assemble GeoDataFrame, harmonize CRS
gdfs = []
for g, a, c in zip(geoms, attrs, crs_list):
    try:
        gdf = gpd.GeoDataFrame([a], geometry=[g], crs=c)
        gdfs.append(gdf)
    except Exception as e:
        # Skip CRS/geometry issues
        pass

if not gdfs:
    raise SystemExit("No features with valid CRS to write.")

target_crs = gdfs[0].crs
gdfs = [g if g.crs == target_crs else g.to_crs(target_crs) for g in gdfs]
```

```
gdf_all = pd.concat(gdfs, ignore_index=True)
print(f"Final features: {len(gdf_all)} in CRS: {target_crs}")

# Clean attribute names (ensure valid for GPKG)
def clean_col(name):
    s = ''.join(ch if (str(ch).isalnum() or ch == '_') else '_' for ch in str(name))
    if s.lower() in ('geometry', 'fid'):
        s = f"{s}_col"
    return s

rename_map = {c: clean_col(c) for c in gdf_all.columns if c != gdf_all.geometry.name}
gdf_all = gdf_all.rename(columns=rename_map)

# Write GeoPackage
os.makedirs(os.path.dirname(OUT_GPKG), exist_ok=True)
if os.path.exists(OUT_GPKG):
    try:
        os.remove(OUT_GPKG)
    except Exception:
        pass

gdf_all.to_file(OUT_GPKG, layer=OUT_LAYER, driver='GPKG')
print(f"GeoPackage written: {OUT_GPKG} (layer: {OUT_LAYER})")

# Tracking log
os.makedirs(os.path.dirname(TRACK_LOG), exist_ok=True)
with open(TRACK_LOG, 'w') as logf:
    logf.write("=== CSV->GPKG positives build tracking ===\n")
    logf.write(f"Timestamp: {datetime.now().strftime('%Y%m%d_%H%M%S')}\n")
    logf.write(f"CSV: {CSV_PATH}\n")
    logf.write(f"Root images dir: {ROOT_IMAGES_DIR}\n")
    logf.write(f"Output GPKG: {OUT_GPKG} layer={OUT_LAYER}\n\n")
    logf.write(f"Total CSV positives: {len(df[df['Classification']=='positive'])}\n")
    logf.write(f"Resolved positives: {len(df_pos)}\n")
    logf.write(f"Written features: {len(gdf_all)}\n")
    logf.write(f"Skipped open errors: {skipped_open}\n")
    logf.write(f"Skipped no CRS: {skipped_no_crs}\n")

print(f"Tracking log: {TRACK_LOG}")

# Preview
print("\nFirst 3 features:")
print(gdf_all.head(3))
```

III.VI YOLOv8 RGBI detection (v8_run04_TDv5_010; Google Colab)

```
#------------------------
#Installations
#---------
!pip install --upgrade --force-reinstall numpy==1.26.4
!pip install ultralytics==8.3.159
!pip install rasterio
!pip install geopandas
!pip install shapely
!pip install pandas

#------------------------
#Libraries
#---------
import os
import cv2
import torch
import rasterio
import pandas as pd
import numpy as np
from ultralytics import YOLO
from tqdm import tqdm
import shutil
import random
from ultralytics import YOLO
from google.colab import drive
import geopandas as gpd
from shapely.geometry import box
from pathlib import Path
import ast

# Mount Google Drive
drive.mount('/content/drive')

print("Setup complete. Starting dataset preparation...")


#-----------------------------------------------------------------------------------------
# 1. YOLOv8 Dataset setup
#-----------------------------------------

# ====== DIRECTORIES ====== #
dataset_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04/dataset'
DATASET_YAML_PATH = f'{BASE_PATH}/dataset_04_rgbi.yaml'
BASE_PATH = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04'

# Positive images and labels
positive_images_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/TDv5/positive_images'
positive_labels_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/TDv5/posivite_labels'
# Negative images (no labels)
negative_images_dir = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/TDv5/negative_images'

# Set random seed for reproducibility
random.seed(42)
# Class information
CLASS_NAME = 'ctp'
NUM_CLASSES = 1

# ================================== #

# Create YOLOv8 directory structure
train_images_dir = os.path.join(dataset_dir, 'images', 'train')
val_images_dir = os.path.join(dataset_dir, 'images', 'val')
train_labels_dir = os.path.join(dataset_dir, 'labels', 'train')
val_labels_dir = os.path.join(dataset_dir, 'labels', 'val')

# Create all directories
for d in [train_images_dir, val_images_dir, train_labels_dir, val_labels_dir]:
    os.makedirs(d, exist_ok=True)
```

```
# Get positive images and labels
positive_images = [f for f in os.listdir(positive_images_dir)
                   if f.lower().endswith(('.png', '.jpg', '.jpeg', '.tif', '.tiff'))]
positive_labels = [f for f in os.listdir(positive_labels_dir)
                   if f.lower().endswith('.txt')]

# Match images to labels
matched_pairs = []
for img in positive_images:
    base_name = os.path.splitext(img)[0]
    label_file = base_name + '.txt'
    if label_file in positive_labels:
        matched_pairs.append((img, label_file))

# Split positive data (80/20)
random.shuffle(matched_pairs)
split_idx = int(0.8 * len(matched_pairs))
train_pairs = matched_pairs[:split_idx]
val_pairs = matched_pairs[split_idx:]

# Copy positive images and labels
for img, lbl in train_pairs:
    # Copy image
    shutil.copy2(os.path.join(positive_images_dir, img),
                 os.path.join(train_images_dir, img))
    # Copy label
    shutil.copy2(os.path.join(positive_labels_dir, lbl),
                 os.path.join(train_labels_dir, lbl))

for img, lbl in val_pairs:
    # Copy image
    shutil.copy2(os.path.join(positive_images_dir, img),
                 os.path.join(val_images_dir, img))
    # Copy label
    shutil.copy2(os.path.join(positive_labels_dir, lbl),
                 os.path.join(val_labels_dir, lbl))

# Copy negative images to training set (no labels)
negative_images = [f for f in os.listdir(negative_images_dir)
                   if f.lower().endswith(('.png', '.jpg', '.jpeg', '.tif', '.tiff'))]
for img in negative_images:
    shutil.copy2(os.path.join(negative_images_dir, img),
                 os.path.join(train_images_dir, img))

# Print summary
print(f"☑ Data preparation complete!")
print(f"Positive images: {len(matched_pairs)}")
print(f"  - Training: {len(train_pairs)}")
print(f"  - Validation: {len(val_pairs)}")
print(f"Negative images: {len(negative_images)}")
print(f"Output directories:")
print(f"- Images: {train_images_dir} (train), {val_images_dir} (val)")
print(f"- Labels: {train_labels_dir} (train), {val_labels_dir} (val)")

# 1. Create dataset YAML
dataset_config = f"""
train: {DATASET_PATH}/images/train
val: {DATASET_PATH}/images/val

nc: {NUM_CLASSES}
names: ['{CLASS_NAME}']
channels: 4
"""

with open(DATASET_YAML_PATH, 'w') as f:
    f.write(dataset_config)
print(f"☑ Dataset YAML created at: {DATASET_YAML_PATH}")


#------------------------------------------------------------------------------------
```

```
# 2. YOLOv8 RGBI model training
#-----------------------------------------

# ========== CONFIGURATION ==========
MODEL_SIZE = 'm'

# Training parameters
EPOCHS = 200
BATCH_SIZE = 16
IMAGE_SIZE = 640
# ===================================


model = YOLO(f'yolov8{MODEL_SIZE}.pt')
model.train(
    data=DATASET_YAML_PATH,
    epochs=EPOCHS,
    imgsz=IMAGE_SIZE,
    batch=BATCH_SIZE,
    device=0,
    name=f'yolov8{MODEL_SIZE}_04-rgbi_train',
    exist_ok=True,
    patience=30
)

# 3. Save the trained model
model_save_path = f'{BASE_PATH}/run04_yolov8{MODEL_SIZE}_rgbi.pt'
model.save(model_save_path)
print(f"💾 Model saved to: {model_save_path}")


#-------------------------------------------------------------------------------------------
# 3. YOLOv8 inference
#-----------------------------------------

# ====== USER CONFIG ====== #
MODEL_PATH = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04/run04_yolov8m_rgbi.pt'

TEST_IMAGES = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/EA_640/batch_1'
OUTPUT_DIR = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04/results/EA/batch_1'
csv_path = os.path.join(OUTPUT_DIR, 'v8m_run04_b1.csv')

CONF_THRESHOLD = 0.10
# ======================== #

os.makedirs(OUTPUT_DIR, exist_ok=True)
model = YOLO(MODEL_PATH)

# ---- Gather GeoTIFFs recursively (no txt list) ----
def is_geotiff(filename: str) -> bool:
    fn = filename.lower()
    return fn.endswith('.tif') or fn.endswith('.tiff')

image_paths = []
for root, dirs, files in os.walk(TEST_IMAGES):
    for fn in files:
        if is_geotiff(fn):
            image_paths.append(os.path.join(root, fn))

# --- Critical padding function ---
def pad_to_multiple_of_32(arr):
    """Pad array to make height/width divisible by 32"""
    _, h, w = arr.shape
    new_h = ((h + 31) // 32) * 32
    new_w = ((w + 31) // 32) * 32
    pad_h = new_h - h
    pad_w = new_w - w
    arr_padded = np.pad(arr, ((0,0), (0, pad_h), (0, pad_w)), mode='constant')
    return arr_padded, (h, w)

def process_image(path):
```

```
    """Process a single 4-channel TIFF image and return results (CSV only, no image saving)"""
    result = {
        'image': os.path.basename(path),
        'path': path,
        'num_detections': 0,
        'confidences': [],
        'classes': [],
        'bboxes': [],
        'error': None
    }
    try:
        # Read image with rasterio
        with rasterio.open(path) as src:
            arr = src.read()  # (C, H, W); expecting 4 channels

            # Validate channel count
            if arr.shape[0] < 4:
                raise ValueError(f'Expected >=4 channels, got {arr.shape}')

            # Pad to multiple of 32 and get original dimensions
            arr_padded, (orig_h, orig_w) = pad_to_multiple_of_32(arr)

            # Prepare 4-channel tensor (take first 4 channels explicitly)
            arr4 = arr_padded[:4, ...]
            img_tensor = torch.from_numpy(arr4.astype(np.float32)) / 255.0
            img_tensor = img_tensor.unsqueeze(0)  # (1, 4, H_pad, W_pad)

        # Run inference
        predictions = model(img_tensor, conf=CONF_THRESHOLD, verbose=False)[0]
        boxes = predictions.boxes.xyxy.detach().cpu().numpy() if predictions.boxes is not None else
np.empty((0,4))
        confidences = predictions.boxes.conf.detach().cpu().numpy() if predictions.boxes is not
None else np.empty((0,))
        classes = predictions.boxes.cls.detach().cpu().numpy() if predictions.boxes is not None
else np.empty((0,))

        # Adjust boxes to original dimensions (remove padding)
        adjusted_boxes = []
        for box in boxes:
            x1, y1, x2, y2 = box
            # Clamp to original image boundaries
            x1 = max(0, min(float(x1), orig_w))
            y1 = max(0, min(float(y1), orig_h))
            x2 = max(0, min(float(x2), orig_w))
            y2 = max(0, min(float(y2), orig_h))
            adjusted_boxes.append([x1, y1, x2, y2])
        adjusted_boxes = np.array(adjusted_boxes)

        result['num_detections'] = int(len(adjusted_boxes))
        result['confidences'] = confidences.tolist()
        result['classes'] = classes.tolist()
        result['bboxes'] = adjusted_boxes.tolist()

        # No image saving here

    except Exception as e:
        result['error'] = str(e)

    return result

# Process all images
results = []
for path in tqdm(image_paths, desc="Processing images"):
    results.append(process_image(path))

# Convert to DataFrame and save
results_df = pd.DataFrame(results)
results_df.to_csv(csv_path, index=False)

# Print summary
success_count = results_df[results_df['error'].isna()].shape[0]
```

```
detection_count = results_df[results_df['num_detections'] > 0].shape
error_count = results_df['error'].notna().sum()

print(f"\n☑ Inference complete!")
print(f"Images discovered: {len(image_paths)}")
print(f"Images processed: {len(results_df)}")
print(f"Successful inferences: {success_count}")
print(f"Images with detections: {detection_count}")
print(f"Errors encountered: {error_count}")
print(f"Results CSV: {csv_path}")
print(f"Output dir (CSV only): {OUTPUT_DIR}")


#-----------------------------------------------------------------------------------------
# 4. YOLOv8 results processing: CSV to Geopackage
#-----------------------------------------------

# --- Set file paths ---
csv_path = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04/re-
sults/EA/batch_1/v8m_run04_b1.csv'
geotiff_folder = '/content/drive/MyDrive/WeiBauDetect/Daten/Orthophotos/EA_640/batch_1'
output_gpkg = '/content/drive/MyDrive/WeiBauDetect/Daten/yolov8/rgbi/run_04/re-
sults/EA/batch_1/v8m_run04_b1.gpkg'

# --- Read CSV and filter for >0 detection ---
df = pd.read_csv(csv_path)
df = df[df['num_detections'] > 0].copy()

# --- Build a recursive index of GeoTIFFs ---
# Map from bare filename to full path; if duplicates exist, prefer first found (warn on dupli-
cates).
root = Path(geotiff_folder)
tiff_exts = {'.tif', '.tiff', '.TIF', '.TIFF'}

filename_to_path = {}
duplicates = {}

for p in root.rglob('*'):
    if p.is_file() and p.suffix in tiff_exts:
        key = p.name
        if key in filename_to_path:
            # record duplicate for awareness
            duplicates.setdefault(key, []).append(str(p))
        else:
            filename_to_path[key] = str(p)

if duplicates:
    print("Warning: Duplicate filenames detected. The first occurrence will be used for each of
these names:")
    for k, paths in duplicates.items():
        print(f"  {k}:")
        print(f"    primary -> {filename_to_path[k]}")
        for other in paths:
            print(f"    ignored -> {other}")

# --- Iterate CSV rows and create features ---
features = []
missing = 0
parse_errors = 0
raster_errors = 0

for idx, row in df.iterrows():
    filename = row['image']

    filename_key = os.path.basename(filename)

    # Parse confidences safely
    if 'confidences' in row and isinstance(row['confidences'], str):
        try:
            _ = ast.literal_eval(row['confidences'])
        except Exception as e:
```

```
            parse_errors += 1
            print(f"Error parsing confidences at row {idx}: {e}")
            # continue anyway; keep raw string if needed
    # Locate the GeoTIFF path using the recursive index
    geotiff_path = filename_to_path.get(filename_key, None)
    if geotiff_path is None:
        missing += 1
        print(f"GeoTIFF not found for {filename_key} under {geotiff_folder}")
        continue

    # Open raster, derive footprint polygon and CRS
    try:
        with rasterio.open(geotiff_path) as src:
            bounds = src.bounds  # left, bottom, right, top
            crs = src.crs
            poly = box(bounds.left, bounds.bottom, bounds.right, bounds.top)
            rec = row.to_dict()
            rec['geometry'] = poly
            rec['src_path'] = geotiff_path  # optional: keep the source path used
            rec['src_crs'] = str(crs) if crs is not None else None
            features.append(rec)
    except Exception as e:
        raster_errors += 1
        print(f"Error processing {geotiff_path}: {e}")
        continue

# --- Build GeoDataFrame and save ---
if features:
    # For safety, pick the CRS of the first successfully opened raster footprint.
    first_crs = None
    for feat in features:
        if feat.get('src_crs'):
            first_crs = feat['src_crs']
            break

    gdf = gpd.GeoDataFrame(features, geometry='geometry')
    # Assign CRS if found
    if first_crs:
        try:
            gdf = gdf.set_crs(first_crs)
        except Exception as e:
            print(f"Warning: Could not set CRS {first_crs}: {e}")

    # Write to GeoPackage
    gdf.to_file(output_gpkg, driver='GPKG')
    print(f"GeoPackage created with {len(gdf)} polygons at: {output_gpkg}")
else:
    print("No georeferenced features created.")

# --- Report diagnostics ---
print(f"Rows in CSV with detections: {len(df)}")
print(f"Features created: {len(features)}")
print(f"Missing rasters: {missing}")
print(f"Confidence parse errors: {parse_errors}")
print(f"Raster read errors: {raster_errors}")
```

III.VII RF patch classification (RF_2class_NDVI_64_100; Google Colab)

```
#------------------------
#Installations
#---------
!pip install rasterio
!pip install geopandas

#------------------------
#Libraries
#---------
import os
from pathlib import Path
import numpy as np
import pandas as pd
import rasterio
from PIL import Image
import shutil
import os
from pathlib import Path
import numpy as np
import random
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import LabelEncoder
import joblib
from tqdm.notebook import tqdm
from concurrent.futures import ProcessPoolExecutor
import geopandas as gpd
from shapely.geometry import box

# Mount Google Drive
drive.mount('/content/drive')

print("Setup complete. Starting dataset preparation...")


#-------------------------------------------------------------------------------------------
# 1. RF patches dataset setup
#-------------------------------------------

# PATHS
base_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/detection/run_02/train-
ing_data")
images_path = base_path / "positive_images"
labels_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/training_data_2cl/posi-
tive_labels")

# Separate base folders for npy and png
output_npy_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_npy")
output_png_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_inspect")

PATCH_SIZE = 64
IMG_SIZE = 640

# Load class names
classes_file = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/training_data_2cl/posi-
tive_labels/classes.txt"
with open(classes_file) as f:
    class_names = [line.strip() for line in f.readlines()]
print(f"Loaded classes: {class_names}")

# Create class-specific folders for npy and png
output_npy_dirs = {cls: output_npy_base_dir / cls for cls in class_names}
output_png_dirs = {cls: output_png_base_dir / cls for cls in class_names}

def get_bbox_pixel_coords(cx, cy, w, h, img_size=IMG_SIZE):
```

```
    """Convert YOLO bbox normalized coords to pixel bbox edges."""
    x_center = cx * img_size
    y_center = cy * img_size
    box_w = w * img_size
    box_h = h * img_size
    left = x_center - box_w / 2
    right = x_center + box_w / 2
    upper = y_center - box_h / 2
    lower = y_center + box_h / 2
    return left, upper, right, lower

def extract_two_patches_inside_bbox(img_array, bbox_coords, patch_size=PATCH_SIZE):
    """Extract two 64x64 patches fully inside bbox."""
    left, upper, right, lower = bbox_coords
    half_patch = patch_size // 2
    center_x = int((left + right) / 2)
    center_y = int((upper + lower) / 2)
    img_height, img_width = img_array.shape[1], img_array.shape[2]

    # First patch window
    x1 = max(center_x - half_patch, 0)
    y1 = max(center_y - half_patch, 0)
    x2 = x1 + patch_size
    y2 = y1 + patch_size
    if x2 > img_width:
        x2 = img_width
        x1 = x2 - patch_size
    if y2 > img_height:
        y2 = img_height
        y1 = y2 - patch_size
    patch_center = img_array[:, y1:y2, x1:x2]

    # Offset patch center x inside bbox and image limits
    offset_range_min = max(int(left) + half_patch, half_patch)
    offset_range_max = min(int(right) - half_patch, img_width - half_patch)
    attempt_offset_x = center_x + half_patch
    offset_x = min(max(attempt_offset_x, offset_range_min), offset_range_max)
    x1_off = offset_x - half_patch
    y1_off = y1
    x2_off = x1_off + patch_size
    y2_off = y1_off + patch_size
    patch_offset = img_array[:, y1_off:y2_off, x1_off:x2_off]

    patch_center = np.transpose(patch_center, (1, 2, 0))
    patch_offset = np.transpose(patch_offset, (1, 2, 0))
    return patch_center, patch_offset

patch_records = []
label_files = list(labels_path.glob("*.txt"))
total_files = len(label_files)
file_counter = 0
patch_counter = 0

# Create output directories upfront
for d in output_npy_dirs.values():
    d.mkdir(parents=True, exist_ok=True)
for d in output_png_dirs.values():
    d.mkdir(parents=True, exist_ok=True)

for label_file in label_files:
    file_counter += 1
    image_file = images_path / f"{label_file.stem}.tif"
    if not image_file.exists():
        print(f"Warning: TIFF image not found for label: {label_file.stem}")
        continue

    with rasterio.open(image_file) as src:
        img_array = src.read()  # (bands, height, width)

    if img_array.shape[1] != IMG_SIZE or img_array.shape[2] != IMG_SIZE:
```

```
        print(f"Warning: unexpected image size for {image_file.name}: {img_array.shape[2]}x{img_ar-
ray.shape[1]}")
        continue

    with open(label_file, 'r') as f:
        lines = f.readlines()

    for idx, line in enumerate(lines):
        parts = line.strip().split()
        if len(parts) != 5:
            continue
        class_id, cx, cy, w, h = parts
        class_id = int(class_id)
        if class_id >= len(class_names):
            print(f"Warning: Unknown class_id {class_id} in {label_file.name}")
            continue
        class_name = class_names[class_id]

        cx, cy, w, h = map(float, (cx, cy, w, h))
        bbox_coords = get_bbox_pixel_coords(cx, cy, w, h)

        patch_center, patch_offset = extract_two_patches_inside_bbox(img_array, bbox_coords,
PATCH_SIZE)

        # Save center patch
        patch_center_npy_path = output_npy_dirs[class_name] / f"{label_file.stem}_{class_name}_cen-
ter_{idx}.npy"
        patch_center_png_path = output_png_dirs[class_name] / patch_center_npy_path.with_suf-
fix('.png').name

        patch_center_npy_path.parent.mkdir(parents=True, exist_ok=True)
        patch_center_png_path.parent.mkdir(parents=True, exist_ok=True)

        np.save(patch_center_npy_path, patch_center)
        Image.fromarray(patch_center[:, :, :3].astype(np.uint8)).save(patch_center_png_path)

        patch_records.append({
            'npy_filename': str(patch_center_npy_path),
            'png_filename': str(patch_center_png_path),
            'class': class_name,
            'patch_type': 'center'
        })
        patch_counter += 1

        # Save offset patch
        patch_offset_npy_path = output_npy_dirs[class_name] / f"{label_file.stem}_{class_name}_off-
set_{idx}.npy"
        patch_offset_png_path = output_png_dirs[class_name] / patch_offset_npy_path.with_suf-
fix('.png').name

        patch_offset_npy_path.parent.mkdir(parents=True, exist_ok=True)
        patch_offset_png_path.parent.mkdir(parents=True, exist_ok=True)

        np.save(patch_offset_npy_path, patch_offset)
        Image.fromarray(patch_offset[:, :, :3].astype(np.uint8)).save(patch_offset_png_path)

        patch_records.append({
            'npy_filename': str(patch_offset_npy_path),
            'png_filename': str(patch_offset_png_path),
            'class': class_name,
            'patch_type': 'offset'
        })
        patch_counter += 1

    if file_counter % 5 == 0 or file_counter == total_files:
        print(f"Processed {file_counter}/{total_files} label files, total patches created:
{patch_counter}")

print(f"Created {patch_counter} patches of size {PATCH_SIZE}x{PATCH_SIZE} with 4 bands (R,G,B,I)")

# Save patch info CSV
```

```
df_records = pd.DataFrame(patch_records)
csv_save_path = output_npy_base_dir / "patches_labels_2cl_rgbi.npy.csv"
df_records.to_csv(csv_save_path, index=False)
print(f"Patch records saved to {csv_save_path}")


#------------------------
# File count
#---------

# PATHS
patches_inspect_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_inspect")
patches_npy_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_npy")

def count_files(base_dir, ext):
    counts = {}
    for class_dir in base_dir.iterdir():
        if class_dir.is_dir():
            counts[class_dir.name] = len(list(class_dir.glob(f"*{ext}")))
    return counts

png_counts = count_files(patches_inspect_dir, ".png")
npy_counts = count_files(patches_npy_dir, ".npy")

print("Patch counts per class:")
print("Class       | PNG files | NPY files")
print("-----------------------------------")
all_classes = sorted(set(png_counts) | set(npy_counts))
for cls in all_classes:
    png_count = png_counts.get(cls, 0)
    npy_count = npy_counts.get(cls, 0)
    print(f"{cls:<11} | {png_count:9} | {npy_count:9}")

total_png = sum(png_counts.values())
total_npy = sum(npy_counts.values())
print("-----------------------------------")
print(f"Total PNG files: {total_png}")
print(f"Total NPY files: {total_npy}")


#------------------------
# Filter positive training data
#---------

np.random.seed(42)  # Fixed seed for reproducibility

# PATHS
patches_inspect_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_inspect")
patches_npy_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_npy")
training_output_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/filtered_training_patches")
training_output_dir.mkdir(parents=True, exist_ok=True)

# Define classes and their checked inspect folders
classes = {
    'old_ctp': 'old_ctp_checked',
    'young_ctp': 'young_ctp_checked'
}

max_samples_young = 1000

for cls, checked_folder in classes.items():
    inspected_dir = patches_inspect_base_dir / checked_folder
    npy_dir = patches_npy_base_dir / cls
    training_cls_dir = training_output_dir / cls
    training_cls_dir.mkdir(parents=True, exist_ok=True)
```

```
    # Get base names (without extensions) of verified PNG patches in checked folder
    verified_png_basenames = {f.stem for f in inspected_dir.glob("*.png")}

    # List all npy patches for this class and filter by verified PNG basenames
    matching_npy_files = [f for f in npy_dir.glob("*.npy") if f.stem in verified_png_basenames]

    print(f"Class '{cls}': found {len(matching_npy_files)} npy patches matching verified PNGs")

    # For young_ctp, randomly select 1000 patches if more available
    if cls == 'young_ctp' and len(matching_npy_files) > max_samples_young:
        selected_files = list(np.random.choice(matching_npy_files, max_samples_young, re-
place=False))
    else:
        selected_files = matching_npy_files

    # Copy selected npy files to training folder
    for npy_file in selected_files:
        shutil.copy(npy_file, training_cls_dir / npy_file.name)

    print(f"Class '{cls}': copied {len(selected_files)} filtered patches to training folder")

print("Filtered training patches selection and copying completed.")


#------------------------
# Negative training data
#---------

# PATHS
tiffs_folder = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/detection/run_01/train-
ing_data/negative")
output_patch_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_npy/negative")
output_patch_dir.mkdir(parents=True, exist_ok=True)

PATCH_SIZE = 64
IMG_SIZE = 640
NUM_PATCHES = 1000

tiff_files = list(tiffs_folder.glob("*.tif"))
if not tiff_files:
    raise Exception("No TIFF files found in directory!")

patch_records = []

for i in range(NUM_PATCHES):
    # Randomly select one TIFF file
    tiff_path = random.choice(tiff_files)
    with rasterio.open(tiff_path) as src:
        img_array = src.read()  # shape (4, H, W)

    # Random patch center ensuring full patch fits
    x_center = random.randint(PATCH_SIZE//2, IMG_SIZE - PATCH_SIZE//2)
    y_center = random.randint(PATCH_SIZE//2, IMG_SIZE - PATCH_SIZE//2)

    half = PATCH_SIZE // 2
    x1 = x_center - half
    y1 = y_center - half
    x2 = x1 + PATCH_SIZE
    y2 = y1 + PATCH_SIZE

    patch = img_array[:, y1:y2, x1:x2]  # shape (4, PATCH_SIZE, PATCH_SIZE)
    patch = np.transpose(patch, (1, 2, 0))  # to (PATCH_SIZE, PATCH_SIZE, 4)

    # Save patch as .npy file
    patch_filename = f"neg_rand_{i}_{tiff_path.stem}.npy"
    patch_save_path = output_patch_dir / patch_filename
    np.save(patch_save_path, patch)

    patch_records.append({'filename': str(patch_save_path), 'source_image': str(tiff_path)})
```

```
    if (i + 1) % 100 == 0 or i == NUM_PATCHES - 1:
        print(f"Created {i + 1}/{NUM_PATCHES} negative patches")

# Save metadata CSV
patches_csv_path = output_patch_dir / "negative_random_patches.csv"
df = pd.DataFrame(patch_records)
df.to_csv(patches_csv_path, index=False)

print(f"Completed creating {NUM_PATCHES} random negative patches. Metadata saved to
{patches_csv_path}")


#-------------------------------------------------------------------------------------
# 2. RF patch classification training
#-----------------------------------------

# Set random seed for reproducibility
RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)

# PATHS
class_paths = {
    'old_ctp': Path('/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/training_data_2cl/fil-
tered_training_patches/old_ctp'),
    'young_ctp': Path('/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/filtered_training_patches/young_ctp'),
    'negative': Path('/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/train-
ing_data_2cl/patches_npy/negative')
}

PATCH_SIZE = 64
NUM_CHANNELS = 4
NUM_CHANNELS_WITH_NDVI = 5  # RGBI + NDVI


X_list = []
y_list = []


for cls, folder_path in class_paths.items():
    npy_files = list(folder_path.glob('*.npy'))
    print(f"Loading {len(npy_files)} patches for class '{cls}'")

    for npy_path in tqdm(npy_files, desc=f'Loading {cls} patches'):
        patch = np.load(npy_path)
        if patch.shape != (PATCH_SIZE, PATCH_SIZE, NUM_CHANNELS):
            print(f"Skipping {npy_path.name}, unexpected shape {patch.shape}")
            continue

        # Calculate NDVI = (NIR - Red) / (NIR + Red)
        red = patch[:, :, 0].astype(np.float32)
        nir = patch[:, :, 3].astype(np.float32)
        denom = (nir + red)
        # Avoid divide by zero
        denom[denom == 0] = 1e-6
        ndvi = (nir - red) / denom

        # Append NDVI as new 5th channel
        patch_with_ndvi = np.dstack((patch, ndvi))

        # Flatten and store
        X_list.append(patch_with_ndvi.flatten())
        y_list.append(cls)


X = np.array(X_list)
y = np.array(y_list)


print(f"Total valid patches loaded: {len(X)}")
```

```
# Encode class labels as integers
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
print(f"Classes encoded as: {dict(zip(label_encoder.classes_, label_encoder.transform(label_en-
coder.classes_)))}")


# Split train/test stratified by class
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=RANDOM_SEED, stratify=y_encoded)


# Train Random Forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=RANDOM_SEED)
rf.fit(X_train, y_train)


# Predictions and evaluation
y_pred = rf.predict(X_test)
print("Classification report:")
print(classification_report(y_test, y_pred, target_names=label_encoder.classes_))
print("Accuracy:", accuracy_score(y_test, y_pred))


# Save model and label encoder
model_dir = Path('/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/trained_models')
model_dir.mkdir(parents=True, exist_ok=True)
model_path = model_dir / 'rf_model_rgbi_ndvi_"3"class.joblib'
joblib.dump(rf, model_path)
print(f"Random Forest model saved to {model_path}")


le_path = model_dir / 'label_encoder.joblib'
joblib.dump(label_encoder, le_path)
print(f"Label encoder saved to {le_path}")

#-------------------------------------------------------------------------------------
# 3. Preprocess inference dataset
#-----------------------------------------

# PATHS
tiff_folder = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/EG_random_reproducible/batch-
1_3000_sample")
output_patches_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/predic-
tion_patches_b1-sample_NDVI")
output_patches_dir.mkdir(parents=True, exist_ok=True)

PATCH_SIZE = 64

patch_records = []
skipped_images = []

tiff_files = list(tiff_folder.glob("*.tif"))

for tiff_path in tqdm(tiff_files, desc="Processing GeoTIFFs"):
    try:
        with rasterio.open(tiff_path) as src:
            img_array = src.read()  # shape: (bands, height, width)
    except Exception as e:
        print(f"Skipping {tiff_path.name} due to read error: {e}")
        skipped_images.append({'image_name': tiff_path.name, 'reason': f'read error: {e}'})
        continue

    _, img_height, img_width = img_array.shape

    if img_height < PATCH_SIZE or img_width < PATCH_SIZE:
        print(f"Skipping {tiff_path.name}, image too small: {img_width}x{img_height}")
        skipped_images.append({'image_name': tiff_path.name, 'reason': 'too small'})
        continue
```

```
    patches_per_row = img_width // PATCH_SIZE
    patches_per_col = img_height // PATCH_SIZE

    img_patch_dir = output_patches_dir / tiff_path.stem
    img_patch_dir.mkdir(parents=True, exist_ok=True)

    patch_idx = 0
    for row in range(patches_per_col):
        for col in range(patches_per_row):
            x1 = col * PATCH_SIZE
            y1 = row * PATCH_SIZE
            x2 = x1 + PATCH_SIZE
            y2 = y1 + PATCH_SIZE

            patch = img_array[:, y1:y2, x1:x2]
            patch = np.transpose(patch, (1, 2, 0))  # (H, W, Bands)

            if patch.shape[2] < 4:
                print(f"Skipping patch {patch_idx} in {tiff_path.name}, less than 4 channels.")
                continue

            # Calculate NDVI = (NIR - Red) / (NIR + Red)
            red = patch[:, :, 0].astype(np.float32)
            nir = patch[:, :, 3].astype(np.float32)
            denom = (nir + red)
            denom[denom == 0] = 1e-6  # Avoid division by zero
            ndvi = (nir - red) / denom

            # Append NDVI as 5th channel
            patch_with_ndvi = np.dstack((patch, ndvi))

            patch_filename = f"{tiff_path.stem}_patch_{patch_idx}.npy"
            patch_save_path = img_patch_dir / patch_filename
            np.save(patch_save_path, patch_with_ndvi)

            patch_records.append({
                'image_name': tiff_path.name,
                'patch_filename': str(patch_save_path),
                'patch_index': patch_idx,
                'bbox_left': x1,
                'bbox_upper': y1,
                'bbox_right': x2,
                'bbox_lower': y2,
                'patch_height': patch_with_ndvi.shape[0],
                'patch_width': patch_with_ndvi.shape[1],
                'channels': patch_with_ndvi.shape[2]
            })

            patch_idx += 1

# Save metadata CSV
metadata_csv_path = output_patches_dir / "patches_metadata.csv"
df = pd.DataFrame(patch_records)
df.to_csv(metadata_csv_path, index=False)

# Save skipped images CSV
skipped_csv_path = output_patches_dir / "skipped_images.csv"
df_skipped = pd.DataFrame(skipped_images)
df_skipped.to_csv(skipped_csv_path, index=False)

print(f"Patch extraction with NDVI completed for {len(tiff_files) - len(skipped_images)} images.")
print(f"Skipped {len(skipped_images)} images, details saved to {skipped_csv_path}")
print(f"Patch metadata saved to {metadata_csv_path}")


#-------------------------------------------------------------------------------------
# 4. RF patch classification inference
#-----------------------------------------

# PATHS
```

```
patches_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/predic-
tion_patches_b1-sample_NDVI")
model_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/trained_mod-
els/rf_model_rgbi_ndvi_2class.joblib")
output_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/predictions_2-clas-
ses_NDVI")
output_dir.mkdir(parents=True, exist_ok=True)

PATCH_SIZE = 64
NUM_CHANNELS = 5  # RGBI + NDVI

# Load trained model once globally
rf = joblib.load(model_path)
classes = rf.classes_
class_zero_index = 0

def process_one_image(img_folder_path):
    npy_files = sorted(img_folder_path.glob("*.npy"))
    if not npy_files:
        return [], img_folder_path.name, False

    patches = []
    for p in npy_files:
        patch = np.load(p)
        if patch.shape != (PATCH_SIZE, PATCH_SIZE, NUM_CHANNELS):
            patches.append(None)
            continue

        # Use the precomputed patch with NDVI included
        patches.append(patch)

    valid_indices = [i for i, p in enumerate(patches) if p is not None]
    valid_patches = [patches[i] for i in valid_indices]

    if not valid_patches:
        return [], img_folder_path.name, False

    features = np.array([p.flatten() for p in valid_patches])

    preds = rf.predict(features)
    probs = rf.predict_proba(features)

    results = []
    for idx, patch_idx in enumerate(valid_indices):
        pred_class = classes[preds[idx]]
        conf = probs[idx][preds[idx]]

        patch_name = npy_files[patch_idx].stem
        patch_num = patch_name.split('_')[-1]
        try:
            patch_num = int(patch_num)
        except:
            patch_num = idx

        results.append({
            'image_folder': img_folder_path.name,
            'patch_filename': npy_files[patch_idx].name,
            'patch_index': patch_num,
            'prediction': pred_class,
            'confidence': conf
        })

    return results, img_folder_path.name, preds.tolist()

# List all image folders (one folder per original image)
image_folders = [f for f in patches_base_dir.iterdir() if f.is_dir()]

detailed_results = []
image_patch_preds = {}

with ProcessPoolExecutor(max_workers=4) as executor:
```

```
    futures = [executor.submit(process_one_image, img_folder) for img_folder in image_folders]
    for future in tqdm(futures, desc="Processing images"):
        patch_results, folder_name, preds = future.result()
        detailed_results.extend(patch_results)
        image_patch_preds[folder_name] = preds

# Save detailed patch-level predictions
detailed_csv_path = output_dir / "npy_cls_2cl_NDVI.csv"
pd.DataFrame(detailed_results).to_csv(detailed_csv_path, index=False)

# Generate summary CSVs with different positive patch thresholds
summary_thresholds = {
    '1_positive_patch_2cl_NDVI': 1,
    '10_positive_patches_2cl_NDVI': 10,
    '20_positive_patches_2cl_NDVI': 20,
}

for summary_name, threshold in summary_thresholds.items():
    summary_list = []
    for image_name, preds in image_patch_preds.items():
        # Count how many patches predicted are NOT the class encoded as '0' (= negative)
        non_zero_count = sum(p != class_zero_index for p in preds)
        is_positive = non_zero_count >= threshold
        summary_list.append({
            'image_folder': image_name,
            'non_zero_patch_count': non_zero_count,
            'summary_prediction': 'positive' if is_positive else 'negative'
        })

    summary_csv_path = output_dir / f"npy_cls_2cl_NDVI_{summary_name}.csv"
    pd.DataFrame(summary_list).to_csv(summary_csv_path, index=False)
    print(f"Summary saved to: {summary_csv_path}")


#-------------------------------------------------------------------------------------------
# 5. CSV to Geopackage
#-------------------------------------------

# PATHS
pred_csv_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/predictions_2-clas-
ses_NDVI/npy_cls_2cl_NDVI.csv")
meta_csv_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/prediction_patches_b1-
sample_NDVI/patches_metadata.csv")
gpkg_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/geopack-
ages/npy_cls_2cl_NDVI.csv.gpkg")
original_geotiffs_dir = Path("/content/drive/MyDrive/WeiBauDe-
tect/Daten/yolov12/EG_random_reproducible/batch-1_3000_sample")

# Load CSVs
meta_df = pd.read_csv(meta_csv_path)
pred_df = pd.read_csv(pred_csv_path)

# Normalize patch_filename in both DataFrames for robust merging
meta_df['patch_filename'] = meta_df['patch_filename'].apply(lambda x: Path(x).name)
pred_df['patch_filename'] = pred_df['patch_filename'].apply(lambda x: Path(x).name)

# Merge on patch_filename
merged_df = meta_df.merge(pred_df, on='patch_filename', how='inner')

# Load GeoTIFFs into dict
raster_dict = {tif_path.name: rasterio.open(tif_path) for tif_path in original_geo-
tiffs_dir.glob("*.tif")}

def get_patch_bbox(row):
    tif_name = row['image_name']
    raster = raster_dict.get(tif_name)
    if raster is None:
        return None
    x_left, y_upper = row['bbox_left'], row['bbox_upper']
    x_right, y_lower = row['bbox_right'], row['bbox_lower']
    top_left = raster.transform * (x_left, y_upper)
```

```
    bottom_right = raster.transform * (x_right, y_lower)
    minx, maxx = top_left[0], bottom_right[0]
    miny, maxy = bottom_right[1], top_left[1]
    return box(minx, miny, maxx, maxy)

# Create geometry column
merged_df['geometry'] = merged_df.apply(get_patch_bbox, axis=1)

# Remove rows without valid geometry
merged_df = merged_df.dropna(subset=['geometry']).reset_index(drop=True)

# Convert to GeoDataFrame and set CRS from rasters
gdf = gpd.GeoDataFrame(merged_df, geometry='geometry')
if raster_dict:
    gdf.set_crs(next(iter(raster_dict.values())).crs, inplace=True)

# Save to GeoPackage
gdf.to_file(gpkg_path, driver='GPKG')

print(f"GeoPackage saved at {gpkg_path}")
print(gdf.head())
```

III.VIII RF segmentation classification (RF_Segmentation; Google Colab)

```
#------------------------
#Installations
#---------
!pip install scikit-image
!pip install rasterio
!pip install scikit-learn
!pip install geopandas

#------------------------
#Libraries
#---------
import os
import rasterio
import numpy as np
from pathlib import Path
from PIL import Image
from skimage.segmentation import slic
from collections import defaultdict, Counter
import time
import shutil
import builtins
from skimage.measure import regionprops, label
from skimage.feature import graycomatrix, graycoprops
import pandas as pd
import difflib
import random
import json
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
import joblib
import matplotlib.pyplot as plt
import concurrent.futures
from tqdm import tqdm
import geopandas as gpd
from shapely.geometry import box

# Mount Google Drive
drive.mount('/content/drive')

print("Setup complete. Starting dataset preparation...")

#-------------------------------------------------------------------------------------------
# 1. RF segmentation dataset setup
#-----------------------------------------

# PATHS
base_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten")
images_path = base_path / "yolov12/detection/run_02/training_data/positive_images"
positive_labels_path = base_path / "yolov12/detection/run_02/training_data/positive_labels"
output_base_dir = base_path / "RandomForest/Segmentation/training_data_pos"

# Create output subdirectories for each class
class_list = ["ctp", "negative"]
for c in class_list:
    (output_base_dir / c).mkdir(parents=True, exist_ok=True)


# Class map for YOLO labels (all 0s in bounding box files mean "ctp")
class_map = {
    0: "ctp",
}


def read_yolo_labels(label_file, class_map, img_width, img_height):
    bboxes = []
```

```
    with open(label_file, "r") as f:
        for line in f:
            parts = line.strip().split()
            if len(parts) != 5:
                continue
            class_idx = int(parts[0])
            center_x = float(parts[1]) * img_width
            center_y = float(parts[2]) * img_height
            box_width = float(parts[3]) * img_width
            box_height = float(parts[4]) * img_height


            x_min = max(0, center_x - box_width / 2)
            x_max = min(img_width, center_x + box_width / 2)
            y_min = max(0, center_y - box_height / 2)
            y_max = min(img_height, center_y + box_height / 2)


            class_label = class_map.get(class_idx, "unknown")


            bboxes.append({
                "x_min": x_min,
                "y_min": y_min,
                "x_max": x_max,
                "y_max": y_max,
                "class_label": class_label,
            })
    return bboxes


def calculate_ndvi(image):
    red = image[:, :, 0].astype(float)
    nir = image[:, :, 3].astype(float)
    ndvi = (nir - red) / (nir + red + 1e-10)
    ndvi = np.clip(ndvi, -1, 1)
    return ndvi


def segment_image(image, n_segments=20, compactness=1):
    return slic(image, n_segments=n_segments, compactness=compactness, start_label=1)


def save_segment(image_array, segment_mask, path):
    masked_img = np.zeros_like(image_array)
    for b in range(image_array.shape[2]):
        band = image_array[:, :, b]
        masked_img[:, :, b] = np.where(segment_mask, band, 0)
    img_norm = ((masked_img - masked_img.min()) / (np.ptp(masked_img) + 1e-10) *
255).astype(np.uint8)
    Image.fromarray(img_norm[:, :, :3]).save(path)


def save_mask(mask_array, profile, save_path):
    mask_uint8 = mask_array.astype(np.uint8) * 255
    profile_mask = profile.copy()
    profile_mask.update(dtype=rasterio.uint8, count=1, compress='lzw', height=mask_uint8.shape[0],
width=mask_uint8.shape[1])
    with rasterio.open(save_path, 'w', **profile_mask) as dst:
        dst.write(mask_uint8, 1)


def bbox_to_pixels(xmin, ymin, xmax, ymax, transform):
    col_min, row_min = ~transform * (xmin, ymax)
    col_max, row_max = ~transform * (xmax, ymin)
    return int(row_min), int(row_max), int(col_min), int(col_max)


image_files = [f for f in os.listdir(images_path) if f.endswith('.tif')]
```

```
total_files = len(image_files)

start_time = time.time()
for idx, img_file in enumerate(image_files, 1):
    img_path = images_path / img_file
    label_file = positive_labels_path / img_file.replace('.tif', '.txt')

    if not label_file.exists():
        print(f"[{idx}/{total_files}] Warning: Label file missing for {img_file}, skipping")
        continue

    with rasterio.open(img_path) as src:
        image = src.read().transpose(1, 2, 0)
        transform = src.transform
        profile = src.profile
        img_height, img_width = image.shape[:2]

    bboxes = read_yolo_labels(label_file, class_map, img_width, img_height)
    ndvi = calculate_ndvi(image)
    image_with_ndvi = np.dstack((image, ndvi))
    segments = segment_image(image_with_ndvi, n_segments=20, compactness=1)

    segment_labels = defaultdict(list)
    for bbox in bboxes:
        rmin, rmax, cmin, cmax = bbox_to_pixels(bbox['x_min'], bbox['y_min'], bbox['x_max'],
bbox['y_max'], transform)
        rmin, rmax = max(0, min(rmin, rmax)), max(rmin, rmax)
        cmin, cmax = max(0, min(cmin, cmax)), max(cmin, cmax)
        overlapping_segments = np.unique(segments[rmin:rmax, cmin:cmax])
        for seg_id in overlapping_segments:
            segment_labels[seg_id].append("ctp")

    final_labels = {}
    for seg_id in np.unique(segments):
        if seg_id in segment_labels:
            final_labels[seg_id] = "ctp"
        else:
            final_labels[seg_id] = "negative"

    for seg_id, label in final_labels.items():
        mask = (segments == seg_id)
        segment_file_name = f"{img_file}_segment_{seg_id}.png"
        mask_file_name = f"{img_file}_segment_{seg_id}_mask.tif"
        save_segment(image_with_ndvi, mask, output_base_dir / label / segment_file_name)
        save_mask(mask, profile, output_base_dir / label / mask_file_name)

    elapsed = time.time() - start_time
    print(f"[{idx}/{total_files}] Processed {img_file} in {elapsed:.2f}s, saved {len(final_labels)}
segments")

print(f"All files processed in {time.time() - start_time:.2f}s")


#------------------------
# Negative training data
#---------

# PATHS
negative_images_path = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/detec-
tion/run_01/training_data/negative")
output_base_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/train-
ing_data_neg")

# Create output folder for negative masks
(output_base_dir / "negative").mkdir(parents=True, exist_ok=True)

def calculate_ndvi(image):
    red = image[:, :, 0].astype(float)
    nir = image[:, :, 3].astype(float)
    ndvi = (nir - red) / (nir + red + 1e-10)
    ndvi = np.clip(ndvi, -1, 1)
```

```
    return ndvi

def segment_image(image, n_segments=20, compactness=1):
    return slic(image, n_segments=n_segments, compactness=compactness, start_label=1)

def save_mask(mask_array, profile, save_path):
    mask_uint8 = mask_array.astype(np.uint8) * 255
    profile_mask = profile.copy()
    profile_mask.update(dtype=rasterio.uint8, count=1, compress='lzw', height=mask_uint8.shape[0],
width=mask_uint8.shape[1])
    with rasterio.open(save_path, 'w', **profile_mask) as dst:
        dst.write(mask_uint8, 1)

image_files = [f for f in os.listdir(negative_images_path) if f.endswith('.tif')]
total_files = len(image_files)

start_time = time.time()
for idx, img_file in enumerate(image_files, 1):
    file_start = time.time()

    img_path = negative_images_path / img_file
    with rasterio.open(img_path) as src:
        image = src.read().transpose(1, 2, 0)
        profile = src.profile

    ndvi = calculate_ndvi(image)
    image_with_ndvi = np.dstack((image, ndvi))
    segments = segment_image(image_with_ndvi, n_segments=20, compactness=1)

    unique_segments = np.unique(segments)
    for seg_id in unique_segments:
        mask = (segments == seg_id)
        mask_file_name = f"{img_file}_segment_{seg_id}_mask.tif"
        save_mask(mask, profile, output_base_dir / "negative" / mask_file_name)

    elapsed = time.time() - file_start
    total_elapsed = time.time() - start_time
    print(f"[{idx}/{total_files}] Processed {img_file} in {elapsed:.2f}s, total elapsed: {to-
tal_elapsed:.2f}s, saved {len(unique_segments)} negative masks")

print(f"All negative images processed and masks saved in {output_base_dir / 'negative'}")


#------------------------
# Training data Mask annotation after manual inspection
#---------

# === INPUT FOLDERS ===
checked_ctp_pngs = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmenta-
tion/ctp_png_inspect_checked")
all_negative_masks_1 = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmenta-
tion/training_data_pos/negative")
all_negative_masks_2 = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmenta-
tion/training_data_neg/negative")

# === OUTPUT FOLDERS ===
ctp_out = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/fi-
nal_masks_training/ctp")
neg_out = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/fi-
nal_masks_training/negative")

ctp_out.mkdir(parents=True, exist_ok=True)
neg_out.mkdir(parents=True, exist_ok=True)

# === BUILD LIST OF MANUALLY CONFIRMED CTP SEGMENTS ===
confirmed_ctp = set(p.stem for p in checked_ctp_pngs.glob("*.png"))
print(f"Found {len(confirmed_ctp)} manually confirmed CTP segments.\n")

# === COMBINE ALL MASK FOLDERS ===
mask_sources = [all_negative_masks_1, all_negative_masks_2]
```

```
total_ctp, total_neg = 0, 0
start_time = time.time()

for folder_idx, folder in enumerate(mask_sources, 1):
    mask_files = sorted(list(folder.glob("*.tif")))
    total_masks = len(mask_files)
    print(f"[{folder_idx}/{len(mask_sources)}] Processing folder: {folder} ({total_masks} masks)")

    for i, mask_path in enumerate(mask_files, 1):
        # This removes "_mask" suffix to match PNG stem: "WBK_1035_seg-1.tif_segment_8"
        mask_stem = mask_path.stem.replace("_mask", "")

        t0 = time.time()

        if mask_stem in confirmed_ctp:
            shutil.copy(mask_path, ctp_out / mask_path.name)
            total_ctp += 1
            status = "CTP"
        else:
            shutil.copy(mask_path, neg_out / mask_path.name)
            total_neg += 1
            status = "NEGATIVE"

        elapsed = time.time() - t0
        overall_elapsed = time.time() - start_time
        print(f"  [{i}/{total_masks}] {mask_path.name} → {status} "
              f"({elapsed:.2f}s, total {overall_elapsed/60:.2f} min)")

    print(f"Completed folder {folder_idx}/{len(mask_sources)}.")

print("\n=== SUMMARY ===")
print(f"CTP masks copied: {total_ctp}")
print(f"Negative masks copied: {total_neg}")
print(f"Processing time: {time.time() - start_time:.2f} seconds")
print(f"CTP masks stored in: {ctp_out}")
print(f"Negative masks stored in: {neg_out}")

#------------------------
# Extract vector features per segment
#---------

# PATHS
mask_root_ctp = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/fi-
nal_masks_training/ctp")
mask_root_neg = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/fi-
nal_masks_training/negative")

img_roots = [
    Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/detection/run_02/training_data/posi-
tive_images"),
    Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/detection/run_01/training_data/nega-
tive"),
]

def extract_features(mask_path, img_path):
    try:
        mask = rasterio.open(builtins.str(mask_path)).read(1) > 0
        with rasterio.open(builtins.str(img_path)) as src:
            img = src.read().transpose(1,2,0)
    except Exception as e:
        return None, f"Raster read error: {e}"

    features = {}

    # NDVI if four bands available
    ndvi = None
    if img.shape[2] >= 4:
        try:
            red, nir = img[..., 0].astype(float), img[..., 3].astype(float)
            ndvi = (nir - red) / (nir + red + 1e-10)
            ndvi = np.clip(ndvi, -1, 1)
```

```
        except Exception as e:
            features["ndvi_error"] = builtins.str(e)

    # Spectral features
    for b in range(img.shape[2]):
        vals = img[..., b][mask]
        if vals.size == 0:
            continue
        features.update({
            f"mean_band_{b}": np.mean(vals),
            f"std_band_{b}": np.std(vals),
            f"min_band_{b}": np.min(vals),
            f"max_band_{b}": np.max(vals),
            f"median_band_{b}": np.median(vals)
        })

    # NDVI stats
    if ndvi is not None:
        vals = ndvi[mask]
        if vals.size > 0:
            features.update({
                "ndvi_mean": np.mean(vals),
                "ndvi_std": np.std(vals),
                "ndvi_min": np.min(vals),
                "ndvi_max": np.max(vals),
                "ndvi_median": np.median(vals)
            })

    # Texture (GLCM) on NDVI
    try:
        if ndvi is not None:
            ndvi_scaled = (255 * (ndvi - ndvi.min()) / (ndvi.ptp() + 1e-10)).astype(np.uint8)
            glcm = graycomatrix(ndvi_scaled, [2], [0], 256, symmetric=True, normed=True)
            for prop in ["contrast", "homogeneity", "energy", "dissimilarity"]:
                features[f"glcm_{prop}"] = float(graycoprops(glcm, prop))
    except Exception as e:
        features["glcm_error"] = builtins.str(e)

    # Geometric features
    try:
        lp = label(mask)
        rp = regionprops(lp)
        if rp:
            p = rp[0]
            features.update({
                "area": p.area,
                "perimeter": p.perimeter,
                "bbox_area": p.bbox_area,
                "compactness": (p.perimeter**2) / (4 * np.pi * p.area) if p.area > 0 else 0,
                "eccentricity": p.eccentricity,
                "solidity": p.solidity,
                "extent": p.extent,
            })
    except Exception as e:
        features["geometry_error"] = builtins.str(e)

    return features, None

# === BUILD IMAGE INDEX ===
print("Indexing all GeoTIFFs recursively...")
all_images = {}
for root in img_roots:
    for img_path in root.rglob("*.tif"):
        key = img_path.name.replace("-", "_").lower().strip()
        all_images[key] = img_path
print(f"Indexed {len(all_images)} total GeoTIFFs.\n")

# === RANDOM SAMPLING ===
MAX_SAMPLES = 12500

all_ctp_masks = list(mask_root_ctp.glob("*.tif"))
```

```
all_neg_masks = list(mask_root_neg.glob("*.tif"))

sampled_ctp = random.sample(all_ctp_masks, min(len(all_ctp_masks), MAX_SAMPLES))
sampled_neg = random.sample(all_neg_masks, min(len(all_neg_masks), MAX_SAMPLES))

# Prepare for incremental saving
all_rows, failed_rows = [], []
counts = {"ctp": 0, "negative": 0}
start_time = time.time()

def save_intermediate_results(filename, data):
    os.makedirs(os.path.dirname(filename), exist_ok=True)
    pd.DataFrame(data).to_csv(filename, index=False)
    print(f"Intermediate results saved to {filename}")

# Process CTP in two halves with intermediate saving
half_ctp = len(sampled_ctp) // 2

print(f"Processing {len(sampled_ctp)} sampled CTP masks in two batches...")

for batch, mask_batch in enumerate([sampled_ctp[:half_ctp], sampled_ctp[half_ctp:]]):
    for i, mask_path in enumerate(mask_batch, 1):
        base_name = mask_path.name.split("_segment")[0] + ".tif"
        normalized_base = base_name.replace("-", "_").lower().strip()

        matched_path = None
        if normalized_base in all_images:
            matched_path = all_images[normalized_base]
        else:
            possible = difflib.get_close_matches(normalized_base, list(all_images.keys()), n=1,
cutoff=0.7)
            if possible:
                matched_path = all_images[possible[0]]

        if matched_path:
            features, error = extract_features(mask_path, matched_path)
            if features and not error:
                features["label"] = 1
                features["mask_file"] = str(mask_path)
                features["source_img"] = str(matched_path)
                all_rows.append(features)
                counts["ctp"] += 1
            else:
                failed_rows.append({"mask": str(mask_path), "error": error or "Empty features"})
        else:
            failed_rows.append({"mask": str(mask_path), "error": "No matching GeoTIFF found"})

        if i % 100 == 0 or i == len(mask_batch):
            elapsed = (time.time() - start_time) / 60
            print(f"  [{i}/{len(mask_batch)}] processed in batch {batch+1}... Elapsed {elapsed:.2f}
min")

    # Save after each half
    save_intermediate_results("/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_training_features_partial_ctp.csv", all_rows)
    save_intermediate_results("/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_failed_masks_partial_ctp.csv", failed_rows)


# Process Negative (in 2 halves)
half_neg = len(sampled_neg) // 2
print(f"\nProcessing {len(sampled_neg)} sampled NEGATIVE masks in two batches...")

for batch, mask_batch in enumerate([sampled_neg[:half_neg], sampled_neg[half_neg:]]):
    for i, mask_path in enumerate(mask_batch, 1):
        base_name = mask_path.name.split("_segment")[0] + ".tif"
        normalized_base = base_name.replace("-", "_").lower().strip()

        matched_path = None
        if normalized_base in all_images:
            matched_path = all_images[normalized_base]
```

```
        else:
            possible = difflib.get_close_matches(normalized_base, list(all_images.keys()), n=1,
cutoff=0.7)
            if possible:
                matched_path = all_images[possible[0]]

        if matched_path:
            features, error = extract_features(mask_path, matched_path)
            if features and not error:
                features["label"] = 0
                features["mask_file"] = str(mask_path)
                features["source_img"] = str(matched_path)
                all_rows.append(features)
                counts["negative"] += 1
            else:
                failed_rows.append({"mask": str(mask_path), "error": error or "Empty features"})
        else:
            failed_rows.append({"mask": str(mask_path), "error": "No matching GeoTIFF found"})

        if i % 100 == 0 or i == len(mask_batch):
            elapsed = (time.time() - start_time) / 60
            print(f"  [{i}/{len(mask_batch)}] processed in batch {batch+1}... Elapsed {elapsed:.2f}
min")

    save_intermediate_results("/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_training_features_partial_negative.csv", all_rows)
    save_intermediate_results("/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_failed_masks_partial_negative.csv", failed_rows)


# Save final combined CSV files
final_features_path = "/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_training_features.csv"
final_failed_path = "/content/drive/MyDrive/WeiBauDetect/RandomForest/Segmentation/Out-
puts/rf_failed_masks.csv"
os.makedirs(os.path.dirname(final_features_path), exist_ok=True)
pd.DataFrame(all_rows).to_csv(final_features_path, index=False)
pd.DataFrame(failed_rows).to_csv(final_failed_path, index=False)

print("\nProcessing complete!")
print(f"Final feature table saved to {final_features_path}")
print(f"Final failed masks log saved to {final_failed_path}")
print(f"Total processed: CTP = {counts['ctp']}, Negative = {counts['negative']}")


#------------------------
# Combine training data
#---------

# Set base path for the CSV files
base_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/segment_fea-
tures/Outputs"

# Define file paths
files = {
    "partial_ctp": f"{base_path}/rf_training_features_partial_ctp.csv",
    "partial_negative": f"{base_path}/rf_training_features_partial_negative.csv",
}

# Load the CSV files
partial_ctp = pd.read_csv(files["partial_ctp"])
partial_negative = pd.read_csv(files["partial_negative"])

# Concatenate all safely filtered dataframes
combined_df = pd.concat([partial_ctp, partial_negative], ignore_index=True)

# Save combined dataframe to new CSV
final_output_path = f"{base_path}/rf_training_features_combined.csv"
combined_df.to_csv(final_output_path, index=False)

print(f"Combined CSV saved with {len(combined_df)} rows to: {final_output_path}")
```

```
#-------------------------------------------------------------------------------------------
# 2. RF segmentation classification training with Hyperparameter Search
#-------------------------------------------

# Load dataset
df = pd.read_csv("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/segment_fea-
tures/Outputs/rf_training_features_combined.csv")

# Columns to drop (non-features)
drop_columns = [
    "label", "mask_file", "source_img", "glcm_error"
]

# Prepare feature matrix X and target y
X = df.drop(columns=drop_columns, errors='ignore')
y = df["label"]

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split dataset (80% train, 20% test), stratified by label
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y, test_size=0.2, random_state=42, stratify=y
)

# Hyperparameter grid for Random Forest
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [15, 30, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 5],
    'max_features': ['sqrt', 'log2']
}

# Initialize base Random Forest classifier
rf_base = RandomForestClassifier(random_state=42, n_jobs=-1)

# GridSearchCV for hyperparameter tuning
grid_search = GridSearchCV(
    rf_base,
    param_grid,
    cv=3,
    scoring='accuracy',
    n_jobs=-1
)

print("Starting hyperparameter tuning...")
grid_search.fit(X_train, y_train)

best_rf = grid_search.best_estimator_
print("Best Parameters:", grid_search.best_params_)

# Train final model with best parameters
best_rf.fit(X_train, y_train)

# Predictions on test set
y_pred = best_rf.predict(X_test)

# Evaluation
print("Classification Report:\n", classification_report(y_test, y_pred))
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))

# Plot feature importance (top 15 features)
feat_importances = pd.Series(best_rf.feature_importances_, index=X.columns)
feat_importances.nlargest(15).plot(kind='barh')
plt.title("Top 15 Feature Importances")
plt.show()
```

```
# Save the trained model
model_save_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/rf_model_segmentation.pkl"
joblib.dump(best_rf, model_save_path)
print(f"Random Forest model saved at {model_save_path}")

# Save the scaler to ensure consistent scaling during prediction
scaler_save_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/scaler_segmentation.pkl"
joblib.dump(scaler, scaler_save_path)
print(f"Scaler saved at {scaler_save_path}")

# Save the exact feature column names used for training (for prediction consistency)
feature_cols = X.columns.tolist()
feature_cols_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/feature_columns.json"
with open(feature_cols_path, "w") as f:
    json.dump(feature_cols, f)
print(f"Training feature columns saved at {feature_cols_path}")


#-----------------------------------------------------------------------------------------
# 3. Preprocess inference dataset
#-----------------------------------------

# PATHS
input_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/EG_random_reproducible/batch-
1_3000_sample")
output_features_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmenta-
tion/prediction_features")
output_features_dir.mkdir(parents=True, exist_ok=True)

def calculate_ndvi(image):
    red = image[:, :, 0].astype(float)
    nir = image[:, :, 3].astype(float)
    ndvi = (nir - red) / (nir + red + 1e-10)
    ndvi = np.clip(ndvi, -1, 1)
    return ndvi

def extract_features_in_memory(segment_mask, image):
    features = {}

    if image.shape[2] >= 4:
        red, nir = image[..., 0].astype(float), image[..., 3].astype(float)
        ndvi = (nir - red) / (nir + red + 1e-10)
        ndvi = np.clip(ndvi, -1, 1)
    else:
        ndvi = None

    for b in range(image.shape[2]):
        vals = image[..., b][segment_mask]
        if vals.size == 0:
            continue
        features.update({
            f"mean_band_{b}": np.mean(vals),
            f"std_band_{b}": np.std(vals),
            f"min_band_{b}": np.min(vals),
            f"max_band_{b}": np.max(vals),
            f"median_band_{b}": np.median(vals)
        })

    if ndvi is not None:
        vals = ndvi[segment_mask]
        if vals.size > 0:
            features.update({
                "ndvi_mean": np.mean(vals),
                "ndvi_std": np.std(vals),
                "ndvi_min": np.min(vals),
                "ndvi_max": np.max(vals),
                "ndvi_median": np.median(vals),
            })
```

```
            try:
                ndvi_scaled = (255 * (ndvi - ndvi.min()) / (ndvi.ptp() + 1e-10)).astype(np.uint8)
                glcm = graycomatrix(ndvi_scaled, [2], [0], 256, symmetric=True, normed=True)
                for prop in ["contrast", "homogeneity", "energy", "dissimilarity"]:
                    features[f"glcm_{prop}"] = float(graycoprops(glcm, prop))
            except Exception:
                pass

    lp = label(segment_mask)
    rp = regionprops(lp)
    if rp:
        p = rp[0]
        features.update({
            "area": p.area,
            "perimeter": p.perimeter,
            "bbox_area": p.bbox_area,
            "compactness": (p.perimeter ** 2) / (4 * np.pi * p.area) if p.area > 0 else 0,
            "eccentricity": p.eccentricity,
            "solidity": p.solidity,
            "extent": p.extent,
        })

    return features

def process_image(img_path):
    try:
        with rasterio.open(str(img_path)) as src:
            image = src.read().transpose(1, 2, 0)

        ndvi = calculate_ndvi(image)
        image_with_ndvi = np.dstack((image, ndvi))

        segments = slic(image_with_ndvi, n_segments=20, compactness=1, start_label=1)
        unique_segments = np.unique(segments)

        results = []
        for seg_id in unique_segments:
            segment_mask = (segments == seg_id)
            features = extract_features_in_memory(segment_mask, image_with_ndvi)
            features.update({
                "source_img": str(img_path),
                "segment_id": seg_id
            })
            results.append(features)

        return results
    except Exception as e:
        return f"Error processing {img_path.name}: {e}"

# Main parallel execution
start_time = time.time()
files = sorted(list(input_dir.glob("*.tif")))

all_results = []

# Adjust max_workers
max_workers = 8
import concurrent.futures
with concurrent.futures.ThreadPoolExecutor(max_workers=max_workers) as executor:
    futures = [executor.submit(process_image, img_path) for img_path in files]
    for idx, future in enumerate(concurrent.futures.as_completed(futures), 1):
        res = future.result()
        if isinstance(res, list):
            all_results.extend(res)
        else:
            print(res)
        if idx % 30 == 0:
            # Save intermediate
            df_intermediate = pd.DataFrame(all_results)
            intermediate_path = output_features_dir / "prediction_features.csv"
```

```
            df_intermediate.to_csv(intermediate_path, index=False)
            print(f"Intermediate save of features at image {idx}, saved to {intermediate_path}")

elapsed = time.time() - start_time

# Final save
features_df = pd.DataFrame(all_results)
final_path = output_features_dir / "prediction_features_parallel.csv"
features_df.to_csv(final_path, index=False)

print(f"Feature extraction completed in {elapsed / 60:.2f} minutes, saved to {final_path}")


#-------------------------------------------------------------------------------------
# 4. RF patch classification inference
#-----------------------------------------

# PATHS
prediction_csv = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/predic-
tion_features"
final_csv = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/results/b1-sam-
ple_cls_seg.csv"
model_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/rf_model_segmentation.pkl"
scaler_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/scaler_segmentation.pkl"
feature_cols_path = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/mod-
els/feature_columns.json"

# Load the intermediate CSV
df = pd.read_csv(prediction_csv)

# Load trained model and scaler
rf_model = joblib.load(model_path)
scaler = joblib.load(scaler_path)

# Load the training feature columns
with open(feature_cols_path, "r") as f:
    feature_cols = json.load(f)

# Select only the features used in training, in correct order
X = df[feature_cols]

# Scale features with saved scaler
print("Scaling features...")
X_scaled = scaler.transform(X)

# Batch prediction with tracking
batch_size = 5000
predictions = []
print("Starting predictions...")
for start in tqdm(range(0, len(X_scaled), batch_size), desc="Classifying segments"):
    end = min(start + batch_size, len(X_scaled))
    batch_pred = rf_model.predict(X_scaled[start:end])
    predictions.extend(batch_pred)

# Save predictions to dataframe
df['predicted_label'] = predictions

# Save final CSV to Google Drive
df.to_csv(final_csv, index=False)
print(f"Predictions saved to {final_csv}")

#-------------------------------------------------------------------------------------
# 5. CSV to Geopackage
#-----------------------------------------

#------------------------
# Aggregate results per GeoTIFF tile
#---------
```

```
# Input prediction CSV path
prediction_csv = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/results/b1-
sample_cls_seg.csv"


# Load prediction dataframe
df_pred = pd.read_csv(prediction_csv)


# Extract just the image file name for grouping
df_pred['image_name'] = df_pred['source_img'].str.split('/').str[-1]


# Group by image and aggregate classifications
# If sum of predicted_label >= n means at least n positive segment
image_classification = df_pred.groupby('image_name')['predicted_label'].agg(
    lambda x: 1 if x.sum() >= 1 else 0).reset_index(name='image_class')


# Summary counts
positive_count = image_classification['image_class'].sum()
negative_count = len(image_classification) - positive_count

print(f"Summary of GeoTIFF classification:")
print(f"Positive images: {positive_count}")
print(f"Negative images: {negative_count}")

# Save or display
print(image_classification.head())

# Save this result to CSV
output_csv = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/results/b1-sam-
ple_cls_summary_1.csv"
image_classification.to_csv(output_csv, index=False)
print(f"GeoTIFF classification summary saved to {output_csv}")


#------------------------
# Save as Geopackage
#---------

# PATHS
tiff_dir = Path("/content/drive/MyDrive/WeiBauDetect/Daten/yolov12/EG_random_reproducible/batch-
1_3000_sample")
classification_csv = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/re-
sults/final/b1-sample_cls_summary_1.csv"
output_gpkg = "/content/drive/MyDrive/WeiBauDetect/Daten/RandomForest/Segmentation/results/fi-
nal/b1-sample_cls_seg_1.gpkg"

# Load classification summary CSV
df_class = pd.read_csv(classification_csv)

polygons = []
labels = []
image_names = []
crs = None

for img_name in df_class['image_name']:
    img_path = tiff_dir / img_name
    if not img_path.exists():
        print(f"GeoTIFF not found: {img_path}, skipping")
        continue

    with rasterio.open(img_path) as src:
        bounds = src.bounds
        crs = src.crs  # Get CRS (assumed same for all images)

    # Create polygon from bounds: box(minx, miny, maxx, maxy)
    polygon = box(bounds.left, bounds.bottom, bounds.right, bounds.top)
```

```
    polygons.append(polygon)
    labels.append(df_class.loc[df_class['image_name'] == img_name, 'image_class'].values[0])
    image_names.append(img_name)

# Create GeoDataFrame
gdf = gpd.GeoDataFrame({
    'image_name': image_names,
    'classification': labels
}, geometry=polygons, crs=crs)

# Save as GeoPackage file
gdf.to_file(output_gpkg, driver='GPKG')
print(f"GeoTIFF classification polygons saved to {output_gpkg}")
```