University of Verona

Bachelor degree in Computer Science

Bachelor's Thesis

Study and application of deep learning techniques for the automatic detection of tree cover from satellite images

Academic Year 2024/2025

Supervisor

Prof. Davide Quaglia

Candidate

Enrico Antonini

matr. VR500151

To Roberto,

somehow, my traveling companion

Contents

1 ■ Introduction

This research aims to analyze and build a method to identify the presence of trees and green areas from satellite images of urban areas, using machine learning models.

In recent years, the advent of artificial intelligence has begun to change the way we work and study, allowing us to carry out research and also generate audio/video media from a simple text typed in a prompt. These types of artificial intelligence, however, although based on very revolutionary LLM models, are general-purpose and work on natural language.

The idea is to use machine learning models that, by leveraging a computer vision algorithm, can quickly identify the presence of objects such as tree crowns or shapes for purposes like urban planning or to support green access indicators, such as the one defined by the 3-30-300 rule.

2 Technical Details

2.1 What is a Neural Network

A neural network is a model that simulates the functioning of the human brain, so that it can learn to recognize images and texts, thanks to a learning process (training) and then subsequently give feedback and predictions about similar or same class elements, with a process called inference.

It is good to specify that not all models are based on neural networks, in fact there are non-neural algorithms that can simulate human cognitive reasoning, such as decision trees: they are the perfect example of models categorized as machine learning, as they learn from the training data provided in inputs, which are not composed of layers and nodes as in traditional neural networks[**Choi2020**].

Specifically, the multi-layer neural networks, or multi-layers, covered in this chapter are a particular subset of machine learning called deep learning.

2.1.1 Layers

A model is composed of several sequentially organized layers, called layers, made by nodes that can be considered as artificial neurons.

Simplifying as much as possible, layers can be input, output, or intermediate:

- Input layer: takes input data. If we use a YOLO model for example, it will be an image.

- hidden layers: receive input, process information through mathematical calculations and functions, and pass results to the next layer. They are the inner layers to the model.
- output layer: Returns the final prediction of the model. In our case, it tells us if it found objects within the image, possibly also specifying additional information such as classes and confidence level.

This layered structure allows the model to transform and combine data, learning to recognize patterns and objects. Each layer adds a new transformation, allowing the model to learn increasingly sophisticated functions[**ibm_layers**].

2.1.2 Weights

Each link between consecutive layer nodes is associated with a weight, which is a numerical value that indicates the strength of the connection between two nodes. During training, these weights are continuously updated so that, thanks to the passage of data through layers, the model can combine inputs more and more effectively to recognize patterns and make accurate predictions. Weights work on information and data, which are transmitted from one node to the next. Specifically a high weight amplifies the information, while a low weight attenuates it: it is precisely the optimization of these weights that allow the model to improve its accuracy and predictive ability[**googlemlglossary_weights**].

2.2 Dataset

A dataset is a structured collection of data, in the case of this research it is mainly formed by images with annotations and is organized to be analyzed and processed by machine learning algorithms. Datasets can be composed of:

- Training set: images used for model training.
- Validation set: images used during training, to evaluate the performance of the model on unseen data and to perform parameter tuning.
- Test set: a set of images used at the end of training, to evaluate the final performance of the model, on completely new data that was not used for training.

The validation set is optional, but very useful for improving the quality of the training of the model.

2.2.1 Importance of dataset

The quality and consistency of the data in the dataset is as important as the algorithm itself: from a dataset with clean images and well-made annotations it is possible to do training (or transfer learning) and train an accurate model.
The dataset must also contain images in the format provided by the model specifications (e.g. YOLO in 640x640)
The subdivision into training, validation and test sets allows to develop a robust model and test its performance on data "never seen before"

2.2.2 Manual Annotation

To build a dataset suitable for the object recognition theme, which is the purpose of this research, you need to have images with resolution, number of channels (e.g. RGB) and format, which are compatible with the model to be trained.
Depending on the neural network to be used then, the images must be accompanied by a file describing the content or object to be identified: for example for YOLO models each image is associated with a file, which contains coordinates and class of each object present within the image.
At this point the neural network that underlies the model, during the training procedure, will learn to distinguish the objects that we have annotated in the dataset and precisely thanks to this procedure, will then be able to evaluate in a more or less precise way, depending on the type of network and the quality of the training dataset, whether or not any image provided to it in input contains objects similar to those annotated, also providing values that indicate the estimate of how precise that detection is.
Platforms like Roboflow offer an editor where you can annotate images and simultaneously generate files with annotations.

2.2.3 Searching for a dataset with annotated images

A second solution is to use pre-annotated datasets, which can be downloaded from Roboflow or GitHub repositories.
In this case it is crucial to make sure that the dataset has been created specifically for the network you want to train and that the annotations are correct. In general, the datasets made available online, if from computer vision projects and documented in research articles, are quite reliable.

2.3 Training

For a machine learning model to learn to recognize significant patterns or features in an image, it is necessary to perform an operation called training, which is in fact a real training of the model.
Without proper training the neural network would not have a strategy to interpret the data and perform the desired task.

The training is carried out starting from a dataset, or a collection of images with their "annotations": in each image of a dataset the position and class of the object is highlighted, which is indispensable to train the network to subsequently recognize these objects in the images that we will have to classify.
There are two types of training that can be performed on a network, full training (from scratch) and transfer learning[**isong2025buildingefficientlightweightcnn**].

2.3.1 Training from scratch

In this type of training, the model is fully trained on annotated data without starting from pre-trained weights. Weights are numerical values that are assigned to connections between nodes in the neural network: whenever an input arrives at a node, it is multiplied by the weight associated with that specific connection, the sum of all these inputs will then determine its output values.
During the neural network training phase, weights are constantly updated using learning algorithms, so as to minimize the difference between model predictions and real values.

Starting from scratch, these weights are not defined but random numbers are assigned that, only during the training process, will be modified to optimize performance.
This type of training is very valid when we have available a considerable number of annotated data and the use of the model is very specific.

2.3.2 Transfer learning

When the dataset is limited, or if the goal is to save time and resources, it is possible to start from a pre-trained model.
The idea is to "transfer" to a model provided with weights and with already significant precision parameters, the part of knowledge necessary to recognize the annotated data in our dataset, we then start from the pre-trained model and adapt it (fine-tuning) to the target, often "freezing" the first layers and optimizing only the final ones.

2.4 Training parameters

Below are listed some parameters, which are normally specified during model training: they are useful for honing the learning process and are also called "Hyperparameters".
These parameters are rather generic, because they are common to many training procedures even between different algorithms[**googlemlglossary**].

2.4.1 Epoch

An epoch represents a complete passage of the dataset through the algorithm being learned, it is a fundamental concept in the training of neural networks, which allows the model to learn by constantly continuing to review the same images, or the same elements of the dataset[**ultralytics_epoch**].
The epoch number is usually specified in each training procedure and determines how many times the model will learn from the dataset train set, affecting the performance and quality of the model.
Choosing the correct epoch number is very important because if it is chosen incorrectly it can create two problems:

- Underfitting: the model was not trained for a sufficient number of epochs. Performances are not adequate on either the training set or the test set

- Overfitting: the model has been trained for too many epochs. In this case he stores the training data too thoroughly and loses his ability to analyze the new data. It has excellent precision on the training set but poor on the validation dataset.

A common technique to avoid overfitting is that of early stopping, in which training is stopped when performance on the validation set stops improving. Sometimes it is possible to specify a parameter called patience, which specifies after how many "non-improvement" epochs the training must stop.

2.4.2 Batch size

Batch size is another important parameter that defines the number of samples processed before the model's internal parameters are updated. Samples are in this case the images, which are used for model training.
By dividing the training set into batches, i.e. reduced subsets, avoiding processing all data simultaneously, which could be computationally problematic in terms of processing speed and occupied memory.
Choosing a batch size too low though, implies a reduced training speed, as the weights of the model are updated much more frequently.

2.4.3 Loss function

It is a fundamental mathematical function in machine learning, which measures how far the predictions generated by the model deviate from real values.

$$\text{Loss} f(prediction, realValue)$$

The function is called both during training for each batch of images, and during validation at the end of each epoch. The value then returned by the loss function, during training, can be used by optimizers that allow you to update parameters, such as weights, to each epoch as a sort of "automatic pilot".

2.5 Hardware needed for training

Machine learning models need special hardware resources to be able to be trained because, using a normal desktop computer that uses the CPU alone to "instruct" the model, is too limiting not so much for the frequency but for the insufficient number of cores it has on board, it is instead necessary to use hardware that allows parallel computing: for this task the GPUs come to our aid.
The latter, in fact, especially those produced by Nvidia, are compatible with the parallel computing platform CUDA (Compute Unified Device Architecture) that takes advantage of the hundreds or thousands of cores on board the chips, thus managing to perform massive mathematical calculations and manage huge amounts of data[**whatiscuda**].
Importantly, heavier machine learning models require a lot of RAM, and it is therefore necessary to equip yourself with GPUs with sufficient video memory (VRAM) on board, to ensure that the model can remain fully resident in memory during training or inference procedures.

2.5.1 Bare metal

To use CUDA on a local machine you need to buy the physical hardware, which generally in the consumer sphere is an expansion board on PCIe buses that allows the connection of one or more monitors and has the task of accelerating calculations that would be too heavy for the CPU, such as calculating polygons and shaders in games or training/inference on machine learning models.
In the industrial sector, however, there is room for solutions not designed for the recreational sector, which are instead marketed as parallel computing accelerators.
There are also reduced versions on embedded/single board computers (e.g. Jetson Orin) that allow the use of CUDA on the edge, for example to evaluate images from a camera.
All of these solutions involve the purchase of dedicated hardware, which needs to be configured correctly in order to work and, especially the most performing GPU models with high VRAM have a major cost.
Python is usually used for training and inference, as it has libraries suitable for working with models (such as Pytorch or TensorFlow) and it is necessary that the dedicated code is developed.

2.5.2 Cloud Resources

A real alternative to on-premise processing is to use cloud services that result in hardware cost savings, as it is all left to the managed platform, but still imply the need to work from Python sources for training or inference with models.
There is usually a monthly budgetable cost with a calculator, which varies depending on the service and hardware made available by the cloud platform.
The most famous providers are Google Cloud Platform, AWS EC2 and Azure VMs, which offer Linux or Windows virtual machines with GPUs.
Alternatively, at a lower cost, services such as Google Colab are available that only support scripting languages (e.g. Python, R) offering a monthly fee access to a number of CPU+GPU processing hours. They are a good compromise to work with machine learning models.

2.5.3 All-in-one solutions

One solution that is taking hold in recent years is to rely on all-in-one services that, against a monthly fee, provide for the management of the dataset and the creation of the model through training. This type of service also offers the ability to access online models, via REST services, with endpoints that allow you to inference without having to worry about developing scripts to run on your machine or in the cloud.
Among the most popular services are definitely Roboflow and Ultralytics.

2.6 Comparing Models

At the time of writing this document (ca. end 2025) several solutions are available, which promise to be effective in identifying objects within an image, in a more or less elaborate way.
Unlike neural networks that became popular in the first half of the 1920s (e.g. ChatGPT, Gemini, Llama or Grok) to be very effective with natural language (NLP) given their nature of LLM, acronym of Large Language Model, in this specific case the research will deal with neural networks for computer vision, such as recursive and non-recursive convolutional networks.

2.6.1 AdaBoost

It is a machine learning algorithm that combines multiple models called weak learners to create a more accurate evolved model[**bejabattais2023overviewadaboostreconciling**]. Weak learners are slightly better performing models or algorithms than a random prediction, for example in a binary classification they reach just over 50%of precision. By doing sequential training on these weak learners and combining them, the final model will be nothing more than the weighted sum of all the smaller models, resulting in more performing and robust.
In the case of AdaBoost the process is adaptive because it corrects errors made in the first round of training, thus making it more effective.
It is important, as previously mentioned, to emphasize that AdaBoost is not a neural network but, by its nature by combining smaller models, it is "simply" a decision tree.
DetecTree uses AdaBoost to classify, pixel by pixel, what is part of a tree and what is not.

2.6.2 CNN

Some object detection algorithms are based on a convolutional neural network (CNN)[**oshea2015introduction**
which is used to allow detection of objects in the image. It is a specific type of deep learning algorithm, designed primarily to analyze visual data, such as images and videos. These algorithms are categorized as deep learning, precisely because of their characteristic of having multiple levels.
In practice it mimics the functionality of the human cerebral cortex, automatically extracting certain characteristics or objects, such as trees, from the image: this is possible through filters called convolutionals, which go in search of specific patterns.
The name of the filters comes from the type of operation that is carried out on the input images, that is the convolution: it is a mathematical operation that in simple terms consists of sliding a filter (also called kernel) over the input image. The filter examines the image and calculates a weighed combination of pixel values, producing an output value. Repeating the process several times creates a new image that is able to highlight specific features such as edges, angles or textures: in essence it then allows the network to recognize visual patterns efficiently while also maintaining the spatial position of the detected objects. An example of CNN is the YOLO network.

2.6.3 R-CNN

In some cases networks can also be R-CNN, i.e. Region-based CNN which, like normal convolutional networks, are also deep learning algorithms.
Again they are models designed for object detection: first multiple proposals of regions (areas potentially containing objects) are generated through specific algorithms, then a CNN is applied to each region to extract characteristics, and finally objects are classified and located in those regions[**r-cnn**]. In essence, no filters are applied to the whole image, but convolution operations are carried out locally in distinct regions.
Among the R-CNNs we can find DetecTree2.

2.6.4 ViT

The commercial LLM models mentioned at the beginning of this section, which in recent years have undergone an incredible evolution, have introduced an architecture called a transformer.
As opposed to (R)CNNs seen earlier, transformers rely on self-attention mechanisms where some of the input data, such as a word or image, is compared and weighed against all other parts to understand which ones are most relevant in the current context. In this way the model can assess the importance of the different parts of a sequence, regardless of their position[**vaswani2023attentionneed**].
A Transformer consists of:

- Encoder: Converts text to input in an intermediate representation. The encoder is a neural network.
- Decoder converts the intermediate representation into a text output. The decoder is also a neural network.

Recently, the Transformer architecture has also been applied to networks dedicated to vision, thus creating Vision Transformers (ViTs): instead of treating the image like a pixel grid, as is the case with (R)CNNs, the ViT splits the image into small pieces of fixed size, which are then transformed into vectors (embeddings), are enriched with spatial position information and then switched to a transformer encoder, which uses both global and local mechanisms.
This process improves performance in tasks such as classification, segmentation, and object recognition, especially with large training datasets. Segformer is a neural network based on this deep learning algorithm.

2.7 Performance measurement

Models, once trained, can be evaluated based on certain metric specifications, which are also common across different neural networks.
Below are listed several general metrics, the specific metrics and dedicated to precise models, are instead reported in the following chapters.

2.7.1 Precision

Indicates the percentage of objects identified by the model that are actually correct. In other words, it indicates, on all the times the model has found an object in an image, how many times it has actually "got right". High accuracy means few false positives.

2.7.2 Recall

It measures the percentage of objects actually present in the image that were found by the model, basically it returns us a value relative to the number of objects recognized compared to how many were currently annotated as good in the dataset. A tall recall is equivalent to a few false negatives.

2.7.3 Intersection over Union (IoU)

Fundamental computer vision metric in object recognition, which measures how much two rectangles (boxes) overlap with each other relative to their extension. It is used in practice to compare how much the predicted bounding box, coincides with the real bounding box (called ground truth).

$$IoU\frac{areaIntersection}{areaUnion}, \in [0,1]$$

for example an IoU of 0.6 indicates that two boxes have the 60%of shared area[**mAPV7**].
The IoU threshold is an input value of the model, which establishes how much an object must be "centered" to be considered correct the prediction: the higher the value, the stricter the valuation.

2.7.4 Mean Average Precision (mAP)

Parameter used to measure the performance of computer vision models, such as YOLO or DetecTree2. The mAP represents the average accuracy on different IoU thresholds, of all classes present in the dataset: it is calculated as the arithmetic mean of the accuracy values of all classes, providing an overall measure of the model's ability to classify objects correctly. It is also crucial to compare different models that are involved in carrying out the same task, or even different versions of the same model.

$$\text{mAP} = \frac{1}{N}\sum_{i=1}^{N} AP_i, \in [0,1]$$

with N number of classes and APi that corresponds to the average accuracy of class i[**mAPV7**].

2.7.5 Mean Average Precision 0.50 (mAP50)

Metric that calculates the Average Precision across all classes, using a fixed IoU threshold of 0.50. A prediction is considered correct (true positive) only if the overlap with the ground truth is at least 50%.
This metric provides a model performance assessment with a tolerance threshold, and is useful for applications where an approximate object location is sufficient.

2.7.6 Mean Average Precision 0.50-0.95 (mAP50-95)

Metric that calculates the Average Precision by considering multiple thresholds of IoU, 0.50 to 0.95 in increments of 0.05 (total 10 thresholds). For each threshold the mAP is calculated, then the average of all these values is made.
This metric is much more rigorous than mAP50, because it evaluates the model's accuracy on different levels of difficulty, requiring both approximate and very precise localizations. A high mAP50-95 indicates that the model is accurate in all contexts, from approximate detection to millimeter accuracy.

2.7.7 F1 score

F1 is the harmonic mean of precision and recall

$$F1 = 2 * \frac{precision + recall}{precision * recall}, \quad F1 \in [0, 1]$$

this value balances precision and recall, penalizing strongly the case where one of the two is very low, in fact the closer you get to 1 the less you have false positives and false negatives. This value is very important because it tells us how precise and complete a model is in its predictions.

3 ■ YOLO

YOLO is a popular convolutional neural network (CNN) that allows for real-time detection of objects within images.
Unlike other computer vision algorithms, YOLO performs its analysis in a single iteration, hence the acronym YOLO, You Only Look Once.

3.1 Principle of operation

The working principle of the YOLO algorithm can be summarized in the following steps: [**yoloExplained**]

- The input image is divided into an n x n grid.
- Thanks to a probability map defined a priori, the positions of multiple bounding boxes are predicted, which highlight the detected object.
- The class of the detected object is returned along with the corresponding bounding box.

the "class" of the object corresponds to the type of annotation present in the dataset used for model training: in our case, since the model must recognize the presence of trees in a given satellite image, we will have only one class relating to trees.

3.2 Type of model used

YOLO models have evolved over time and are still being developed to increase their accuracy and performance; each new version seeks to achieve higher accuracy and lower inference times than previous models.
YOLO's deep learning algorithm is developed using the PyTorch framework.

3.2.1 PyTorch

It is an open-source deep learning framework in Python, created to facilitate the creation, training and deployment of neural network models [**pytorch**]. Among its main features are:

- Support for tensors, i.e. multidimensional arrays, that can be processed on CPU or GPU with parallel computation.
- Includes modules for creating complex neural networks with gradient calculation for optimization.
- It offers libraries such as torchvision to support computer vision and torchtext for natural language processing.
- Cross-platform compatibility and broad support thanks to its large community.

3.2.2 Model version

The version of YOLO used in this project is version 11, introduced in 2024, which guarantees better accuracy in predictions and higher speed. Additionally, a version optimized for edge devices, such as smartphones or embedded systems, is available.
Unlike some older versions of YOLO, here we have the ability to use pre-trained models in training procedures, thus making training possible even with reduced datasets.
YOLO models are available in various "sizes": in fact, in addition to the extralarge versions, which are more precise and complete, there are also medium and small versions that are suitable for use on devices with more limited resources.
Observing some comparison statistics between various YOLO systems, with the same dataset, there is a significant improvement in average accuracy values (mAP50) compared to previous

versions even for medium-sized models (e.g. YOLOv11m vs YOLOv10m) and a reduction in the time required to perform model inference. [**yoloBenchmarks**]

3.2.3 YOLO Ultralytics

Ultralytics is a commercial platform that has developed libraries, usable under the AGPL-3.0 license, specifically designed to facilitate training and inference with YOLO models [**yolov11Ultra**]. Using these libraries also offers the significant advantage of being able to convert YOLO PyTorch models to other formats, such as Tensorflow JS, making them usable on web frontend applications.

3.3 Preparing the dataset

As already introduced in the previous chapter, having a well-formed dataset is crucial for proceeding with YOLO model training: without an adequate number of images and annotations, training would indeed be ineffective.
In this specific case, which requires the detection of trees in the urban fabric, the choice of data source is not trivial: most models that recognize this type of plant are trained on datasets that include satellite images of countryside or wooded areas.
For the training of this model, however, a dataset distributed online by the University of Lleida in Spain was used [**lleidaDataset**], consisting of a set of training images and an evaluation set. It contains precise annotations of trees in an urban context, with images already in YOLO-compatible format (640x640).

3.4 Training script

The following are some fundamental functions for training a YOLO model, leveraging the Ultralytics libraries: they are not to be considered exhaustive, but illustrate the basic operations for training and collecting statistical data related to the quality and performance of the model.

3.4.1 Init

```
import torch
```

```python
from ultralytics import YOLO
from pathlib import Path
import os

baseFolderPath = "../myDataset"
datasetYamlPath = f"{baseFolderPath}/dataset.yaml"
```

3.4.2 Parameters

The necessary parameters, specific to the training of the Lleida YOLO11x model, to be provided to the Ultralytics library training function, are defined below:

Property	**Type**	**Range**	**Description**
imgsz	int		Input image size, must be square, so only one value is specified for all sides
device	string	`cpu`, `cuda`, `mps`	Declares the type of device on which training is launched
workers	int		Number of processes or threads that can run simultaneously
save_period	int		Number of epochs required before saving model weights
project	string		Main path where trained data is saved
name	string		Project name, used to create a subdirectory containing outputs
pretrained	boolean	`true`, `false`	If true, starts from a pre-trained model (transfer-learning)
lr0	float	(1e-5, 1e-1)	Initial learning rate. Low values make the training process more stable but slower
lrf	float	(0.01, 1.0)	Final learning rate, specified as a fraction of lr0. Manages how much the learning rate should slow down during training
cos_lr	boolean	`true`, `false`	If true, the initial rate scheduler follows a cosine curve instead of growing linearly. Improves convergence and leads to higher precision
cache	string	`none`, `ram`, `disk`	Specifies where data is temporarily saved to ensure faster learning speed
cls	float	(0.2, 4.0)	Loss function multiplier, useful for controlling training progress. Can be increased if the model struggles to recognize objects.
single_cls	boolean	`true`, `false`	If true, the model considers all annotations as a single class

Table 3.1: Training parameters for YOLO11

The Python object containing the parameters is initialized as follows:

```python
trainArgs = {
    'data': datasetYamlPath,
    'epochs': 300,
    'imgsz': 640,
    'batch': 16,
    'device': device,
    'workers': 8,
    'patience': 100,
    'save_period': 5,
    'project': f"{baseFolderPath}/runs/detect",
    'name': 'treeDetectionYolo11Scratch',
    'pretrained': True,
    'lr0': 0.01,
    'lrf': 0.1,
    'cos_lr': True,
    'cache': "disk",
    'single_cls': True
}
```

3.4.3 Training function

```
1 results = model.train(**trainArgs)
2 metrics = model.val()
```

With these two lines, the actual training procedure is launched.

On line 1, the function *model.train(**trainArgs)* is called which, thanks to the *train_args* set previously, begins the training process: the training results are then saved in the variable *results*. In the second line, the script performs model validation using the validation set, which is fundamental for measuring the performance acquired during training. The function output returns evaluation metrics such as mAP (mean Average Precision), precision, recall, etc.

3.4.4 Output of the training procedure

```
Epoch   GPU_mem  box_loss  cls_loss  dfl_loss Instances     Size
1/300     15.3G     1.989     1.919     1.688        63      640: 100%|**********| 54/54
    [00:41<00:00, 1.31it/s]
Class     Images  Instances     Box(P         R      mAP50  mAP50-95): 100%|**********| 7/7
    [00:04<00:00, 1.40it/s]
all        218       3262   0.000154   0.00276   7.76e-05   2.15e-05
```

During the training procedure, some useful statistics are displayed in the console to monitor training progress.

Above are a couple of example lines with output values, from both the training and validation functions.

Training statistics:

- *Epoch*: progressive epoch count.
- *GPU_mem*: total VRAM used (in this case training was done with CUDA).
- *box_loss*: measures the distance between the position and size of the bounding boxes predicted by the model compared to the actual (ground truth) bounding boxes.
- *cls_loss*: measures the loss relative to classification, i.e. how well the model is identifying objects.
- *dfl_loss*: improvement of bounding box regression.
- *Instances*: number of objects processed in this batch/epoch.
- *Size*: size of input images.
- *Progress bar* %|...|: n/n [00:00:00 z it/s] the number (n) of batches processed in a certain period of time in seconds at a certain speed defined as z measured in batches per second.

Validation statistics:

- *Class*: indicates for which classes validation is performed.
- *Images*: number of images in the validation set.
- *Instances*: number of trees in the validation set images.
- *Box(P)*: precision in bounding box detection.
- *R*: recall, see above
- *mAP50*: see above
- *mAP50-95*: see above

3.5 Training Results

3.5.1 Graphs

The Ultralytics platform, after completing the training phase, produces some graphs that help better visualize the performance of the trained model. Perhaps the most significant graph is the one related to the F1 curve:

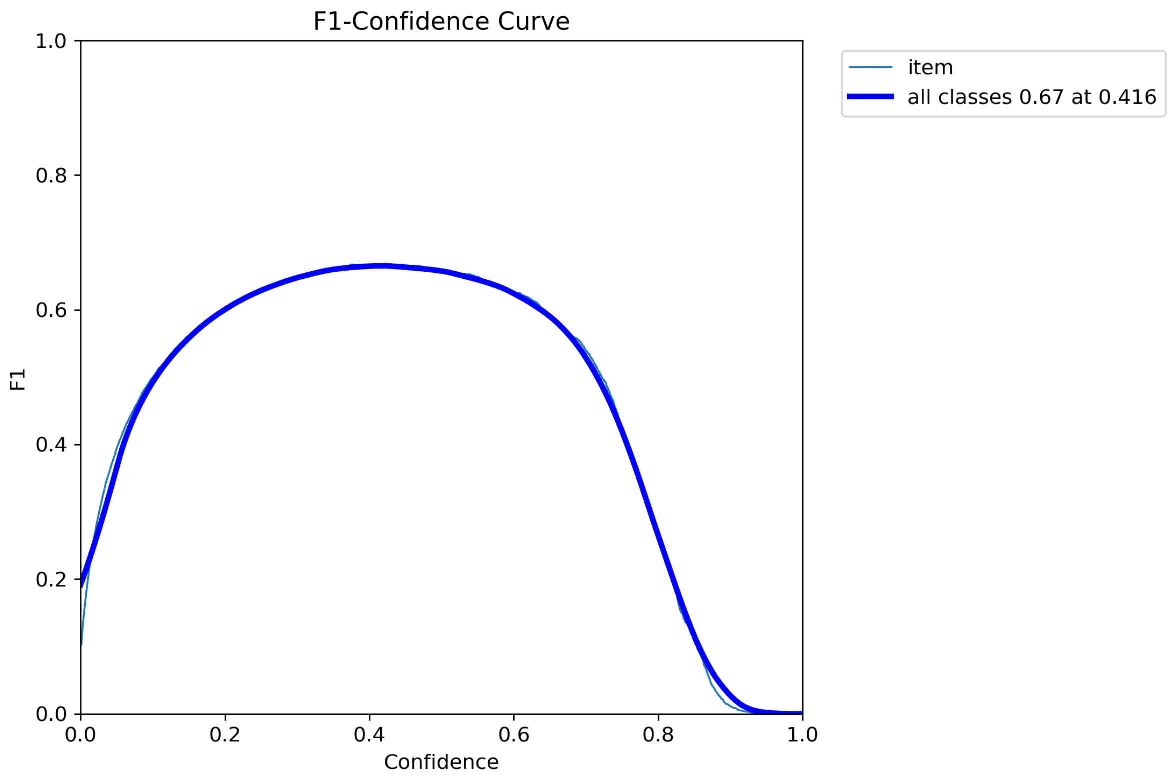

Figure 3.1: F1-Confidence Curve. Confidence value on the x-axis, F1 value on the y-axis.

The maximum of the curve represents the optimal point where the confidence threshold provides the best compromise between precision and recall. The maximum F1 score of 0.67 is obtained with confidence 0.416.

This graph is very important for choosing the confidence value to provide during the inference phase, where false positive (precision) and false negative (recall) phenomena are minimized as much as possible.

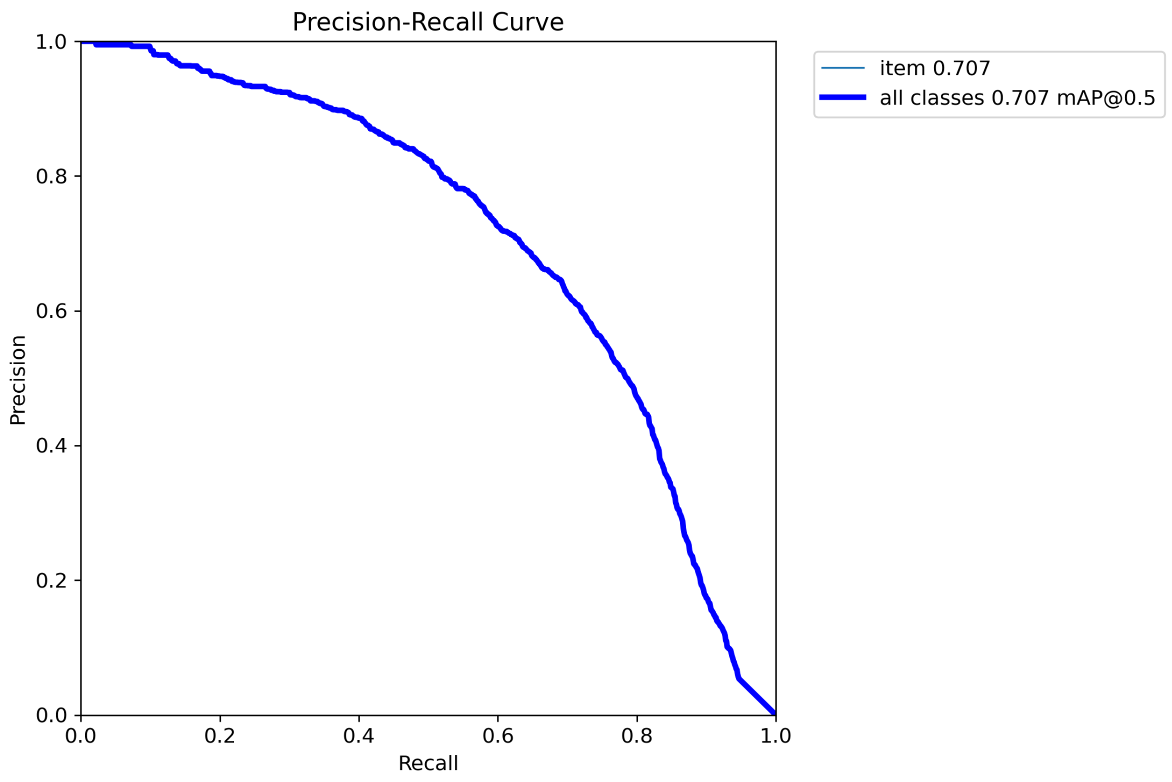

Figure 3.2: Precision-Recall Curve. Recall value on the x-axis, precision value on the y-axis.

This graph shows the relationship between precision and recall, with the mAP50 value of 0.707, indicating the average precision at recall intervals for all classes, in this case only for trees, at the IoU threshold of 0.5.

Essentially, it highlights the balance between the model's ability to correctly identify objects (precision) and the ability to find them all (recall) for various confidence threshold levels.

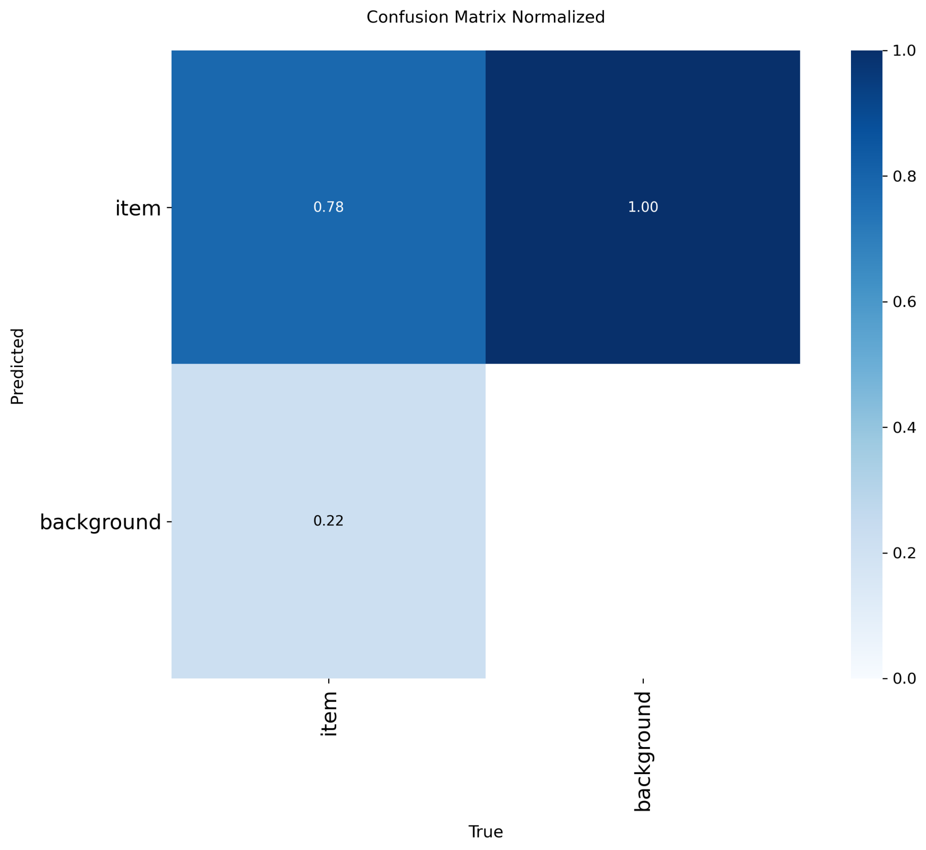

Figure 3.3: Confusion matrix. Shows the classes that the model is able to detect.

The confusion matrix shows how the model classifies the classes on which it was trained. In this case there is only one class "item", i.e. the tree, while "background" is everything that is not considered a tree. The graph can be summarized as follows:

- Predicted "item", True "item" (0.78): the model correctly identified 78% of trees.
- Predicted "background", True "item" (0.22): 22% of trees were classified as "background" (false negative).
- Predicted "item", True "background" (1.00): All "background" was classified as such (true negative).
- Predicted "background", True "background" (0.00): No "background" was misclassified as tree (false positive absent).

3.6 Inference

After training the model with the Lleida dataset, it is possible to proceed with inference on other images to verify the presence of trees.

Below is the Python function for YOLO inference with Ultralytics, using the PyTorch library, along with OpenCV which is used in this case to work with input and output images.

It is important to ensure that the input images are in the same format as those prepared for training, in this case 640x640; they must therefore respect the resolution and RGB channel composition.

3.6.1 Predictive analysis function

```python
def analyze_tree_coverage(imagePath, modelPath, conf=0.05):
    model = YOLO(modelPath)
    img = cv2.imread(imagePath)
    results = model(img, conf=conf)

    for result in results:
        if result.boxes is not None:
            boxes = result.boxes.xyxy.cpu().numpy()
            confidences = result.boxes.conf.cpu().numpy()

            for box, confidence in zip(boxes, confidences):
                if confidence >= conf:
                    x1, y1, x2, y2 = box.astype(int)
                    cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)

    outputPath = Path(imagePath).stem + "_result.jpg"
    cv2.imwrite(outputPath, img)
```

The function dedicated to searching for trees within the image requires three input parameters:

- *image_path*: path of the image to be analyzed
- *model_path*: path of the trained model to use for inference
- *conf*: minimum confidence level within which the predictor considers the object as valid

The first four lines concern the initialization of the YOLO model, reading the image using OpenCV, and the direct call to the prediction function on line 4, where the detection results are saved in *results*.

OpenCV is a very versatile library that offers functions and utilities for computer vision: here it is used to read the image from file and to overlay the bounding boxes that delimit the detected trees [**opencv_library**].

Subsequently, the prediction results are analyzed one by one: for each result, it is verified whether a bounding box is associated, i.e. the rectangle that outlines the object found within the image. If present, it extracts the coordinates and detection confidence, saving them in *boxes* and *confidences* respectively.

Each extracted box is then drawn, via OpenCV, on the output image, only if the detected confidence exceeds the threshold specified as an input parameter.

3.7 Importance of the urban dataset

The dataset used for training this specific YOLO model consists of a series of satellite images containing trees located in urban areas. These datasets are rare because they are very specific. Most of those available online or on major platforms are in fact images collected in rural contexts. Furthermore, thanks to the large amount of images present in rural datasets, the values of mAP50, mAP50-95 and F1 are generally much higher.

Considering therefore the difficulty of accessing this type of data, why use and even train a model on an urban dataset when many pre-trained models are available? The answer is less trivial than one might think; in fact, not only the annotated object is relevant, its "context" is also important, as in this case the surrounding urban environment.

The test performed was very simple: run inference on a specific image both with the model trained with the Lleida dataset and with a model trained on a dataset available on Roboflow, which shows images and annotations in an extra-urban context [**tree-project-cyyr8_dataset**].

The results of the comparison, performed on a satellite image of Lungadige San Giorgio in Verona, show that the model trained with the Lleida dataset correctly detects most of the trees present along the river, while the model trained on forests erroneously identifies only a few canopies, confusing urban vegetation with the background.

The model trained with a dataset containing urban images therefore performs considerably

better than the one trained with images of forests and plantations.
A similar difference was also found by researchers at the University of Beijing, where in an article they discuss the use of YOLov4-Lite for tree detection in urban and non-urban plantations (such as orchards), reporting difficulties due to the high rate of false detections in forests, caused by the similarity between the colors of the background and those of the canopy [**yolo4plantation**].

4 ■

DetecTree

DetecTree is a machine learning algorithm that is not limited to the detection of individual trees, but also allows for the identification of entire wooded areas or green zones within satellite images. As introduced in the technical chapter earlier, it cannot be categorized as a neural network, since the algorithm based on the AdaBoost detector is not composed of nodes and layers but "simply" of a decision tree.

4.1 Principle of operation

The operating principle of the DetecTree algorithm can be summarized in the following points [**bosch2020detectree**]:

- Tile subdivision: the aerial image is divided into smaller tiles to generate a mosaic of "tiles" of a specific size.
- Choice of training tiles: tiles to be used for training are selected using GIST descriptors, i.e. numerical arrays that synthesize visual and semantic characteristics of an image portion.
- Ground truth masks: for each tile chosen for training, "tree/non-tree" binary masks must be provided using image editing tools.
- Training: for each pixel, an array of 27 features is extracted and an AdaBoost classifier is trained, which is effectively a binary classifier that associates the feature vector with the

"tree/non-tree" classes.

- Testing: proceeding with inference on test tiles, the classification is then refined with an algorithm, improving detection on adjacent pixels classified as tree.

The set of 27 features is a representation given to each individual pixel by the AdaBoost classifier during training, to allow the model to distinguish pixels belonging or not belonging to a tree. This type of classification is performed based on both color information and image context. The features are distinguished as follows:

- 6 color features: contain pixel color data, such as RGB channels and possible derived combinations.
- 18 texture features: statistical measures (e.g. mean, variance, contrast, etc.) computed on local regions around the pixel, often derived from models.
- 3 entropy features: measure the unpredictability of intensity values in a specific region around the pixel.

4.2 Type of model used

In this case, the GitHub project repository makes the pre-trained model available, so that it can be used directly without having to resort to the training procedure but, above all, without having to resort to manual annotation of images for the training dataset.

4.3 Parameters

In the case of DetecTree, in addition to the image that highlights tree cover, the percentage value of the cover present within the analyzed image is also available.

4.4 Inference

```
import skops.io as sio

untrusted_types = sio.get_untrusted_types(file="detectree_model.skops")
model = sio.load("detectree_model.skops", trusted=untrusted_types)

```

```
    clf = dtr.Classifier(clf=model)
    pred = clf.predict_img(image_path)
```

These five lines of code are responsible for loading the model and performing inference with DetecTree.
In order to classify the image, you must first import the Skops library [**skops**], which is a Python library used to import models based on Scikit-learn, a well-known open-source platform for machine learning in Python.
The model could potentially be imported directly from Scikit, but this is not recommended, as the library uses a standard Python module (Pickle) to serialize and deserialize models, which can execute arbitrary code during deserialization, exposing the host on which it is running to potential attacks.
In the following lines, the model types that define which structure or class the object has (such as a model, vector, or related data) are loaded. In this case, it is chosen to load all types, even those that are not automatically considered safe to load: even when loading types considered "unverified", the protection that Skops offers against Pickle's deserialization attacks still remains.
Once the model is loaded, classification proceeds by performing the prediction with *predict_img*, which returns the label map (e.g. 2D array of classes) for the input image.

4.5 False positives

The use of AdaBoost nevertheless introduces a significant issue as, working by similarity, it is not only imprecise at distinguishing between trees and meadows, but is heavily penalized when green areas are close to other zones with similar characteristics, such as watercourses.
In this case, in fact, DetecTree struggles to delimit the boundaries of green areas, erroneously classifying areas that it should not consider as wooded.
A significant example of this problem can be found in the Ponte Aleardi area in Verona, near the gardens of the monumental cemetery: the algorithm mistakenly classifies a portion of the waters of the Adige river as a tree-covered area, due to the chromatic proximity to the foliage of the trees on Lungadige Porta Vittoria.

5 Detectree2

This deep learning algorithm is not a direct evolution of DetecTree, as the implementation is not derived from AdaBoost but is instead a true region-based convolutional neural network (R-CNN) [**detectree2**].
At the base of Detectree2 is Detectron2, an advanced platform developed by FAIR (Facebook AI Research), built on the PyTorch framework [**wu2019detectron2**].

5.1 Principle of operation

Detectree2 aims to detect, within a satellite image, the trees it contains by searching for the set of branches and leaves located in the upper part of the trunk, i.e. the crown.
To achieve this, the Detectron2 R-CNN algorithm has been adapted, which has pre-trained models on which transfer learning can be performed to "add" the knowledge necessary to employ them in tree detection.

5.1.1 Detectron2

At the heart of Detectree2 is an R-CNN-FPN, i.e. a region-based convolutional network with Feature Pyramid Network, which uses a pyramid of multi-resolution feature maps. This allows for improved object detection by extracting image representations at different levels of detail, thus facilitating the recognition of objects of different sizes.
A feature map is a representation produced by a convolutional layer of a CNN: during pro-

cessing, the network applies filters to the input data to detect specific features, such as edges, textures, or particular patterns.
Schematically, the network architecture can be summarized in three blocks [**detectron2_modeling**]:

- Backbone Network: extracts feature maps from the input image at different resolutions via FPN.
- Region Proposal Network (RPN): analyzes the feature maps to detect regions containing objects, returning bounding boxes and confidence values.
- Box Head: crops and resizes feature maps corresponding to the detected regions, processing them to classify the object and further refine the bounding box positions.

5.1.2 Modifications introduced by Detectree2

Detectree2 adds to the Detectron2 library the functionality of managing georeferenced inputs and outputs and of delineating individual tree crowns through the generation of masks that precisely circumscribe the object in the image.
This increases precision and performance in crown segmentation, thus obtaining an improvement over the detection offered by Detectron2.

5.2 Type of model used

Also in this case, as with DecTree, the GitHub project repository makes multiple pre-trained models available [**detectree2_models**].
The models have been trained with different datasets; as specified in the repository readme, it is therefore necessary to select the model most suited to the type of trees to be detected. In the specific case of this research, the model used is *urban_trees_Cambridge20230630*, which was specifically trained to detect tree crowns in urban areas.

5.3 Parameters

The parameters for Detectree2 are those common to convolutional neural networks, such as those already seen for YOLO. In fact, the Cambridge model reports among its statistics some parameters that were introduced in the technical details chapter:

- Learning rate: 0.01709
- Workers: 6
- Batch size: 623
- AP50: 62.0

Given the considerations seen in previous chapters, an AP50 of 62.0 can be considered a good result: it indicates that the model is able to localize the objects for which it was trained, with a good balance between precision and recall.

5.4 Inference

After downloading the appropriate model, in this case Cambridge, it is possible to proceed with inference.

According to the requirements listed on the GitHub repository, for detection to be accurate, it is necessary that the images have the same format as those used in the training phase, in which square tiles covering approximately 200m were used. After a quick estimate, considering the coordinate/pixel ratio of the images provided by the provider and also taking into account the zoom level set, the resolution corresponds to approximately 364x364. To this parameter is also added, as a fundamental requirement, the order of the channels, which must be BGR.

Below are some code sections dedicated to inference with this specific model.

5.4.1 Initial configuration

```python
from detectron2.engine import DefaultPredictor
from detectree2.models.train import setup_cfg

def init(modelPath: str, outputFolder: str):
    cfg = setup_cfg(update_model=modelPath)
    cfg.OUTPUT_DIR = outputFolder
    cfg.MODEL.DEVICE = "cpu"

    predictor = DefaultPredictor(cfg)
```

In the first part of the code, it is essential to correctly initialize Detectron2 and Detectree2. Since the former leverages the structure and network of the latter, except for some features already listed previously, it is necessary to import the *DefaultPredictor* from Detectron2 and the model configuration from Detectree2, as can be seen from the first two lines of the listing.

In the following lines, the model configuration is initialized, specifying as parameters the model path defined in *update_model*, along with the output directory path and the type of device to be used for inference.

The last line instead concerns the initialization of the *DefaultPredictor* starting from the configuration declared for Detectree2.

5.4.2 Prediction

```python
import cv2
from shapely.geometry import Polygon

def predict(
        predictor: DefaultPredictor,
        imagePath: str,
        conf: float = 0.5) -> Dict[str, Any]:
    image = cv2.imread(imagePath)
    outputs = predictor(image)

    instances = outputs["instances"].to("cpu")
    scores = instances.scores.numpy()
    validIndices = scores >= conf
    filteredScore = scores[validIndices].tolist()

    polygons = []
    if hasattr(instances, 'pred_masks'):
        masks = instances.pred_masks.numpy()[validIndices]

        for mask in masks:
            contours, _ = cv2.findContours(mask.astype(np.uint8), cv2.RETR_EXTERNAL,
                 cv2.CHAIN_APPROX_SIMPLE)
            if contours:
                largestContour = max(contours, key=cv2.contourArea)
                polygonCoords = largestContour.reshape(-1, 2).tolist()

                if len(polygonCoords) >= 3:
                    try:
                        crownGeometry = Polygon(polygonCoords).simplify(0.3, preserve_topology=True)
                        if crownGeometry.is_valid and hasattr(crownGeometry, 'exterior'):
                            exterior_coords = crownGeometry.exterior.coords
                            coords_without_last = exterior_coords[:-1]
                            readable_coords = []
                            for x, y in coords_without_last:
                                readable_coords.append([float(x), float(y)])
                        else:
                            polygons.append([])
                    except:
                        polygons.append([])

    results = {
        "polygons": polygons,
        "scores": filteredScore,
        "num_detections": len(filteredScore)
    }

    return results
```

This function is mainly used to run the inference on the image and find the polygons that compose the tree crowns. A dictionary is returned to the function caller, containing polygons, prediction scores, and the number of detected crowns.

First, the image is loaded from OpenCV, which transforms it into a multidimensional array in *ndarray* format by NumPy, the Python library par excellence for multidimensional array processing and numerical computing [**numpy**]. This allows the image to be transformed into data that has a shape, i.e. a form derived from the image, accompanied by a specific data type called dtype. More generally, in the context of machine learning, this multidimensional matrix is also called a tensor.

Subsequently, the prediction is launched thanks to the *predictor* initialized in init, directly on the image in matrix format, which returns an *instances* object containing the following parameters detected from the image:

- *pred_masks*: multidimensional matrix (tensor) of shape (N, H, W) where N corresponds to the number of detected elements, H to height and W to width, representing a mask for each detected object, therefore for each crown. Each mask highlights the pixels that belong to that specific detected tree.
- *pred_boxes*: bounding boxes for each detected object, as coordinates of the rectangle surrounding the mask, equivalent to YOLO.
- *scores*: confidence score for each detected tree.
- *pred_classes*: class of each detected object, in this case trees since Detectree2 is limited to those.

the *.to("cpu")* applied to the object containing the detected instances converts tensors to a format that can be processed by a CPU.
In lines 15, 16 and 17, the confidence filter is applied; anything that is not equal to or greater than that threshold is automatically discarded, as the *filteredScore* array contains only tree-related data detected with tolerated confidence.

The next step is then fundamental for extracting the masks of the elements considered valid, as they exceed the threshold.
Each mask detected by Detectree2 is transformed into a polygon, thanks to OpenCV's *findContours* function; it is then evaluated which is the largest (the tree may have been identified with crowns of different sizes) and it is verified that it is actually a polygon with at least three points to avoid mistakenly considering points or lines as crowns. The coordinates are extracted thanks to Shapely, which represents it with its *Polygon* type structure. Shapely is a Python library that allows the manipulation and analysis of geometric objects, such as points, lines and polygons [**shapely2025**].
An additional method is also applied to the polygon, *simplify(0.3, preserve_topology=True)*, which is fundamental for reducing the number of polygon vertices while maintaining its shape. This reduces the weight of the output, which could be really significant if there were no filter on the vertices.
Once the crown is isolated, it proceeds with the last phase which consists of parsing the external coordinates of the *crownGeometry* polygon. In this step, the last point is removed, which coin-

cides with the first to ensure polygon closure (*coords_without_last*), and a list of coordinates represented as decimal value pairs is generated. The function thus returns the identified crowns with the corresponding scores.

5.5 Comparison with DetecTree

DetecTree and Detectree2 are two solutions designed to isolate, in urban and non-urban satellite images, wooded areas or even individual trees planted in city parks. Although they share the name and target objects (trees) to which these models are dedicated, they do not use the same type of network nor the same technology stack.

The goal of this research is to identify trees; in both cases it is possible to do so, but only in the case of Detectree2 can they be isolated individually, without confusing them with other green areas, as YOLO, for example, manages to do.

In a very schematic and tabular way, it is possible to compare the two models, taking into account the definitions and technical details discussed so far:

Feature	**DetecTree**	**Detectree2**
segmentation	semantic, distinguishes between tree and non-tree pixels	object-based, detects tree crowns
model	traditional classifier (AdaBoost)	Deep Learning (R-CNN)
inputs	RGB	RGB + multispectral images
outputs	tree cover map, percentage of cover relative to the image	tree crown polygons, multiple classes
usage	tree cover analysis, urban planning	individual tree study

Table 5.1: Comparison between DetecTree and Detectree2

6 SegFormer

The advent of modern LLMs has introduced numerous innovations in the field of machine learning: although designed primarily for natural language, they have brought features also applicable to computer vision, as in the case of transformers.
These form the basis of a deep learning algorithm called SegFormer, a portmanteau of "segmentation" and "transformer", which uses transformers to classify the pixels of objects within images through what, as illustrated in the second chapter, is called vision transformer (ViT) architecture [**xie2021segformersimpleefficientdesign**].

6.1 Principle of operation

SegFormer is an algorithm that uses transformers to isolate objects, but is not limited to that: it performs true semantic segmentation, i.e. it assigns a class to each pixel of the image, in this case "tree" and "background".
The transformer implemented in this specific algorithm is a variant of the classic ViT, called MixTransformer (MiT). In addition to the MiT, there is also a decoder called Multi-layer Perceptron (MLP), which is fundamental for semantic segmentation of images.
The operation can be summarized as follows:

- MixTransformer is the backbone of SegFormer, i.e. the part of the neural network that extracts all features from the input data, namely the images. MiT, which is essentially the network encoder, implements a hierarchical structure that divides the image into parts (patches) and then processes them with transformer blocks.

- Each stage of the MiT produces feature maps that have different resolutions and depths, as seen previously for Detectron2, maintaining a spatial structure with reduced complexity compared to the ViT, which instead works directly on the entire image.
- The SegFormer decoder is an MLP, which aggregates the multi-scale features obtained by the MiT during the encoding phase, providing a complete pixel-by-pixel representation. This combines both the global context (arrangement of objects in the image, what is "in front", what is "behind" in the background) and local information (object edges).
- Aggregation occurs without convolutions, as the decoder learns to weigh and combine information received from the backbone, making predictions more robust and precise.

Having understood how SegFormer operates, it would be easy to compare this type of algorithm to CNNs or R-CNNs, already encountered in previous chapters; after all, they process images and transform them into numbers, such as tensors or matrices. The real difference is how spatial and contextual image information is processed: CNNs use convolutional filters and extract features and objects from images, analyzing small portions and following a spatial hierarchy, while ViTs divide the image into patches and treat them as if they were independent tokens, similar to the way LLMs for natural language treat words. In essence, CNNs are more effective at extracting local objects by highlighting them, such as the single crown of a tree, while transformers are more effective at capturing the overall context, using mechanisms to model global relationships between all patches.
Precisely for this reason, SegFormer is not able to distinguish one single tree from another by isolating its crown, but it can recognize where trees are in an extremely effective way [**visionplatform2025vit**].

6.2 Type of model used

In the network it is possible to download different models of SegFormer, with more or less developed variants of encoders and decoders, suitable for isolating objects of various kinds within the images. Some are also linked and available from the official repo on NVlabs, Nvidia's artificial intelligence research project[**nvlabs_segformer**].

The available and trainable models are in pt format, so they can be loaded with PyTorch as seen in previous chapters.

6.2.1 Model version

At the beginning of the chapter we saw that SegFormer is made up of encoders and decoders. However, there is not only one type of MiT encoder, but there are several versions and revisions that have been released over time to increase the precision and scalability of the model, in this case the starting model is Nvidia and is named*mit-b3*.
The MiT is released in fact in several cuts, from b0 to b5, where b3 indicates an intermediate capacity: as the version of b increases, parameters, depth and representative capacity also increase.
It is also necessary to specify that the term "starter model" is important, as it is not used as-is, which could also work if we were to find ourselves in case the need was to identify various types of objects, but it is used to create a new model after training it with a dedicated dataset as, as previously seen for YOLO, it is necessary to carry out specific training on an urban dataset to ensure that the only two classes to be recognized are background.

6.3 Preparing the dataset

Given the relatively recent implementation of SegFormer, it is not easy to acquire well-trained datasets for model training, especially if they are as specific as may be required by this research. In particular, given the requirements related to trees photographed by satellite in urban fabric, the idea was to use the YOLO dataset, which meets all the necessary characteristics, converting it to train a SegFormer model.
The concept is to use the original tiles, with the only difference of replacing the files containing the coordinates of the bounding boxes with masks, so as to turn out the dataset compatible with SegFormer. A corresponding binary mask is then generated for each image of the dataset, where the area covered by the YOLO annotations is highlighted.

The area of the mask is precisely white, but it is only to highlight for educational purposes what is the area that is considered as "covered" by trees. In fact, in this specific case, the color of the boxes would be almost imperceptible: SegFormer can be trained to recognize multiple objects at once, each object class must be assigned a color, black is commonly considered as the background color, while in the case of trees a color very close to black is assigned, almost imperceptible to the naked eye, representing the first class. These colors are not actual RGB tints, but numeric (whole) values in a grayscale or indexed image, which SegFormer interprets as class labels.

6.3.1 YOLO Dataset Converter

The conversion of the dataset can be done with a simple Python script, which is responsible for creating the masks starting from the path of the YOLO dataset, as long as it is properly organized in the val, train and test folders with images and files that define the bounding boxes, then that contain the annotations.

```python
import os
from PIL import Image
import numpy as np

def create_segmentation_mask(boxes, img_width, img_height, num_classes=1):
    mask = np.zeros((img_height, img_width), dtype=np.uint8)

    for i, (class_id, x1, y1, x2, y2) in enumerate(boxes):

        orig_coords = (x1, y1, x2, y2)
        x1 = max(0, x1)
        y1 = max(0, y1)
        x2 = min(img_width, x2)
        y2 = min(img_height, y2)

        if x2 <= x1 or y2 <= y1:
            continue

        mask_value = int(class_id) + 1
        if mask_value > num_classes:
            mask_value = 1

        mask[y1:y2, x1:x2] = mask_value
        actual_value = mask[y1, x1] if y1 < img_height and x1 < img_width else -1

    return mask
```

Several parameters are passed to the function:

- *boxes*: is the list of bounding boxes defined for the image, in the format [*class_id*, *angle_High_sx_x1*, *angle_High_sx_y1*, *angle_Low_dx_x2*, *angle_Low_dx_y2*]
- *img_width*: Input image width.
- *img_Height*: height of the input image.
- *num_Classes*: number of classes, integer value that must be set to 1 in case the class is unique, as in this.

Using Pillow[**clark2015pillow**], a library for image manipulation, an empty image of the resolution equal to that of the source image is initialized. Next for each bounding box, starting with the origin coordinates at the top left and bottom right, the relative mask is created, checking however if it is formed by a valid square or the coordinates are incorrect.
The bounding box area is then filled, taking into account the class stated in YOLO's annotation. In this specific case the*class_id*can only be 1, as there is only one type of annotated object that corresponds to the tree, while zero is used for the background.
Finally, the function returns the segmentation mask in the form of a NumPy array, which represents the image of the mask. From the mask in NumPy format it is then possible to save it in an rgb image thanks to OpenCV*cv2.imwrite(str(outputPath), mask)*.

6.4 Training Scripts

As with YOLO, to reduce the considerable training time of the SegFormer models, the idea was to resort to the transfer learning technique, especially for mit-b4 and mit-b5.
In this case it is mandatory to equip yourself with a GPU with CUDA support, any other solution that does not provide for its use could prove to be really inefficient.

6.4.1 Init

```
import os
import numpy as np
import argparse
```

```
from PIL import Image
from transformers import (
    SegformerForSemanticSegmentation,
    SegformerImageProcessor,
    pipeline
)

image_processor = SegformerImageProcessor.from_pretrained("nvidia/mit-b3")
```

In addition to the already known OpenCV, NumPy and Torch, these functions are based on the use of Hugging Face libraries.*SegformerForSemanticSegmentation* e *SegformerImageProcessor*are part of the Hugging Face transformer library, allow you to train and use the SegFormer model. Hugging Face is a well-known platform in the open source community on artificial intelligence, especially for its Transformer library and its Model Hub, where developers can share pretrained models[**wolf-etal-2020-transformers**].

Finally, it is stated as*image_Processor*the Hugging Face pre-processor optimized for the SegFormer model to use, i.e. the aforementioned*nvidia/mit-b3*. The processor performs a number of operations on input images, such as:

- transformation of the image from the original format, in this case NumPy, to the Tensor format compatible with PyTorch.
- normalization carried out on the RGB channels of the input image to be uniform to the format used in the pretrained model.

6.4.2 Parameters

```
model = SegformerForSemanticSegmentation.from_pretrained(
            "nvidia/mit-b3",
            num_labels=self.numClasses,
            ignore_mismatched_sizes=True
        )

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=40,
    per_device_train_batch_size=2,
    eval_strategy="epoch",
    remove_unused_columns=False,
)
```

The array*treeClasses*contains the classes for tree segmentation, in this case only background and tree, with its numeric ids and the colors to be associated with each class to produce the display of the masks in overlay (if implemented).

Then there are, a bit as already seen for the training of the YOLO model, the training parameters:

- *model*: It is the model, in this case pre-trained, for semantic segmentation loaded with the dedicated function of the Hugging Face library.
- *num_Train_Epochs*: Number of training epochs.
- *for_Device_Train_batch_size*: Number of images (or samples) processed at the same time.
- *eval_strategy*: Parameter indicating when to carry out the validation phase. If "epoch" is specified, it is carried out at the end of each epoch.
- *output_dir*: Output path where trained PyTorch models are saved, at the end of each epoch.

6.4.3 Training function

```
def trainModel():
    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=trainingSet
    )

    train_result = trainer.train()
    trainer.save_model()
```

This function is responsible for the actual training of the model, which takes place thanks to the Hugging Face libraries.

First, initialize the*Trainer*, a high-level class that manages model training, including:

- Forward and backward pass of the model.
- Calculation of loss and optimization of loss function.
- Periodic evaluation of metrics.
- Saving checkpoints at the end of each epoch and at the end of training.
- Logging and console monitoring of training info.

The forward pass is the process of propagating the input through the network, to get to the output output, in this specific case it is the prediction of tree cover.
The backward pass is after the prediction, the loss function that measures the error between prediction and real value mapped in the mask is calculated. In doing so, the weights are updated backwards, via an optimizer.

With*trainer.train()*training is launched, the results are then copied into*train_result*and later*trainer.save_Model*
the model to the path specified for the output. This model can then be used later for inference.

6.4.4 Output of the training procedure

```
epoch Training Loss Validation Loss Mean Iou Mean Accuracy Overall Accuracy Per Category Iou
    Per Category Accuracy
1 0.988000 0.408625 0.712276 0.807701 0.914090 [0.9053025529456493, 0.5192490975926009]
    [0.955735232841766, 0.6596663623426008]
```

The procedure produces, as for YOLO training a series of values for each epoch, which indicate the progress of the training of the model.
Training Loss e *Validation Loss*measure the quality of optimization of the model. Low loss values indicate that learning in training and validation are good.
*Mean Iou*is the average of the IOUs (Intersections over Union) calculated for each class. As seen for YOLO, it indicates the ratio of the overlap area between that affected by the model prediction and the ground truth. The result is better with a high value.
*Mean Accuracy*accuracy average for each class, while*Overall Accuracy*global accuracy on all pixels in the image.

For Category Iou e *For Category Accuracy*show an array with values for each class of IoU and Accuracy. In this case two classes are shown, "background" and "tree".

6.5 Inference

Once the model is trained it is possible to proceed to inference on satellite images. The inference procedure is also carried out thanks to the Hugging Face libraries, which in this case facilitate the prediction operation, but are not strictly necessary: the model produced in any case has a format that can be loaded by PyTorch, so to use it it it is also possible to proceed with the implementation directly published by Nvidia, or with other third-party libraries.
In this case, for consistency and practicality, the code that exploits Hugging Face libraries is shown.

6.5.1 Predictive Analysis Function

```python
def imagePrediction(image_path: str, model_path: str, confidence: float):
    image = Image.open(image_path).convert("RGB")
    processor = SegformerImageProcessor.from_pretrained("nvidia/mit-b3")
    model = SegformerForSemanticSegmentation.from_pretrained(model_path)

    inputs = processor(image, return_tensors="pt")
    with torch.no_grad():
        outputs = model(**inputs)

    logits = outputs.logits

    probabilities = torch.nn.functional.softmax(logits, dim=1)

    tree_prob = probabilities[0, 1].cpu().numpy()
    mask = (tree_prob > confidence).astype(np.uint8)
```

The prediction function accepts two parameters in input,*image_path* e *model_path*, respectively image path and trained model. It is also required to specify the confidence threshold which, as with other models seen so far, is the limit within which a detected object is considered good or not.
Next it is necessary, as previously for the dataset, to preprocess the images thanks to the*SegformerImageProcesso*

where also in this case the "origin model" that establishes the data format should be specified. Thanks to this preprocessing you have a great advantage because it is possible to provide in input images also of different resolution, since it will then be the processor to standardize them. The actual loading of the trained model takes place in line four, and the next line is then effectively processed the image.

The inference is launched on the image thanks to the call that the library exposes, directly in the class that manages the model*model(**inputs)*and provides some important information in output:

- *.logits*: raw scores of the prediction.
- *.loss*: value of loss function.
- *.attentions*: attention weights, i.e. the numerical values that are assigned to the subjects "observed" by the model, for example for a tree could be 0.8 for the leaves, 0.6 for the trunk. The closer to 1, the more important the value.

In this specific case, to define the mask that isolates trees, the scores are sufficient, which are passed to a PyTorch function called activation. This particular function, Softmax, takes raw score data that assigns a score to the pixel, such as attention weights and is converted into a probability, i.e. the more score a pixel totals, the more likely it is to be a tree.

The last two instructions are those that extract the binary mask containing the prediction result "tree" or "non-tree", extracting the probabilities of the class "tree" with NumPy and discarding those that are below a certain confidence threshold.

6.5.2 Quality of Prediction

In order to display the result of the prediction it is then possible, with OpenCV go to superimpose the mask to the original image. The implementation is not shown here but can be found, with some variation, on the original repo of SegFormer in the tools and demos.

It is significant to see the result of the prediction that is carried out by SegFormer, in fact the graphic representation highlights all the model data such as the tree cover, with the relative binary mask that goes to outline the contours of the urban areas planted and also the probability, in the form of heatmap that represents the probability distribution of the model.

It is interesting to note that, unlike DetecTree, the areas covered by trees are highlighted but not

because of the simple proximity of pixel color as seen precisely with the AdaBoost algorithm: in this case not even the shadows of the trees themselves, which stand out on the Adige river, are included in the mask.
The result of the prediction includes the binary mask that outlines the contours of the planted areas and a heatmap that represents the probability distribution of the model, allowing you to accurately visualize the areas identified as tree cover.

6.6 Watershed Application

SegFormer is very effective at isolating groups of trees within an image, but not being able to distinguish them on time. In this regard it is possible to use a post-processing technique called watershed, therefore downstream of SegFormer's detection[**kornilov2018watershed**].
This is a technique based on mathematical morphology, which analyzes the image by separating adjacent objects in a completely agnostic manner regarding their content: it is not able to distinguish what is tree and what is not but, starting from the isolated areas thanks to SegFormer, the only type of insulating objects will be implicitly trees, thus allowing to identify the foliage and the number of trees present.
Summing up the operation in brief:

- The algorithm takes an input grayscale image.
- The image is interpreted as a surface, where pixels represent altitudes.
- You select pixels or starting regions called markers, which can identify the labels of the lowest points, the local minima, or the highest ones like the local maxima. In our case it will be the maximums, that is, the tips of the trees, that will be identified.
- The image is segmented by a floating process, hence the term watershed (filling basins), where the regions are "filled" from markers.
- In each step the pixels bordering the floated regions are sorted according to the gradient value, placing them in a priority queue: the pixel of minor (or greater in the case of maximums) value is assigned to the basin of the nearest marker.
- The boundaries between the basins constitute the watershed line, i.e. the border between the different elements of the image.

6.6.1 Segmentation function

Here a function is reported that deals with finding regions and elements thanks to the watershed process, starting from a mask supplied in input as a parameter.
To make the operation more efficient, you can take advantage of two well-known libraries in Python, such as Scikit and SciPy: the first is used for image processing and has already been introduced in the chapter on Detectree, while the second is an open-source library that offers numerous data processing tools, including optimization algorithms, interpolation and signal processing[**2020SciPy-NMeth**].

```python
from scipy.ndimage import distance_transform_edt, label
from skimage.feature import peak_local_max
from skimage.morphology import watershed
from skimage.measure import regionprops
import numpy as np

def watershedSegmentation(mask, minDistance, offsetX=0, offsetY=0):
    distanceMap = distance_transform_edt(mask)

    peakCoords = peak_local_max(
        distanceMap,
        min_distance=minDistance,
        labels=mask,
        footprint=np.ones((3, 3))
    )

    markers = np.zeros_like(mask, dtype=np.int32)
    for i, (y, x) in enumerate(peakCoords, start=1):
        markers[y, x] = i

    labelsWatershed = watershed(-distanceMap, markersLabeled, mask=mask)

    treeCrowns = []
    for treeId in range(1, labels.max() + 1):
        crownMask = (labels == treeId)
        treeCrowns.append(crownMask)

    return treeCrowns
```

The function*distance_Transform_edt*of SciPy is used to create the distance map, which is subsequently provided in input to the function*peak_Local_max*, belonging to the scikit-image library, to locate markers.

Then the coordinates of the peaks are transformed into a numerical marker map, where each marker has a unique label assigned to the growth of the watershed basins.

Then it is invoked*watershed*, scikit-image function, which progressively "floods" the image starting from the markers, and then stops only when the boundary lines meet, thus detecting the boundaries of the objects.

Finally, the masks of the objects detected and stored in the array are isolated*treeCrowns*, which can be used for area, perimeter or overlay display on delimited objects.

6.6.2 Segmentation Result

The watershed segmentation process produces three progressive outputs: starting from the source satellite image, the SegFormer mask is first generated that highlights the detected tree areas in green, and then the watershed segmentation is applied that distinguishes the individual crowns of the trees, identifying them with different colors. This allows each individual tree to be counted and located within the areas previously identified as planted.

7 Application of the 3-30-300 rule and tree map

The previous chapters deal with the analysis of models and networks that, properly trained and parameterized, manage to isolate green areas and trees.
The research is focused in particular on the detection of trees present in the urban fabric, not only to compare and evaluate the performance of the models themselves, but also to try to derive in a completely automatic way the coordinates of each tree, in latitude and longitude, so that it can be mapped on information systems such as GIS.
The Geographic Information System (GIS) is a technology that allows to analyze georeferenced data to display cartographic elements organized in overlapping layers, allowing to obtain structured visual representations from raw data[**gisUsgs**].

7.1 Rule of 3-30-300

The ultimate goal of this research is therefore to produce a data stream, properly formatted according to the GIS software or the implemented solution, that can contain a precise geographical position of the trees in the city of Verona.
This will then be used to calculate a particular indicator, in reference to the 3-30-300 rule.

7.1.1 Definition

The 3-30-300 rule is a principle devised by Professor Cecil Konijnendijk, which defines the criteria for ensuring access to green in cities, so as to make them more livable and more sustainable[**Konijnendijk2023**]. The three criteria are:

- 3 visible trees: each resident should be able to see at least 3 trees from their own window.
- 30%of tree cover in the neighborhood: at least 30%of the area of the neighborhood should be covered by trees, considering the foliage.
- 300 meters away from the nearest green space: each resident should have a park at most 300 meters from their home.

7.1.2 Data collection

To be able to calculate the indicator 3-30-300, which is the subject of another specific research on the subject, it is therefore no longer enough to be content with detecting trees, but instead it is necessary to create a real map from satellite images, so that you can locate each tree in a distinct way and with a geographical location.

7.2 Map of trees

The models analyzed so far, to identify trees from satellite images, return images with annotations and bounding boxes to highlight the confidence and position of trees in the image itself: would it be possible to use this type of information, to also identify the position of trees in a not only visual but also geographical way? To answer this question it is necessary to familiarize a particular image format, which precisely combines coordinates of the image with geographical data: the GeoTIFF.

7.2.1 TIFF and GeoTIFF

TIFF is a popular raster image format, which originated as a method for exchanging images between printers and scanners, given its feature of being able to support properties and data related to the image itself. On the one hand it leads this format to be very flexible, on the other hand the fact that it can support many compression schemes and features, it leads TIFF to be complicated to use because the application that has to read it must be able to fully support the format[**tiff**].

GeoTIFF leverages the flexibility of TIFF to include geographic location metadata, thus becoming a de facto standard for organizations and the community of people working with geospatial data[**geotiff**].
This type of format is also supported by GIS systems.

7.2.2 From JPG to GeoTIFF

Unfortunately, in many cases satellite maps available on location platforms, commercial or open source, are in vector format or JPG or PNG raster and it is therefore impossible to use that type of images in a georeferenced context. For this objective, the simple satellite images used for tree detection are not enough to extract usable data to do analysis on maps and indicators via GIS.
To overcome this problem, Rasterio, a Python library that not only can read GeoTIFF formats, but is also able to manipulate them by associating a specific pixel position with a spatial position in terms of latitude/longitude coordinates, comes to our aid.[**rasterio**]. Rasterio leans on the well-known GDAL[**GDAL**], an acronym for Geospatial Data Abstraction Library, in fact offers the Python interface to go and take advantage of the powerful library in C++ that processes raster files.

7.2.3 Pixel-coordinate conversion

Satellite map providers usually offer endpoints that allow you to download small "tiles" of a map, one at a time, called tiles: it would be unthinkable in fact to download the entire map of an entire city. In order to access the tiles you need to specify three parameters, namely the zoom and the lat/long coordinates of the northwest corner of the tile, in doing so you will have a small tile (e.g. 512x512) but which has a precise geographical position.
To arrive at the result it is possible to proceed in two ways:

- Transforming tiles into GeoTIFF: If the model to be used for tree detection accepts into input images that have the same resolution as downloaded tiles, you can transform the image directly into GeoTIFF with Rasterio.

- Merge the tiles into a larger image and convert it to GeoTIFF: if the model has specific resolution inputs other than those of the downloaded tiles, then it is necessary to merge all the images and then cut it into the tiles of the correct size, first converting to GeoTIFF the "collage" image derived from the composition of the other tiles.

7.2.4 GeoJSON detection and generation

Once the tiles have been converted to GeoTIFF it is possible to proceed with the inference on each individual tile and relate the position in pixels to the geographical position, an operation possible thanks to Rasterio.
After the detection process is finished, in order to share the precise coordinates of the trees present in the analyzed images so that they are usable by GIS, the solution indicated is to export a GeoJSON. The latter is precisely a particular format of JSON that is used for the exchange of geospatial data: a GeoJSON can describe an object detected on the map as a geographical point, with also explanatory properties, such as the class of the object represented or the name.
It also supports geometry types, such as points, lines and polygons, making it also indicated to represent for example the bounding boxes returned by YOLO[**GeoJSON**].

7.3 Conversion to GeoTIFF with Rasterio

To merge and convert tiles using Rasterio, you need to use Python. Unlike the solutions previously seen for model training, in the case of operations with images and simple inference for tree detection, it is not strictly necessary to use a GPU: it is enough to launch the scripts on your machine taking care to enhance the correct parameters.

Below are some snippets of code, which make up the important parts of the tile processing scripts. It also presents the inference phase with YOLO11 which, while representing one of the possible alternatives for tree detection, allows to illustrate the entire process up to the generation of GeoJSON files. This code is responsible for saving a JPG or PNG raster image, in a GeoTIFF image. Before arriving at the save function, in the full script there is a list of commands that are used to extract the zoom level, latitude and longitude from the filename, these lines are left out as they are of trivial implementation.

More interesting, however, is to observe how these values are converted into tile coordinates, as a matrix of squares, an operation necessary to create bounds to be saved later as GeoTIFF.

7.3.1 Lat/long conversion and zoom into slippy coordinates

```python
def deg2num(lat_deg: float, lon_deg: float, zoom: int) -> Tuple[int, int]:
    lat_rad = math.radians(lat_deg)
    n = 2.0 ** zoom
    x = int((lon_deg + 180.0) / 360.0 * n)
    y = int((1.0 - math.asinh(math.tan(lat_rad)) / math.pi) / 2.0 * n)
    return (x, y)
```

*deg2num*is a standard conversion function for this type of operation, in the network you can find others that use third-party libraries, but very important to understand the mechanism of conversion to tile coordinates.

This type of tile coordinates are also called "slippy"[**osm-slippy-map-tilenames**], which refers to the type of map that we all consult on the web, which allows you to move ("to slip" means "slip") and enlarge/shrink the display, with zoom (e.g. Google Maps).

The conversion of these coordinates takes place through the projection called Web Mercator, which is a "digital" variant of the cartographic projection of Mercator, cartographer and astronomer who in the 16th century converted the spherical Earth's surface to a flat surface, thus creating the maps that we consult even today.

7.3.2 Calculating geographical boundaries

```python
def calculateBounds(x: int, y: int, zoom: int) -> Tuple[float, float, float, float]:
    n = 2.0 ** zoom
    west = x / n * 360.0 - 180.0
    east = (x + 1) / n * 360.0 - 180.0

    north_rad = math.atan(math.sinh(math.pi * (1 - 2 * y / n)))
    north = math.degrees(north_rad)

    south_rad = math.atan(math.sinh(math.pi * (1 - 2 * (y + 1) / n)))
    south = math.degrees(south_rad)

    bounds = (west, south, east, north)
```

This part of code is critical to finding what are called image "bounds". In this specific case, the tiles downloaded via APIs by the map provider, indicate as the lat/long coordinate of the north-west center of the tile, it is necessary at this point to isolate which are the bounds in the four cardinal points, so as to find what are the geographical limits of the tile:

- *West*: longitude of the left (west) edge of the tile
- *south*: latitude of the bottom (south) edge of the tile
- *east*: longitude of the right (east) edge of the tile
- *north*: latitude of the top (north) edge of the tile

These four bounds will then be needed by Rasterio in order to create the GeoTIFF.

7.3.3 GeoTIFF saving

```
def saveGeotiff(bounds: Tuple[float, float, float, float], outputPath: str) -> bool:
    west, south, east, north = bounds
    outputImage = Image.new('RGB', (TILE_SIZE, TILE_SIZE))

    transform = Affine.translation(west, north) * Affine.scale((east - west) / width, (south -
        north) / height)

    imageArray = np.array(outputImage)
    imageArray = np.transpose(imageArray, (2, 0, 1))

    with rasterio.open(
        outputPath,
        'w',
        driver='GTiff',
        height=height,
        width=width,
        count=imageArray.shape[2],
        dtype=imageArray.dtype,
18          crs=CRS.from_epsg(4326),
19          transform=transform,
20          compress='lzw'
21      ) as dst:
22          dst.write(imageArray)
```

The function for saving the image as GeoTIFF takes as input the bounds from *calculate_bounds* and the *output_path* where to save the image. First, an RGB image must be created that can contain the tile, which is instantiated with Pillow.
The image is then passed to the Rasterio library as an array through a conversion that takes place thanks to NumPy.
Rasterio, in order to correctly write the data in the GeoTIFF file, expects an array so structured:

- shape: bands, height, width
- numeric type: uint8, float32...

To proceed with the writing of the image, in line 4, a particular transformation matrix must first be made:

- *Affine.translation(west, north)*: is a translation transformation: the upper left corner of the image corresponds to the northeast in geographical coordinates.
- *Affine.scale(...)*: executes a pixel scale, where each pixel, on the horizontal axis, is*(east - west) / width*degrees of longitude, while each pixel, on the y (vertical) axis, is worth*(south - north) / height*degrees of latitude.
- Multiplication: Gets a related matrix that allows Rasterio to connect each pixel of the image to the exact georeferenced position.

The last operation before writing the GeoTIFF is to transpose the axes of the image. This change is necessary because the Pillow RGB image uses the format (height, width, bands), while Rasterio requires the format (bands, height, width), with the number of bands (three in the case of the RGB format) specified as the first parameter. Then it is enough to invoke Rasterio, providing all the necessary parameters to make sure that the GeoTIFF in output is perfectly compatible with GIS systems:

- *outputPath*: path of the output file

- *'w'*: file opening mode, in this case write
- *driver*: type of raster format to use, "GTiff" is our GeoTIFF
- *height*: number of lines in pixels of the image
- *width*: number of columns in pixels of the image
- *count*: number of image bands
- *dtype*: type of pixel data (e.g. uint8, float32)
- *crs*: Coordinates Reference System, in this case sets the WGS84 system (lat/lon, GPS standard).
- *transform*: affine transformation matrix that connects pixels to geographical coordinates (bounding box + resolution).
- *compressed*: compression algorithm for the file

7.3.4 Tree Detection

```
def analyzeGeotiffTreeCoverage(imagePath, modelPath, conf=0.25):
    with rasterio.open(imagePath) as src:
        transform = src.transform

        imageArray = src.read([1, 2, 3])
        imageArray = np.transpose(imageArray, (1, 2, 0))

        crs = src.crs
        width = src.width
        height = src.height
```

As already mentioned earlier there are many ways to make the detection of trees, here is the code for inference with YOLO by way of example.

The script performs, at least in part, the reverse transformation process from Rasterio image to NumPy. This is because YOLO asks for an RGB image in input, so with the channel matrix organized in a standard way. Pillow is used again for conversion.

```
model = YOLO(model_path)
results = model(img_array, conf=conf)
```

The inference on the image is then launched.

The code for this type of inference differs from that used for simple inference with YOLO. While the latter identifies the position of the tree in the image by returning an image with the bounding boxes highlighted through OpenCV, in this case the detected position is converted to geographical coordinates (latitude/longitude) corresponding to the actual location of the tree.

```
for result in results:
    if result.boxes is not None:
        boxes = result.boxes.xyxy.cpu().numpy()
        confidences = result.boxes.conf.cpu().numpy()

        for box, confidence in zip(boxes, confidences):
            if confidence >= conf:
                x1, y1, x2, y2 = box.astype(int)

                centerX = (x1 + x2) / 2
                centerY = (y1 + y2) / 2

                lon, lat = transform * (centerX, centerY)
```

This is how the results are analyzed, in a similar way as seen in the YOLO chapter but in this case, thanks to the transformation matrix provided by Rasterio, the function is able to derive the geographical coordinates from the precise point in pixels of the tree detected in the image.

At this point we can build, thanks to all the parameters collected by the detection in the image and the conversion of the pixels to lat/long, the GeoJSON that will feed GIS.

In the case of DetecTree2, the output of the prediction to be written in the GeoJSON is not represented by a point, but by a polygon that delimits the entire crown of the tree.

7.3.5 GeoJSON Generation

Gather all the latitude and longitude coordinates of the trees detected in the images, the last step involves creating the GeoJSON, following this format to maintain compatibility with GIS software[**GeoJSON**]:

```
{
"type": "FeatureCollection",
"crs": {
    "type": "name",
        "properties": {
            "name": "EPSG:4326"
        }
    },
"features": [
    ...
    ]
}
```

A GeoJSON file must explicitly specify the type of annotations it contains. In this case, it is*FeatureCollection*, indicating the presence of an array of geographic elements such as polygons or points.

The crs property, on the other hand, indicates the type of coordinates that are specified in features such as EPSG:4326, also known as WGS84, which is equivalent to the classic notation of latitude and longitude with decimal degrees.

Finally the features array will contain a collection of objects, i.e. the points corresponding to the position of the trees, which have this format:

```
{
    "type": "Feature",
    "geometry": {
        "type": "Point",
        "coordinates": [
          10.986155775846253,
          45.445288495847386
        ]
    },
    "properties": {
        "tree_id": 0,
        "confidence": 0.810298502445221,
        "image_name": "image.tif",
        "image_path": "/data1/data2/image.tif",
        "source_crs": "EPSG:4326",
        "bbox_x1": 600,
        "bbox_y1": 84,
        "bbox_x2": 634,
        "bbox_y2": 123,
        "pixel_x": 617.0,
        "pixel_y": 103.5
    }
}
```

The declaration of the "Feature" type is fundamental, where the type of geographical feature is specified, in this case Point, with its coordinates in the format declared by the parameter of the dedicated json.

It is also possible in a GeoJSON to declare custom properties, which contain data related to the declared feature. In the case of YOLO for example it makes sense to report a numerical id of the detected tree, the level of confidence, the path of the image and the coordinates of the bounding boxes.

It is important to point out that properties are not mandatory anyway, only the type and geometry of the feature are.

7.4 Result on QGIS

By importing the GeoJSON thus generated onto QGIS, an open source GIS software, the detected trees appear as georeferenced points on the map, allowing the tree cover distribution to be displayed.

8 Analysis of results

In the previous chapters, different algorithms have been treated that allow, starting from satellite images, to derive the position of the trees both as coordinates in pixels and geographical, finally exporting a GeoJSON that defines in a more or less precise way the shape of the crown or the bounding box that delimits the tree itself. Other algorithms, such as SegFormer and DetecTree, are instead able to isolate groups of trees.
These two modes are interesting as they are applicable for calculating the indicator dealt with in chapter 7, where both the distance from individual trees and the distance from parks or planted green areas is considered.
However, in order to verify which of the models achieved the best results, it is necessary first of all to identify the real geographical position of the trees present on the urban fabric.

8.1 Ground Truth Dataset

To proceed with the verification it is essential to be able to access a dataset containing the position in latitude and longitude of the planted trees, in other words our ground truth, or the verified reference data. Otherwise it is impossible to compare the accuracy of the various models seen previously, which are based only on visual detection.

8.1.1 Access to public databases

In some cases the region, municipality or third-party entities provide APIs or datasets that map trees and/or plantings. When this type of information is easily accessible, the process of building the ground truth database is greatly simplified, since it is already available as validated and clean data, i.e. filtered by possible false positives or false negatives.
It is necessary, however, to consider that the trees mapped in these datasets, could contain only the areas and trees that make up the vegetation on public soil, therefore not considering private planted properties that are detected instead by satellite images alone.
The format of the data varies by public body, region or municipality: in some cases the simple coordinates are provided in JSON/CSV, in others in GeoJSON ready to be imported into GIS applications, in others they are in the form of a CHM package. The latter is a particular format that is generated by LIDAR (Light Detection and Ranging) sensors placed on aircraft that generate, thanks to laser pulse technology, a 3D representation of the territory and the crowns of the trees.

8.1.2 Manual Annotation

Another alternative is to create a dataset manually, with applications such as QGIS, importing the reference satellite image and annotating the coordinates in a point and click on a layer, later exportable to GeoJSON.
This solution makes it possible to annotate all trees, on public and non-public land, but it is extremely time-consuming. It is also crucial to verify that the satellite image used in the software (e.g. a GeoTIFF file) is correctly georeferenced, since an incorrect georeference would result in an inaccurate annotation of the geographical coordinates of each tree detected.

8.2 Templates Benchmark

Once the ground truth is defined, it is possible to proceed to the comparison with the predictions generated by the models: according to the source of the data it is necessary to establish the comparison parameters on which to calculate the detection quality metrics.
A GeoJSON made available by the municipality of Milan was used for this analysis[**ComuneMilanoAlberiGeo**
which maps all the trees planted on public land within the municipal boundaries. For reasons of

cost and computational complexity, the detection was carried out on the satellite map measuring 3km radius from the center of Milan. Data is freely available under the Creative Commons license

8.2.1 Requirements

As previously mentioned, this GeoJSON from an institutional source exclusively includes trees on public areas, excluding those on private land. It is therefore not possible to accurately determine actual false positives, as numerous trees absent from the municipal dataset may have been detected correctly by previously analyzed models, thus creating an overestimation of detection errors.
In addition, the data provided by the municipality map trees as single georeferenced points instead of as polygons so, to make this comparison possible, a tree is considered correctly detected when the georeferenced point falls within a YOLO bounding box or segmentation polygon (which represents the canopy for DetecTree2 or the planted area for SegFormer). Trees whose points do not fall into any polygon are classified as undetected.
To properly compare models, however, it is necessary to distinguish between those that identify individual trees (such as YOLO and DetecTree2) and those that identify larger areas, such as plant areas or green areas (SegFormer). This distinction is crucial since different detection approaches require different assessment methodologies.

8.2.2 Model Parameters

The results are based on detections made with the following confidence parameters:

- YOLO11: 0.416, which as defined by the F1 curve after training is the value that perfectly balances precision and recall. To compensate for the accuracy of the detection, an 11m proximity buffer is applied around the bounding boxes.
- Detectree2: 0.25, defined for a detection that favors higher recall values but reduces missed detection.
- SegFormer: 0.30, offers the best balance between precision and recall, detected after a number of tests (SegFormer does not have a global F1 value, it works at the pixel level with probability per class).

- DetecTree: has no confidence parameterization, it also performs a pixel classification, but in a different way from SegFormer.

8.2.3 Detection Script

The comparison function between the GeoJSON file of the municipality of Milan, used as ground truth, and the polygons predicted by the models is relatively simple and linear:

```python
import geopandas as gpd

def calculateTreeCoverage(groundTruthPath: str, modelTreesDetectedPath: str):
    trees = gpd.read_file(groundTruthPath)
    detections = gpd.read_file(modelTreesDetectedPath)
    detectedTrees = set()

    for _, polygon in detections.iterrows():
        treesInside = trees[polygon.geometry.contains(trees.geometry)]
        detectedTrees.update(treesInside.index)

    coveragePercentage = (len(detectedTrees) / len(trees)) * 100
```

In the first two lines, files are loaded into two GeoPandas dataframes, a Python library that allows data to be processed in tables and csv, with support for geospatial data management and analysis[**kelsey_jordahl_2020_3946761**].
After initializing a set to contain the actual trees detected, the data is crossed: for each polygon detected by the machine learning model, the geometric operation is used*contains()*, which isolates all ground truth trees contained within the detection range. The indices of these trees are then added to the set*DetectTrees*.
At this point the detection percentage is nothing more than the ratio of unique trees detected to the total of trees in the ground truth.

8.3 Tree area detection

In this case there are two algorithms that return areas populated by trees, instead of single crowns: for completeness DetecTree was included.

8.3.1 SegFormer

Metric	Value
Detected trees	25,029
Undetected trees	19,365
Polygons with trees	4,076
Polygons without trees	9,202
Detected trees percentage	**56.38%**

Table 8.1: SegFormer detection results

SegFormer is very effective at detecting trees in urban settings and, considering that the starting dataset is the same as the one used for training the YOLO11 model, is significantly more precise when we consider the coverage area instead of the individual trees, demonstrating good selectivity in detecting actually planted areas.

8.3.2 DetecTree

Metric	Value
Detected trees	17,563
Undetected trees	27,294
Polygons with trees	8,006
Polygons without trees	436,865
Detected trees percentage	**39.2%**

Table 8.2: DetecTree detection results

DetecTree is confirmed to be able to detect the majority of trees but, as introduced in the dedicated chapter, for tree cover alone it is extremely imprecise because it mixes the crowns with the green areas. To demonstrate this, there is precisely the fact that green areas divided into more than 400 thousand polygons have been identified by the algorithm; of these, only 1.8% contain trees: certainly some polygons will contain plantings in private areas, but the majority will contain generic "urban green".

8.4 Detecting individual trees

Let's now see the results on the data inferred by the two object detection models, which exploit convolutional networks: as previously mentioned they work on the features/characteristics of the image, to be able to isolate the objects in them.

8.4.1 YOLO11

Metric	**Value**
Detected trees	22,956
Undetected trees	21,438
Polygons with trees	10,817
Polygons without trees	11,861
Average trees per polygon	2.62
Detected trees percentage	**51.71%**

Table 8.3: YOLO11 detection results

In this case the polygons are derived from the bounding boxes which, being designed to highlight the detected object, do not constitute the actual crown of the tree.
Considering only the trees present in the GeoJSON of Milan, we have an average of 2.62 trees per detection: i.e. 24.2% of the bounding boxes contain a single tree, while 71.0% contain small groups of 2 to 5 trees, demonstrating good accuracy in the detection of individual trees and small groups.

8.4.2 DetecTree2

Metric	**Value**
Detected trees	10,355
Undetected trees	34,039
Polygons with trees	5,216
Polygons without trees	9,659
Average trees per polygon	2.01
Detected trees percentage	**23.33%**

Table 8.4: DetecTree2 detection results

DetecTree2 is a complicated model to balance, as at low confidence values it is able to detect more trees but introduces an important number of false positives. At this confidence value, identified as good after various inference tests on satellite images, it seems to be significantly less effective than the other two: probably the fact of being in an urban context does not help it, in fact it seems to be much more suitable for forests or forest ecosystems[**Ball2022DetectTree2Documentation**].
Considering the trees it was able to successfully detect, we can still notice an average of 2.01 trees per polygon; thus DetecTree2 shows a good ability to segment single crowns: 49.3% of the polygons contain a single tree and 47.1% contain small groups of 2 to 5 trees.

8.4.3 SegFormer + Watershed

Watershed is an image segmentation technique that allows you to separate connected objects. It has been applied to the results of SegFormer to attempt to subdivide tree areas into individual trees.

Metric	Value
Detected trees	24,280
Undetected trees	20,114
Polygons with trees	1,917
Polygons without trees	1,845
Average trees per polygon	13.87
Detected trees percentage	**54.69%**

Table 8.5: SegFormer + Watershed detection results

Watershed application to SegFormer results slightly worsens performance: detection rate drops from 56.38% to 54.69%. This suggests that attempting to further subdivide the detected areas may introduce some errors.
With an average of 13.87 trees per polygon, the SegFormer + Watershed algorithm mainly detects areas with tree groups: only 5.6% of the polygons contain a single tree, while 39.9% contain groups of more than 10 trees.

8.5 Comparative comparison

The models analyzed show distinctive features depending on their architecture and detection approach.

8.5.1 Detecting Single Trees

For the detection of individual trees, the models with the best performance in terms of precision are:

Algorithm	Coverage	Trees per polygon
YOLO11	51.71%	2.62
DetecTree2	23.33%	2.01
SegFormer+Watershed	54.69%	13.87

Table 8.6: Comparison of algorithms for single tree detection

YOLO11 and DetecTree2 are distinguished by their ability to identify individual trees (average 2-3 trees per detection), with DetecTree2 particularly precise in the segmentation of single crowns (49.3% detection with only one tree). YOLO11 with an 11m buffer achieves a detection rate of 51.71% while maintaining an average of 2.62 trees per detection, making it very effective for individual tree detection. SegFormer+Watershed, despite having a slightly higher detection rate (54.69%), mainly identifies groups of trees (average 13.87 trees/detection), making it less suitable for applications that require precise counting of individual trees.

8.5.2 Tree area detection

To identify planted areas and tree groups, SegFormer base is the most effective:

Algorithm	**Detection rate**
SegFormer	56.38%
DetecTree	39.2%

Table 8.7: Comparison of algorithms for tree area detection

SegFormer offers the best detection rate (56.38%) with its ability to accurately identify tree groups, while DetecTree has an excessive false positive rate (98.19%), which makes it unsuitable for practical tree detection-only applications.

8.6 Analysis and improvements

The results achieved by the various models denote a decent ability to detect trees: SegFormer, the best performing model among the three, detects just under 57%of the trees planted on the municipal territory. This is a significant percentage for this specific task, notoriously complex given the need for high-quality satellite imagery to achieve good detection levels. The most common factors that can compromise results are related to the quality of the training dataset and the type of input images used for inference.

8.6.1 Species of trees in the Lleida dataset

The dataset used for training the two models that returned the best feedback in terms of coverage comes from studies conducted in the urban area of Lleida, Spain, previously introduced in the chapter dedicated to YOLO. The documentation on the tree species of the territory of

Lleida[**paeria_lleida_especies_arbories**]highlights the presence of some species typical of the Mediterranean and river environment, some of which may not be fully representative of the Milanese urban ecosystem.
The climatic and biogeographical difference between Lleida and Milan could influence the effectiveness of the model trained on this dataset when applied to the Lombard context, characterized by a temperate continental climate and a different urban tree composition.

8.6.2 Proximity buffer

As we have seen, a model like SegFormer is able to give an acceptable result in terms of detection. In order to further improve performance, there is a methodology applicable in the post-processing phase, which consists of implementing a proximity buffer around existing surveys[**zhang2018proximity**].
To evaluate whether to apply a buffer on the detection carried out by SegFormer, that is, the most performing, it is necessary to identify the problem: the 74.0%of the missed trees is within 50m from the already mapped detections, so they are visually present but not detected due to:

- Shades masking trees, caused by the time satellite images were taken.
- Edges of the crowns where segmentation stops prematurely.
- Overlays between adjacent trees that confuse the neural network.

By detecting a tree at a given location, it is statistically likely that there are other trees in its vicinity that the model has not seen: the buffer helps to recover false negatives.
The results with this spatial post-processing procedure manage to detect 38,304 trees compared to 25,029, bringing the detection rate of trees in the Milan municipal territory to 86.28%.

8.6.3 Performance Assessment

This detection approach using deep learning + spatial post-processing, which achieves 86.28% tree detection, far exceeds acceptable standards for urban tree detection (typically 60-80%) and should be considered an excellent result for several reasons:

- High urban density with complex overlaps between buildings and vegetation.
- Diversity of tree species with varying morphologies.
- Urban shadows and occlusions due to city architecture.
- Intensive pruning that modifies the natural shapes of the crowns.

Manual verification work is reduced to 13.72% of the total dataset, representing significant time and resource savings for urban forest management applications.

9 Satellite map services

In order to make the inference on the machine learning models described in the previous chapters, it is necessary to have satellite images of adequate quality. The market offers different platforms for the acquisition of this type of images, each with different technical characteristics, tariff plans and limitations of use.
This chapter analyzes the main services available, comparing them according to criteria relevant to automatic tree cover detection applications: maximum resolution (zoom level), download possibilities, compatibility with ML inference and costs.

9.1 MapTiler

MapTiler is a platform that offers satellite map services with excellent resolution, particularly in Europe and the USA.

9.1.1 Tariff plans

- **FREE**: \$0/month (limited non-commercial use) - 100K credits
- **FLEX**: from \$25/month - 500K credits
- **UNLIMITED**: \$295/month - 5M credits
- **CUSTOM**: on request - annual/quarterly billing, volume discounts

9.1.2 Technical Details

- **Maximum Zoom**: up to z20
- **Download limits**: limited FREE plan, paid plans with quotas based on tile requests (1 satellite tile 4 credits)
- **ML Inference**: allowed via API
- **Download**: not allowed (except on-premise custom licenses)

MapTiler also offers on-demand imagery satellite via partnership with Satellogic. This platform was used for inference on the 24 Italian cities being studied.

9.2 Stadia Maps

Stadia Maps is a map provider with good global coverage, but with restrictive terms for machine learning applications.

9.2.1 Tariff plans

- **FREE**: $0/month - 200K credits (50K satellite tiles) - NO commercial use, NO satellite imagery
- **STARTER**: $49/month - 2M credits (500K satellite tiles) - NO satellite imagery
- **STANDARD**: $249/month - 15M credits (3.75M satellite tiles) - satellite imagery included
- **PROFESSIONAL**: $799/month - 60M credits (15M satellite tiles) - free 14-day trial
- **ENTERPRISE**: custom - billions of credits/year - SLA, on-premises, perpetual licenses

9.2.2 Technical Details

- **Maximum Zoom**: z18-19
- **Download limits**: based on tile requests (tiles 512x512px, 1 satellite tile 4 credits)
- **ML Inference**: not allowed (commercial use requires license)

- **Download**: not allowed

Despite the good quality of the images, the terms of service do not allow use for machine learning applications.

9.3 LandViewer (EOSDA)

LandViewer, developed by EOS Data Analytics, is a platform oriented to the analysis of satellite images with integrated advanced tools.

9.3.1 Tariff plans

- **FREE**: $0 - 10 downloads/day of medium resolution images (Sentinel, Landsat), full access to analysis tools, 256GB personal cloud storage, 20+ pre-set vegetation indices, unlimited viewing
- **PREMIUM**: price on request - monthly/annual, increased download limits, advanced features, priority support
- **HIGH-RES (On-Demand)**: pay-per-km^2 - Airbus Archive 50cm: $3.80/km2 (no minimum order), Airbus Archive 30cm:$18/km2 (minimum order 25km2)

9.3.2 Technical Details

- **Maximum Zoom**: z19-20+ (up to 0.30m/px with high-res)
- **Download limits**: FREE 10 downloads/day, Premium and High-Res according to plan
- **ML Inference**: allowed
- **Download**: allowed (GeoTIFF and various formats)

The platform includes analysis tools such as NDVI and change detection, with data sources such as Sentinel-2, Landsat, MODIS, KOMPSAT, SuperView and Gaofen. However, it is extremely expensive for high-resolution images: for a medium-sized city the cost can be around $2000.

9.4 Google Maps Platform

Google Maps Platform provides excellent coverage globally with excellent resolution.

9.4.1 Tariff plans

- **FREE**: 10K-100K free calls/month for SKU (since March 2025)
- **Pay-as-you-go**: after the free limit - Dynamic Maps $7/1000 requests, Static Maps $2/1000 requests, Map Tiles API various tiers

9.4.2 Technical Details

- **Maximum Zoom**: variable z18-22 (depends on the zone, verifiable via MaxZoomService)
- **Download limits**: based on API calls
- **ML Inference**: not allowed
- **Download**: not allowed directly

Despite the excellent resolution and the presence of a watermark on each tile, the terms and conditions do not allow use by inference with machine learning models.

9.5 Mapbox

Mapbox is a widely used platform for web and mobile applications, with excellent image quality.

9.5.1 Tariff plans

- **FREE tier**: available
- **Paid plans**: from$50/month - satellite tiles: tier 750K, 2M, 4M, 20M

9.5.2 Technical Details

- **Maximum Zoom**: z0-21+ (can do overzoom)
  - z0-8: NASA MODIS
  - z9-12: Maxar + Landsat
  - z13-16: Maxar Vivid
  - z16+: Vexcel aerial (sub-meter in North America/Europe)

- **Download limits**: based on tile requests and MAU
- **ML Inference**: to be verified (they mention ML support but with limits)
- **Download**: not directly allowed

9.6 Copernicus Data Space Ecosystem / Sentinel Hub

The European Union's Copernicus programme offers completely free access to Sentinel satellite data.

9.6.1 Tariff plans

- **COMPLETELY FREE**: free and unlimited access to all data

9.6.2 Technical Details

- **Maximum Zoom**: z16 (Sentinel-2: 10m/px, Sentinel-1 SAR available)
- **Download limits**: no limit, free access
- **ML Inference**: fully supported (dedicated libraries available as eo-learn)
- **Download**: unlimited and free

Despite free access and full support for ML applications, the maximum resolution of z16 (10m/px) is insufficient for detecting individual trees, making this platform unsuitable for the purposes of this research.

9.7 Planet Labs

Planet Labs operates one of the largest constellations of Earth observation satellites.

9.7.1 Tariff plans

- **Pricing on request**: Research/education programs up to 3000km2 available

9.7.2 Technical Details

- **Maximum Zoom**: z20+ (from 3m up to 30cm depending on the satellite)
- **Download limits**: based on undersigned plan
- **ML Inference**: allowed with appropriate license
- **Download**: allowed

Registration requires approval with long time frames and limits are scarce for private users. It could be a good solution for academic projects, subject to verification of API availability.

9.8 Maxar SecureWatch / DigitalGlobe

Maxar Technologies, through DigitalGlobe, offers some of the highest resolution commercial satellite imagery available.

9.8.1 Tariff plans

- **Enterprise**: prices on request

9.8.2 Technical Details

- **Maximum Zoom**: z22+ (up to 30cm/px)
- **Download limits**: enterprise
- **ML Inference**: N/D
- **Download**: N/D

Very expensive platform and oriented mainly to government and enterprise customers. The high image quality does not make it accessible for individual academic projects.

9.9 Geoportal Veneto Region

The Geoportal of the Veneto Region provides access to orthophotos of the regional territory.

9.9.1 Tariff plans

- **FREE**: $0 - full access without registration, non-commercial and commercial use permitted under IODL 2.0 license

9.9.2 Technical Details

- **Maximum Zoom**: orthofoto AGEA 2021/2018 (z19-20)
- **Download limits**: only viewing for orthophotos (no download)
- **ML Inference**: N/D
- **Download**: not allowed (view only)

It offers good resolution for the Veneto territory, but the limitation to visualization alone prevents its use for machine learning applications.

9.10 Comparison and considerations

The choice of the platform depends heavily on the specific use case. For applications of automatic detection of tree cover through deep learning, the determining factors are:

- **Resolution**: you need at least z18-19 to be able to identify individual trees
- **ML Compatibility**: Many platforms explicitly prohibit use by inference
- **Possibility of download**: essential for processing images locally
- **Costs**: High-resolution solutions are often prohibitive

MapTiler was chosen for this search as a compromise between image quality (z20), permission to use for ML inference and low costs. Free platforms such as Copernicus, while fully usable for ML, do not achieve the resolution needed to identify individual trees in an urban context.

9.11 Disclaimer

All trademarks mentioned in this document belong to their respective owners. Their citation is for informational purposes only and does not imply any affiliation, approval or sponsorship by the trademark holders.

10 Thanks

After so many pages of science, it is also necessary to insert a human page: my thanks.
This is the concluding chapter of a journey that began in 2004 but, somehow, ended only in 2025: in some ways it was a real journey.

First of all I have to thank my family, my wife Linda, my mother Carlina and my "second parents" Susanna and Aldo, because without them I would have been like a shipwreck in the middle of the sea at the mercy of differential equations, spherical capacitors and Fourier transforms. They helped me keep the bar straight ahead of the earth.

I also thank Prof. Davide Quaglia, who was not only a supervisor but also a mentor, always available for an atypical student like me, who studied more at night than during the day.

And then I can not but spend a word also for my supporters, my personal "Curva Sud", friends, colleagues and companions of a thousand adventures: Alberto R., Alberto S., Christian, Francesco, Manuel and Michele, who have followed me in all the successes and setbacks that I have had in this path, thank you very much.

Well last but not least, thanks to my mascot Milo. Woof.