TEFT Summer Internship

Kystverket - Team-ML

Project Report

Martin Valderhaug Larsen
Elias Lerheim Birkeland
Simon Lervåg Breivik

KYSTVERKET

TEFT-lab Sparebanken Møre NTNU

August 12, 2023

Abstract

The main goal of this project was to do image analysis with artificial intelligence and machine learning to classify building facade colors. The motivation was a more realistic representation of Kystverkets' digital twin application. We utilized the YOLO algorithm and Google Maps API to detect buildings and classify their color from image data. This is currently a work in progress, and the report will provide an overview of our current methodology, results, issues, and potential solutions. In addition to the building project, we have trained a YOLO object detection model to identify excavators on ships. This was a much simpler problem than the building facade colors and will be used as an example of how Kystverket can use object detection in general. Two separate GitHub repositories will be provided. Building-colors contains the building color detection model. Python-image-detection contains the excavator demo and tutorials on how to use the YOLO object detection models in general. For more documentation, we refer to the readMe.md files found there.

Table of Contents

1 Introduction

This project is a result of a summer internship at Kystverket/TEFT-lab in Ålesund. Kystverket wanted to explore the potential of image recognition through a summer project. A specific case was to use machine learning to improve their digital twin. Color classification of ships was considered, but ultimately the main goal became automatic color classification of buildings from image data, to improve the realism of the digital twin. The hope was to make something tangible out of the "AI" world. This method would preferably also be transferable to other similar problems.

1.1 Building Color Classification For the Digital Twin

Some might ask why it is important to color the buildings in the simulator accurately. In fact, we also wondered the same thing. With the digital twin being used as a supplement in the training of the Norwegian Pilot Services, some got a bit annoyed when the buildings along the coast had a completely different color from the real world. The digital twin also has other potential use cases, and it aims to be as realistic a representation of the Norwegian coastline as possible. This makes it natural for the buildings to have the correct color. Anyway, we now had a project on our hands, trying to do something that even Microsoft Flight Simulator had not done before.

1.2 Detecting Objects on Ships

Having started work with object detection models, we quickly realized their potential. We therefore spoke greatly about our revelation when we had our meetup with the other Kystverket summer interns. With our friends in Arendal seeing potential in exploiting our models (and labor), they expressed interest in us building a ship excavator detector. Kystverket has detailed data on almost all ships along our coast, but they still miss some relevant information about ships. One such thing was if a ship has an excavator or not. This is due to the fact that many ships are classified in databases as "general cargo", which makes it unclear what they are actually carrying. The presence of an excavator implies certain types of bulk cargo. This is useful information for analysis of the sea transport market. We therefore saw an opportunity to train an object detection model, so that we could run through images of ships to find if they have an excavator or not. Toward the end of the project, they also expressed interest in trying to classify side ramps on ships, a task that is a bit more complicated than excavators, but much simpler than the building color classification.

1.3 Project Team

With Kystverket being a part of TEFT-lab, the project ran over a period of 6 weeks, and gave us access to some guidance through NTNU. Combining this with feedback from our employer, we had a solid team for this task. We would like to thank our employers and supervisors for their help, and especially Odd Sveinung Hareide for being very important to the project, and for not wasting any time in answering our questions.

Name	Field of study	Role
Martin Valderhaug Larsen	Informatics with Artificial Intelligence	Intern
Elias Lerheim Birkeland	Industrial Economics and Technology Management	Intern
Simon Lervåg Breivik	Cybernetics and Robotics	Intern
Odd Sveinung Hareide	Kystverket	Employer
Haakon Akse Nordkvist	Kystverket	Employer
Hans Georg Schaatun	NTNU	Supervisor
Rituka Jaiswal	NTNU	Supervisor

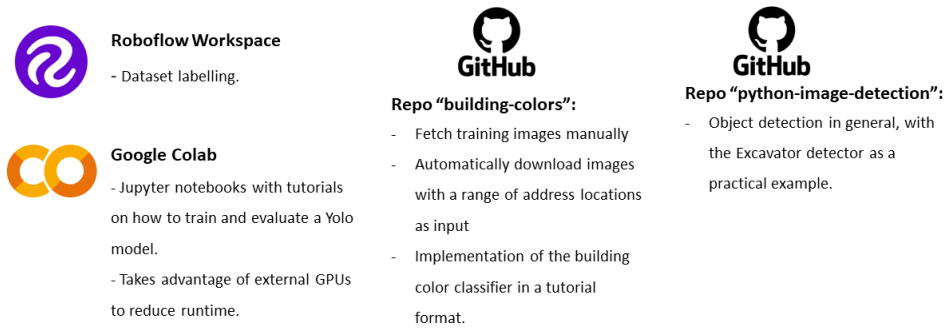
Table 1: The people working on the project

1.4 Project Overview

The code we delivered at the end of the project can be found on GitHub. In addition, we used Roboflow for dataset labeling, and Google Colab for faster training and inference with the models. Below are links to the different parts of the project. Further details on how to use them are contained inside the GitHub repositories readme-files.

- building-colors - repository with the building color project
- python-image-detection - repository with the excavator project
- Roboflow workspace - workspace with all our current labeled datasets
- Object Detection 101 and Object Detection Buildings - tutorials on how to train and export a custom object detection model in Google Colab.

The overall structure of our project is presented in Figure 1



.

Figure 1: Overview of project

2 Methods

2.1 Choosing an Approach

Starting out, building a detector from scratch was considered. With there being theories surrounding this, we wondered if it would be the way to go. However, we quickly realized that our theoretical knowledge did not meet the requirements. Therefore, we had to consider pre-built object detection frameworks. In retrospect, we found the object detection frameworks great, and we would recommend this regardless of the groups' prior knowledge.

Mask R-CNN It would be an overstatement to say that we considered many different frameworks. We started off looking at Mask R-CNN, however, we ran into some problems when implementing, while also being a bit skeptical of the amount of data needed to make a custom model. We ultimately moved away from it.

Detectron 2 We briefly looked at detectron 2 from Meta. But after struggling with training on a custom dataset, this did not lead anywhere.

YOLO The next framework considered was YOLO, "You only look once". Even though the model says so, we had to look more than once to implement it. After looking a handful of times, we managed to implement it while being more optimistic about the amount of data needed. And with our project having a limited amount of time, we chose to go down this path.

2.2 You Only Look Once

Object detection models use machine learning techniques, particularly deep learning, to perform the task of identifying and localizing objects within images or videos. The models detect objects, draw bounding boxes around them, and assign probabilities. YOLO is an advanced object detection model that has gained prominence in the field of computer vision. YOLOv8's benefits include remarkable accuracy in identifying multiple objects within a single frame, rapid processing that allows for seamless integration into real-time systems, and the ability to operate effectively even on resource-constrained devices. As a result, YOLOv8 has emerged as a powerful tool in the realm of object detection, offering enhanced performance and practicality for a wide range of visual recognition tasks. Here you can read more about Yolov8. We use YOLO from the Ultralytics python module. YOLO was a suitable choice for us because it can detect buildings and classify their color at the same time. It is also simple to learn and use, which was good because we only had a few weeks.

2.3 Creating a Custom YOLO Model

The process of creating a custom object detection model roughly consists of three steps - creating a dataset, training the model, and implementing the model.

Labeling Dataset For labeling our datasets we have used Roboflow. Roboflow provides a user-friendly way of annotating images and allows teams to annotate simultaneously. In order to get started, we suggest having a look at this blog post. Our already labeled datasets are available in this workspace. Some tips for annotating can be found here.

Training To train a model we take advantage of the GPUs provided by Google in Colab. Have a look at our tutorial Object Detection 101 to train your own custom object detection model.

Using a Model For a practical introduction on how to use the model you have trained, have a look at the readme.md-files in either of our GitHub repositories.

3 Building Color Detection

3.1 Data Source

Satellite Imagery With satellite imagery being the most widespread type of geographic photography, there is a huge amount of imagery that can be extracted easily. Thus, we naturally started by considering this as our data source through Google Earth. However, to everyone's surprise, if you take a picture of the building from above, you end up with the color of the roof, which was not very useful.

Google Streetview Having realized we needed to move away from the skies, we considered using Google Streetview. With Google Streetview covering most of the buildings along roads, we had data for a respectable amount of buildings. With this solution, we could go to a specific location and point the camera in that direction. Here is a relevant tutorial and paper. However, this solution also had some flaws. The main one is that it only gives one position to take an image from. This often leads to a blocked view of the building, forcing us to make a model for filtering everything but the building from the image. This filter problem could quickly become more complex than the image classification itself. As seen in Figure 2, we would have a hard time trying to classify the building color. Again, we had to look further for possible data sources.

Figure 2: An example of a shortcoming when using street view

45-degree Imagery Having tried images from the skies and from the ground, we found an in-between solution - something called 45-degree imagery. As the name suggests, this is a type of satellite image where we take the images from a 45-degree angle. From such a view, we should be able to get a view of the walls of the buildings. The buildings can also be viewed from four different angles. In the case of an obstructed view, we could thus rely on the other angles. With all this considered, we chose to use 45-degree imagery as our data source.

Google Earth 3D Lastly, it is worth noting that Google Earth 3D was considered as a possible data source. However, we found that the buildings often take weird shapes with not-so-good colors. This has to do with the buildings being generated from the 45-degree imagery they provide. With 45-degree imagery being the raw data of the 3D models, we found it more appropriate to work with the 45-degree imagery itself. In addition, there was an easy-to-use API available for the 45-degree

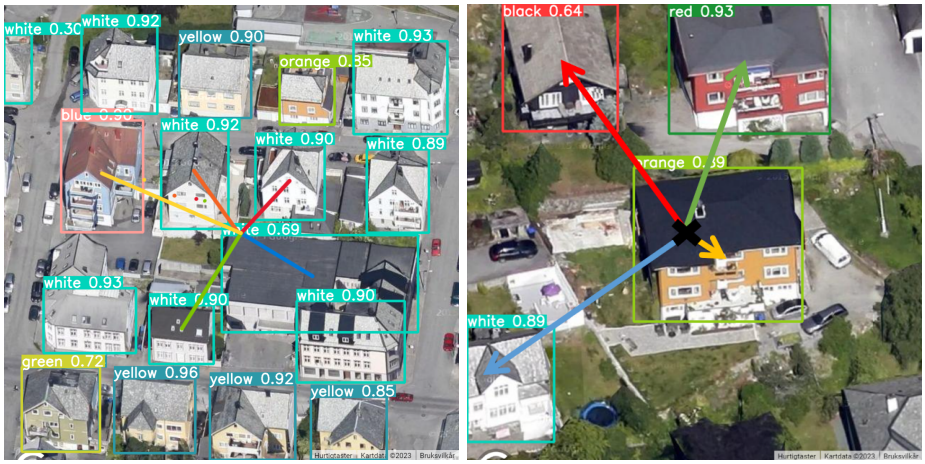
images, whereas with Google Earth 3D we found no easy way to get images. However, if one does find a way to get these images, it could potentially be a powerful data source because of the ability to rotate 360 degrees around the maps.

1881.no Skråfoto 1881.no has 45-degree imagery of more locations than Google and of better quality. Their data comes from Norkart and is not free to use, which is why we did not use it. In tests we did with our model it worked with these images as well, and the data source should be considered if needed. Preferably after improving the model.

3.2 45-degree Approach

We collected more and more images for training the model. In the end, we had annotated 676 images and a total of 1826 buildings. We used this to train our model.

An example output from the model is shown in Figure 3. This image also shows a key concept when choosing what color a building is: we take the most centered bounding box. In the following example, it would be the white building with the red line. The reason why we choose the centermost building is the fact that we pass in an address location (latitude, longitude), and assume that the building is located at the center of the image.



(a) ex.1 (b) ex.2

Figure 3: Two example images. Also plotted are helping lines in order to see which building is most in the center.

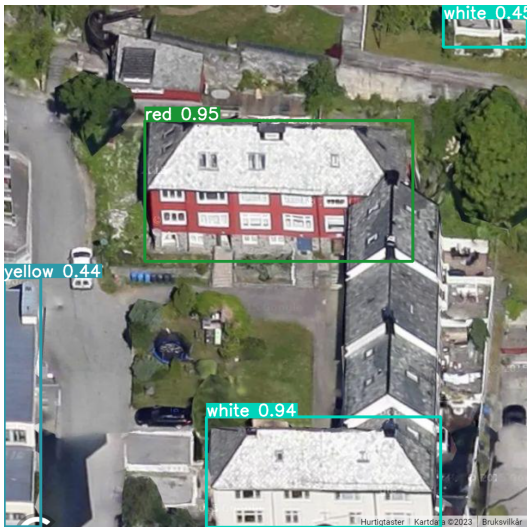
What Our Code Does:

1. Obtains images of the building from the four cardinal directions
2. Predict all buildings in each image
3. Choose the centermost box in each image
4. Add up the probabilities of each color
5. Pick the most likely color

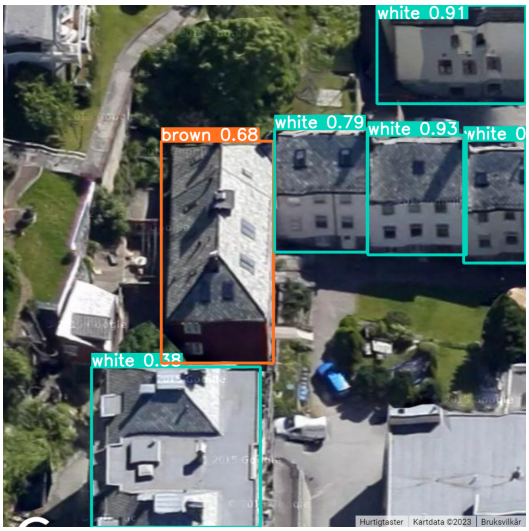
In order to tell what color a building is, we follow the recipe above. An example of this is shown in Figure 4

(a) result: ('bad', 0), box too near edge

(b) result: ('red', 0.663)



(c) result: ('red', 0.948)



(d) result: ('brown', 0.685), shadow

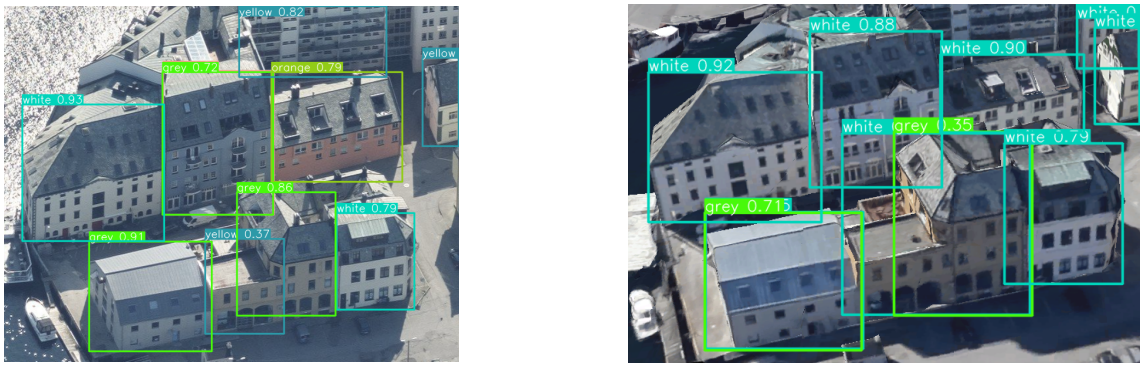
Figure 4: Here you can see an example run of our model. The end result would be red: since the sum of the red ones is higher than the brown. Keep in mind that the model only takes into account the centermost buildings.

3.3 Obstacles

Our approach faces several obstacles, although it was initially designed as a proof of concept. Consequently, we have not extensively delved into the legal aspects. However, it is worth noting that Google has expressed reservations against utilizing their services for model creation. Furthermore, if Google were to be pursued for larger-scale implementation, associated costs would arise. Our request volume was relatively modest, and we got a long way with three free API keys which granted us roughly 75 000 requests per month.

The image quality occasionally proves suboptimal, primarily due to their automated capture. Additionally, our coverage is limited to specific parts of Norway due to the constraints of angled

imagery. Thus, achieving comprehensive nationwide building prediction is currently unattainable. Nevertheless, Google covers a significant number of cities within Norway, rendering it suitable for our present purposes. Our model is adaptable to diverse image sources, including 1881.no's "skråfoto" service. Some test runs with our model on other image sources can be seen in Figure 5. These examples are not great, but shows that it is possible to build on the existing model when using other image sources.



(a) 1881

(b) Google Earth

Figure 5: Model output for other sources

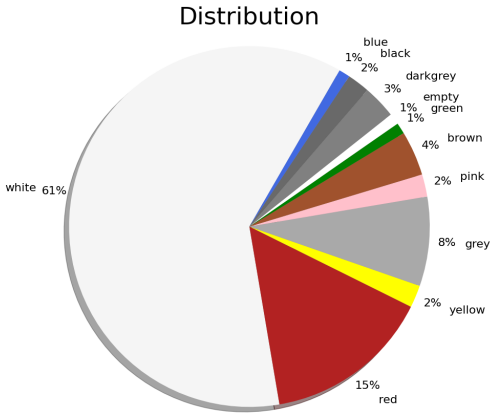
Regarding processing speed, our model can analyze 5000 images within approximately 8 minutes, utilizing the work PC provided by Kystverket. But a bigger issue is that we currently have to download all the images, and then reupload them to Google Drive before running the model. We advocate for exploring ways to enhance this process. Employing a network cable might alleviate bottlenecks stemming from internet connectivity. Alternatively, identifying alternatives to Google Drive and Colab could prove beneficial. Moreover, integrating Python directly into the image collection code offers a clever solution, but we did not have time to explore this further. We wanted to test this out by either making a backend/frontend using Flask or directly with PyScript. In addition, section 5.2 shows how you can use a YOLO model directly in html/javascript.

3.4 Results

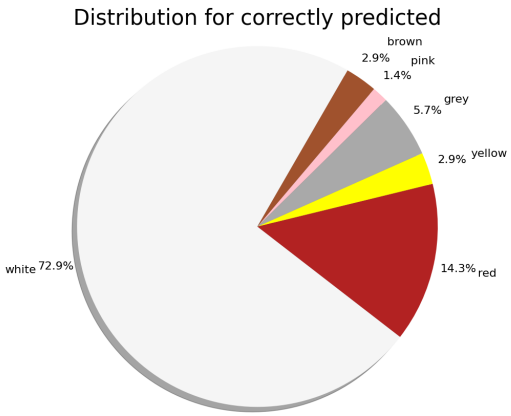
We tested the model on a lot of small test sets (5-35 images). In terms of a larger scale, we used postal codes 6006, Ålesund, and 6005, Ålesund. They both led to a JSON file containing the result for all the addresses contained in the postcodes: for 6005 (Aspøya), 430 addresses, and for 6006 (Hessa), 1366 addresses. In Figure 6, all the colors for 6006 are displayed on a map. In order to get a sense of the final performance we picked out 100 random addresses from Hessa and created plots showing some of the results.

Figure 6: A map with the results

Firstly the distributions are shown in Figure 7. Here we can see that the white buildings are very overrepresented. We could have handpicked the examples, in order to get a fair dataset, but we preferred that the test would be tested in the real world, where there are many white buildings. This may lead to false confidence when we present our final result, which was **70 out of 100**. Some of the examples which the model managed to predict correctly, may have been luck since many of the buildings are white. Therefore even if the model targets the wrong building, if the whole neighborhood is the same color, then it still gets the correct prediction.



(a) The distribution for all of the 100 addresses in the random dataset from Hessa



(b) The distribution for the correctly classified addresses. In total 70 addresses

Figure 7: Some pie charts showing the distributions

Another interesting fact is where the model made wrong predictions. As can be seen in Figure 8, there are not too many patterns in the wrong predictions.

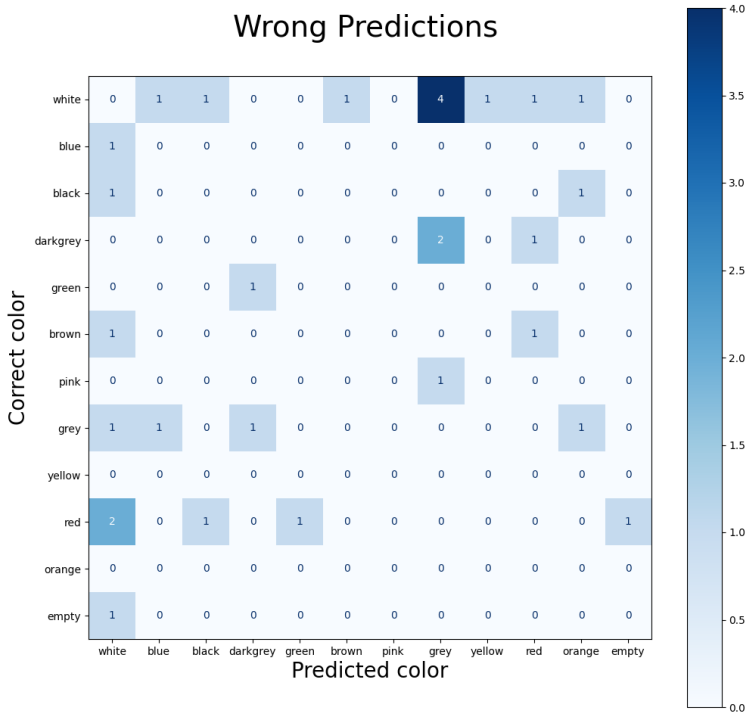


Figure 8: Showing the wrong predictions. The horizontal axis shows the predicted color, and the vertical shows the real color. The errors are quite spread out.

Having seen through all the errors, they often have images with shadows, bad focus, or many buildings in a tight area. Often it is influenced by a neighboring building. Here is a small portion of the predictions from the test. Figure 9, displays correct predictions, while Figure 10 displays wrong predictions.

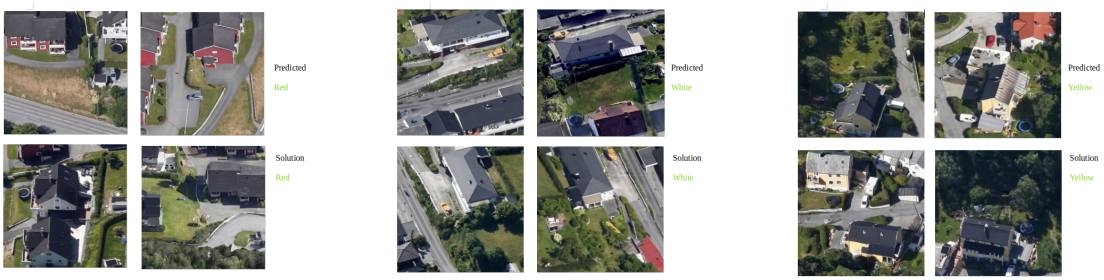


Figure 9: Some examples of correct predictions

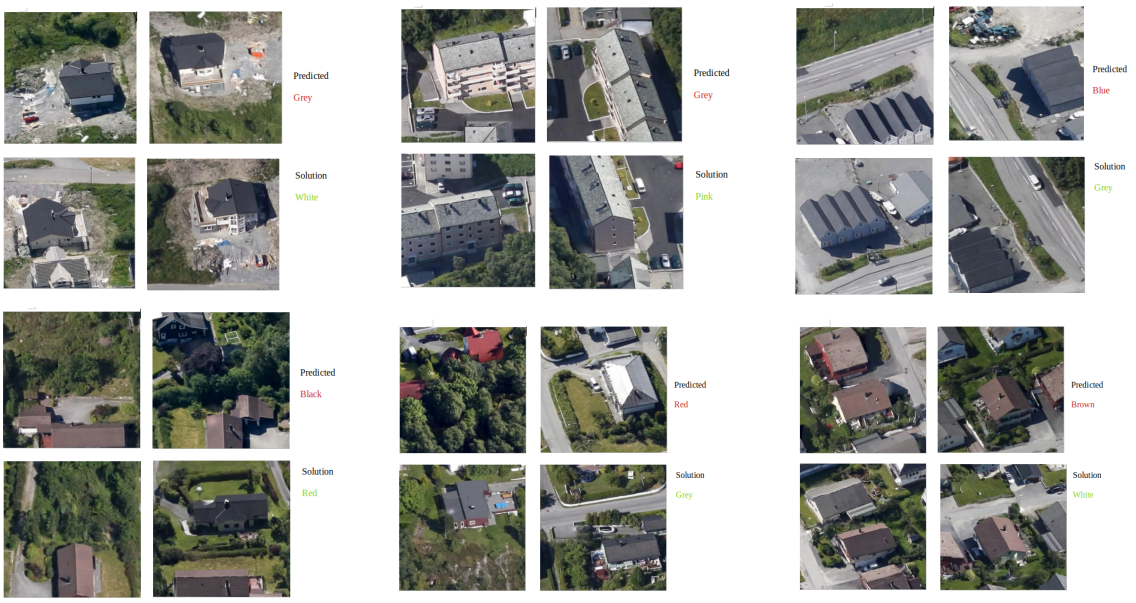


Figure 10: Some examples of wrong predictions

Lastly, it is worth mentioning the things included in the JSON file. A portion of the file content is displayed in Figure 11.

Address	Postal Code	Area	Lat	Long	Predicted	North	East	West	South
Solsidevegen82B	6006	AALESUND	62.458	6.129705180045769	grey	['grey', 0.5]	['grey', 0.729]	['grey', 0.799]	['grey', 0.56]
Solsidevegen84A	6006	AALESUND	62.4579	6.129414952716177	grey	['grey', 0.589]	['grey', 0.764]	['grey', 0.68]	['grey', 0.822]
Solsidevegen84B	6006	AALESUND	62.458	6.129354235283569	grey	['grey', 0.409]	['grey', 0.757]	['darkgrey', 0.357]	['grey', 0.788]
Solsidevegen86A	6006	AALESUND	62.4579	6.129115208064444	grey	['darkgrey', 0.394]	['grey', 0.715]	['grey', 0.6]	['grey', 0.714]
Solsidevegen86B	6006	AALESUND	62.458	6.129150521865829	grey	['grey', 0.419]	['grey', 0.765]	['grey', 0.756]	['grey', 0.654]
Solsidevegen88A	6006	AALESUND	62.4579	6.12885524939219	grey	['grey', 0.598]	['grey', 0.833]	['darkgrey', 0.81]	['grey', 0.547]
Solsidevegen88B	6006	AALESUND	62.458	6.128790805553899	grey	['grey', 0.424]	['blue', 0.772]	['darkgrey', 0.755]	['grey', 0.438]
Solsidevegen90A	6006	AALESUND	62.4579	6.128554700740935	grey	['empty', 0]	['blue', 0.706]	['grey', 0.537]	['grey', 0.36]
Solsidevegen90B	6006	AALESUND	62.4579	6.128584674586953	grey	['grey', 0.679]	['grey', 0.576]	['darkgrey', 0.71]	['grey', 0.53]
Stavnesvegen10	6006	AALESUND	62.4578	6.118434462867057	white	['empty', 0]	['white', 0.813]	['black', 0.404]	['green', 0.351]
Stavnesvegen101	6006	AALESUND	62.4546	6.103964241371374	brown	['brown', 0.914]	['black', 0.811]	['white', 0.725]	['red', 0.623]
Stavnesvegen103	6006	AALESUND	62.4546	6.103419643080230 5	red	['white', 0.596]	['blue', 0.75]	['grey', 0.677]	['red', 0.931]

Figure 11: This is what is contained inside the final JSON file

We would now like to present some more theoretical stuff about our models performance. After training the model you obtain a bunch of graphs and metrics on your models performance. The most important one is the mAP50 - *mean average precision* with an intersection over union threshold of 50%. This is quite a complicated metric but is useful because it takes into account both how good the boxes your model draws are, as well as the classification. Make sure you have a good grasp on the terms precision, recall, confidence threshold, and intersection over union. In Figure 12 you can see that the overall performance is not optimal at the moment, with some colors being especially bad. We did however notice a significant increase in mAP50 by increasing the dataset from around 400 to the current 674 images. This increased the mAP50 from around 0.4 to the current 0.648.

```
Validating runs/detect/train/weights/best.pt...
Ultralytics YOLOv8.0.20 🚀 Python-3.10.12 torch-2.0.1+cu118 CUDA:0 (Tesla T4, 15102MiB)
Model summary (fused): 268 layers, 43615089 parameters, 0 gradients, 164.9 GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50  mAP50-95): 100% 5/5 [00:09<00:00,  1.80s/it]
                   all        160        441      0.654      0.582      0.648      0.464
                 black        160         16      0.583      0.525      0.413      0.283
                  blue        160         24      0.635      0.363      0.578       0.41
                 brown        160         30      0.579      0.433      0.469      0.364
              darkgrey        160         10      0.418        0.5      0.548      0.443
                 green        160          9      0.775          1      0.962      0.649
                  grey        160         47      0.506      0.298      0.344      0.232
                orange        160         14      0.584      0.786      0.776      0.637
                  pink        160         22      0.751      0.636      0.729      0.481
                   red        160         48      0.887      0.656      0.872      0.632
                 white        160        178      0.806      0.559       0.76      0.499
                yellow        160         43      0.676      0.651      0.672      0.471
Speed: 0.6ms pre-process, 18.0ms inference, 0.0ms loss, 3.1ms post-process per image
Results saved to runs/detect/train
```

Figure 12: Summary of model performance

3.5 Future Work

An obvious source of error is our simple definition of the color spectrum. With more time, we would have started by splitting blue into a light and dark category. And we would go through the whole dataset and made sure the colors we have defined are consistent within each category. Also the colors brown, grey, darkgrey and black need improved training data quality. One could start with a better and larger dataset to see how far it would take the mAP.

A different path, which we have discussed with our supervisor, would be to use a building detector and then a classical approach for determining the color. For example, working with the hsv (hue, saturation, and value) components using a K-Means clustering to find the main colors. The hue represents the fundemental quality of the color as humans experience it, and could in theory be used to precisely determine the color of each building. We have trained a general building detector, whose output you can see in Figure 13. The building detector has an mAP50 of around 0.9, so it would be interesting to try this out. Because of the lack of time available, we did not look more into this possibility. We could also have looked into supplementing the decisions using Streetview. This however would force us to train a new model using streetview images, which we did not have the time for.

Figure 13: Buildings. Ideally one could crop out the bounding boxes in order to use them for the remaining task of deciding the color.

4 Object Detection For Ships

As mentioned in the introduction, object detection for ships was a suggestion from Kystverket in Arendal, more specifically Haakon Akse Nordkvist. If any development on this topic is done, we think he would appreciate being contacted. The excavator detection and much more can be found in this Github repository.

Background In short, Kystverket want as much data as possible from ships. Therefore, automated models for detecting objects on ships could improve their datasets.

Datasets Finding ships that contain an excavator is not the easiest. Thankfully, the team in Arendal has a handful of datasets on different ship characteristics. These sets contain the ship's IMO number, a unique identifier for each ship. By using the IMO number, you can identify and research each ship.

Image sources When we have the relevant IMO numbers, the next step is to find images of the ships. With the "Klima of Miljø" group in Ålesund having done research into different ships, they recommended using Kystdatahuset to look up ship information. With Kystdatahuset using images from ship-info.com, that became our source for obtaining images. This gave us one image for each ship that could be used for image classification. Looking back at it, we could have looked into other sources that maybe contained a broader selection of images for each ship.

4.1 Excavator

Detecting excavators was the original idea from Arendal. They handed us a dataset containing about 140 ships with excavators. We combined these images with about 200 images of ships without excavators, ending up with a dataset with roughly 350 images. The dataset can be found here. This made a solid excavator detector.

In our Python image detection repository, there is an excavator demo program "ship_demo.ipynb", this walks you through the process of reading the dataset, loading the images, and saving the predictions.

4.2 Side Ramps

The next request from Arendal was to look at ships with side ramps. In short, this is a ship with a ramp that can be used for unloading the ship unassisted. The data on which ships have side ramps is not sufficient, and improving them could therefore be a further extension of the project. Having looked at different side ramps, it is clear that this is more complicated than the excavators. The side ramp are often quite hidden and take different forms.

To attack this problem, it can be an idea to classify different types of side ramps. With some side ramps looking like basic ramps, and some ones looking like boxes, it can help the model find distinct characteristics with each type of side ramp. As seen in Figure 14, all the ships have side ramps, but they have different appearances. Some research should be done into how different side

ramps are shaped. With this information, different classes can be made, giving the model more accuracy. Unfortunately, we did not have time to begin with this. However, the team in Arendal

Figure 14: Some predicted images

has a dataset on ships with side ramps, and they can be contacted if the reader wants to continue this quest. Remember that we left a guide on how to build a model in section 2.3.

4.3 Results

Excavator Firstly, our dataset and some results can be found on the public Roboflow page here. This shows an accuracy for the model way above 95% . We also see in our own research that the smaller models perform around 95%, leaving some margin of error. Looking at some predictions in Figure 15, we could say the results look reasonable.

Figure 15: Some predicted images

Looking at some of the misclassifications in Figure 16, on the left, the excavator is not classified due to it being behind the ship crane. On the right, the picture is taken while the ship is being loaded by an excavator. The model will therefore not understand that the excavator is not aboard the ship. Overall we should be happy with the model.

Figure 16: Left image: the excavator is not found. Right image: the excavator is not on the ship

Side ramps With us not having done much on this topic, we naturally don't have any results.

4.4 Future Work

Alternative image sources For the future of object detection on ships, more or other image sources should be considered. With us having only used one photo for each ship, the consequence

of a misclassification is big. If we as an example had 5 images, we could confidently assume the ship had an excavator if it is found for all 5 images. Further, we could also flag different ships where the images would not agree if the ships had an excavator or not. With this solution, these ships could be checked manually to ensure the best data quality possible.

Big specialised model For the future, making a big model for detecting all the different characteristics on a ship should be considered. E.g. a model which can detect excavators, side ramps, containers, cranes or any other objects of interest. Running the model on all ships of interest would give even more details about each ship. Such a model could be changed and expanded as the needs/technology of ships is developed. In addition, with YOLO being fast enough for real-time use, the model could be implemented on Kystverkets CCTV cameras if the need for this ever arises.

5 Future Work in General

5.1 Making a Custom Model

We believe that the reader should be able to make an object detection model. With our explanation in section 2.3, and maybe looking at one of our custom data sets in Roboflow, it becomes more of a step-by-step process that should be easier to follow.

5.2 Potential Issues With Code

Yolo is rapidly evolving, so in case our code stops working, there could be many potential reasons behind this. Some things to consider: Read our readMe.md files. For the buildings project, and for the excavator project. Search for potential issues using your error message inside of yolov8-repository-issues.

Excavator webpage During the project, a webpage for excavator detection was built. This was mainly to show the concept of converting our model into another format (.onnx), but was published mostly for fun. As clearly noted in the code of that model, the program is only a basic demo, and some bugs might appear. However, it shows how the .onnx format can be used to make webpage models.