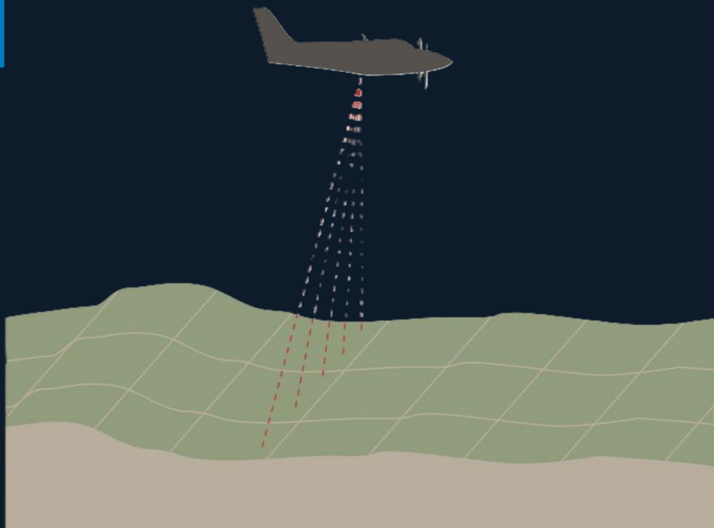
Magnus Sellebjerg, Christian Lorentsen, Jacob Møller-Jensen, Benjamin Siddique, Thomas Hansen

Applied Machine Learning

Population Estimation



Using CNNs, Transformers & Clustering to determine population counts on small-scale orthophotos

Background & Data

What Are Orthophotos?

Aerial photographs corrected for perspective distortion

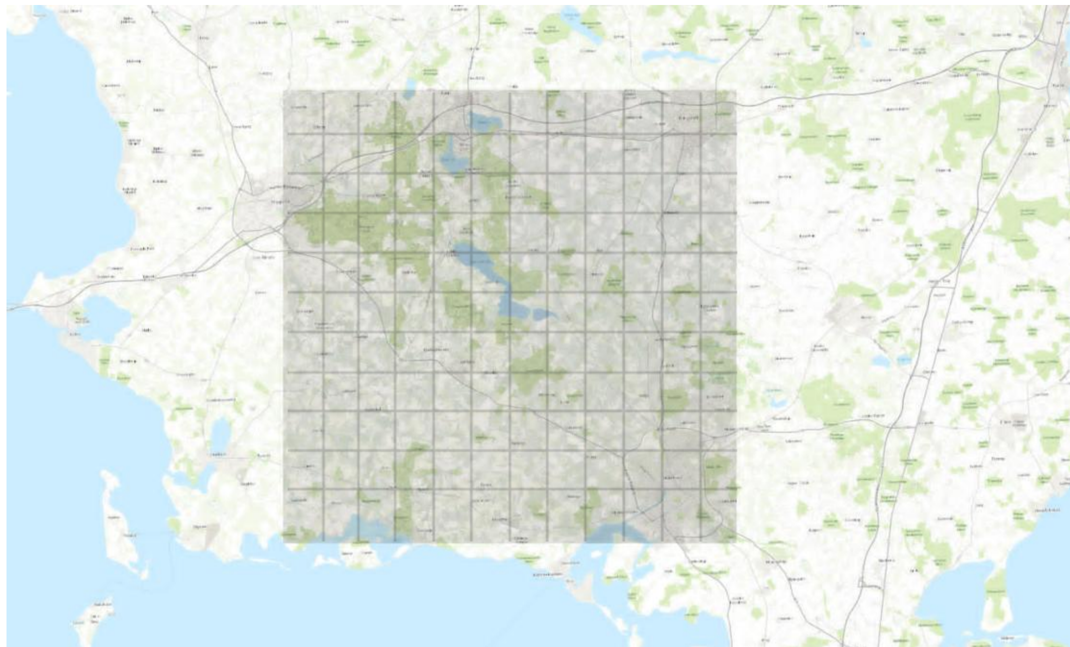
Projected onto maps with geospatial logging

Squarenet (Kvadratnet) from DST

Standardized spatial grid by Statistics Denmark (DST).

Linked to official population counts

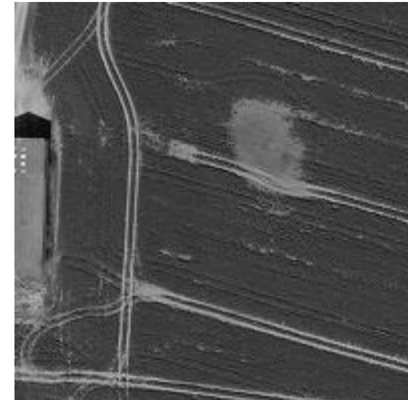
Available in different sizes (100m, 250m, 1km, 10km, 100km), 100 m was chosen.



Selected region for analysis, grid is not to scale

What the Data Looks Like

IMAGE RESOLUTION	Image SIZE	DATA REGION	DATA SIZE
1 px = 0.5 m	[200×200] px	[30×30] km	88 869 jpgs

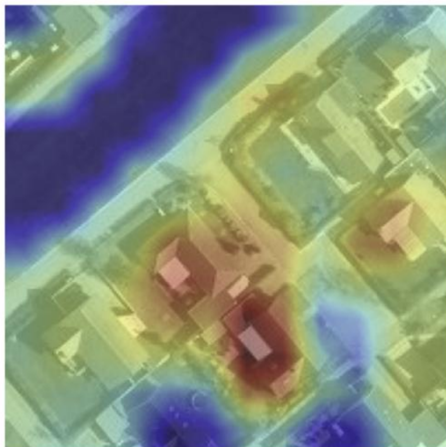


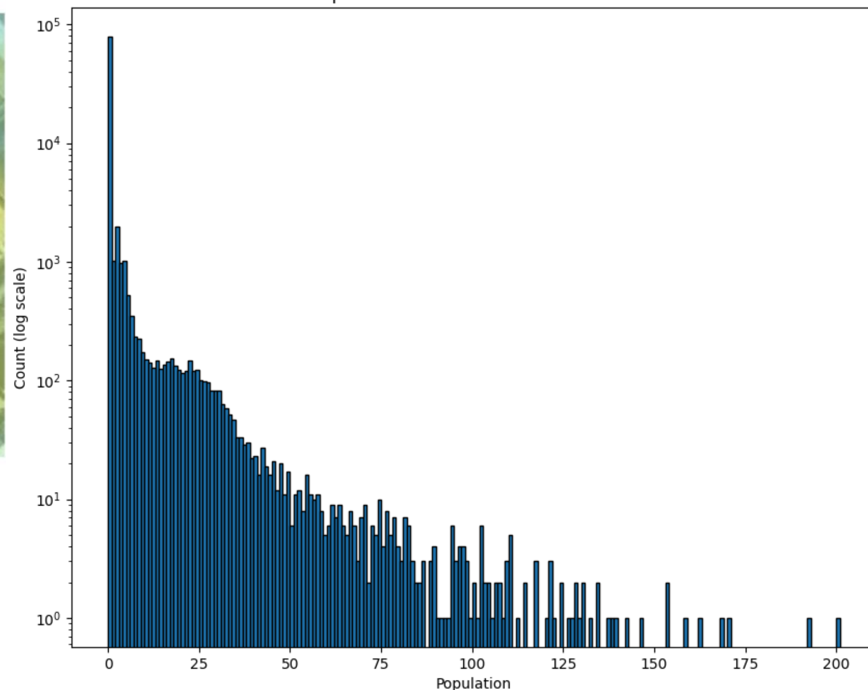
Methods: Preliminary testing (Quick and dirty CNN)

pred=25 p_present=0.99 | truth=28

input

Grad-CAM (count)

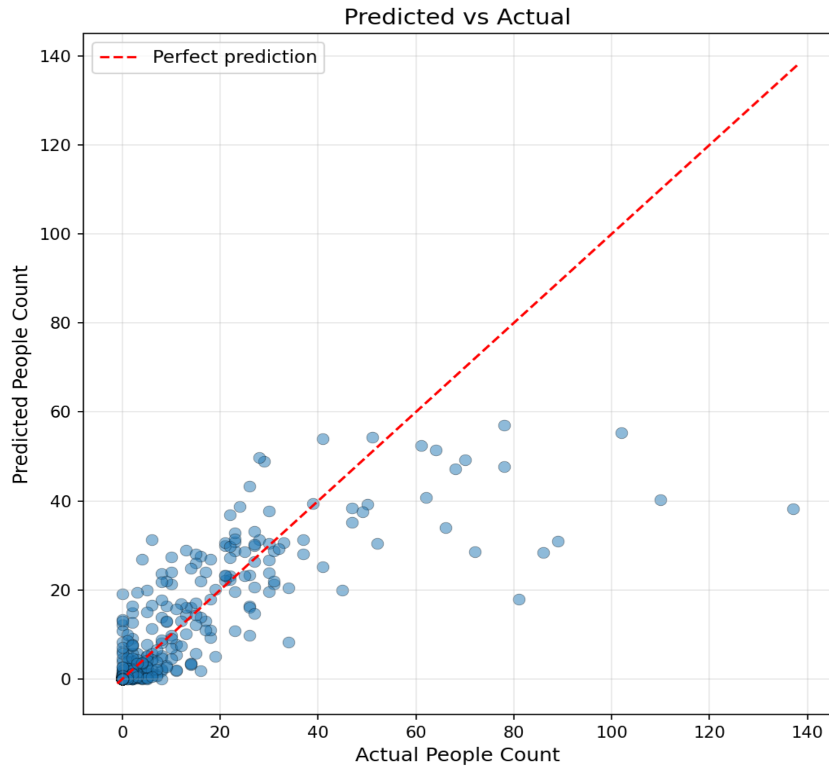


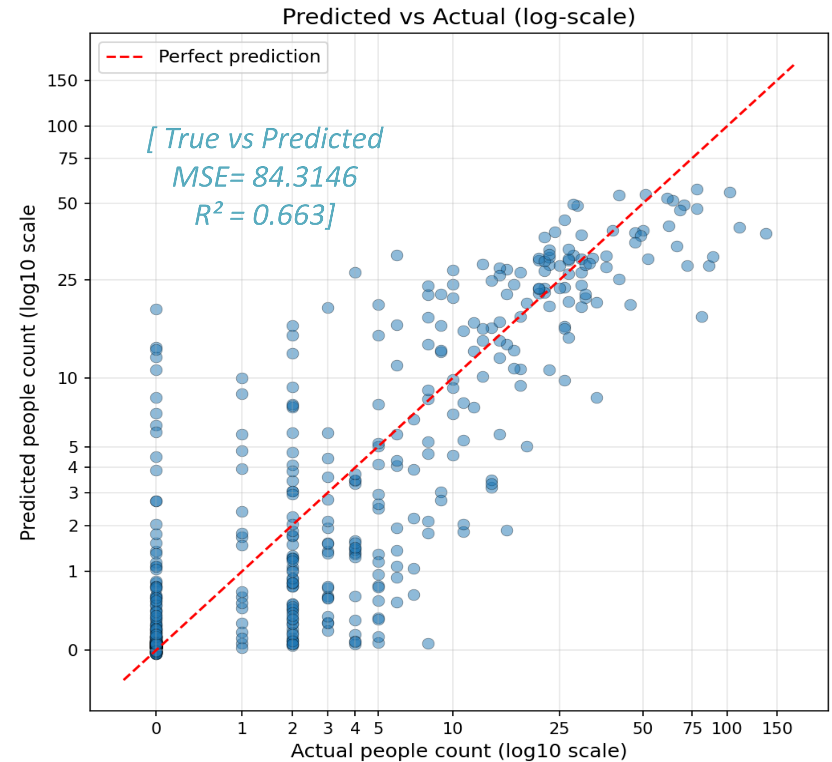


88% of images are uninhabited

Dataset heavily skewed, balanced to avoid bias
Simple approach captures important features

Methods: Problems Identified





Failure Cases: GradCAM Analysis

GradCAM highlights which image regions drove the model's incorrect predictions.

“Empty houses” and smaller objects are cause for confusion

Objects in grid shapes mimic urban zones

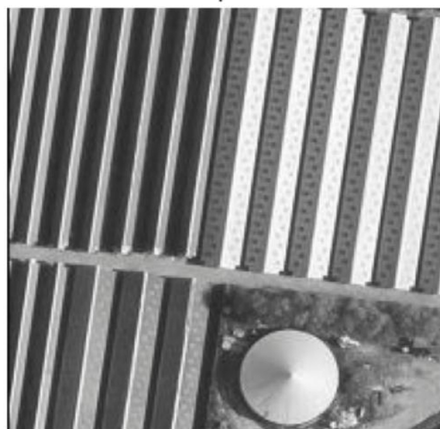
pred=15 p_present=0.84 | truth=0

input

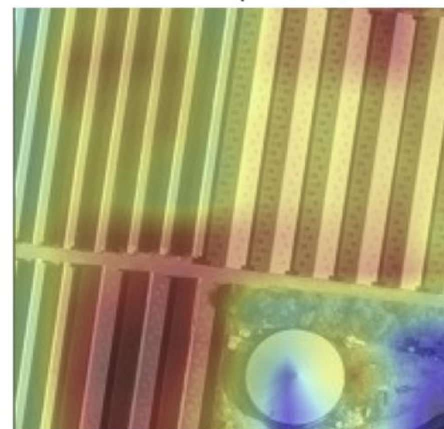
Grad-CAM (presence)

pred=45 p_present=1.00 | truth=0

input



Grad-CAM (presence)



Different strategies

Clustering

Group images by visual similarity (urban, agricultural, nature) to compare with expectations.

Classification → Regression

First classify each cell as populated vs. empty; apply regression only to populated cells, addressing class imbalance.

300×300m Squares for more context

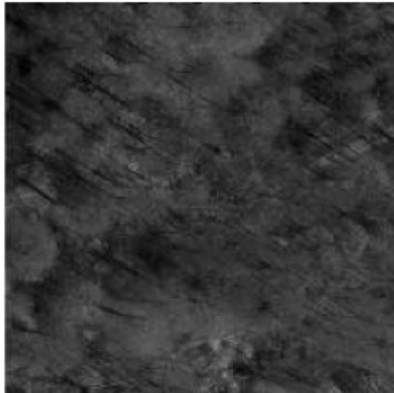
Expand the receptive field to capture surrounding neighbourhood context for improved spatial awareness.

Transformer Models

Attention-based architectures might capture long-range spatial dependencies across the full image.

Clustering Effects

005859_100m_61397_6775_0



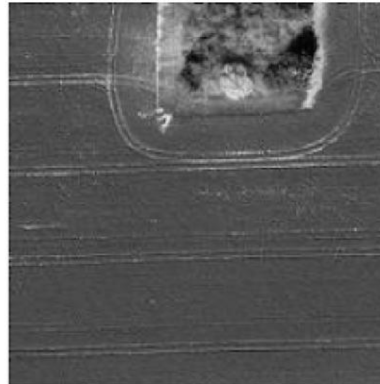
Examples of 3 generated clusters

050175_100m_61438_6669_0

058755_100m_61376_6698_21

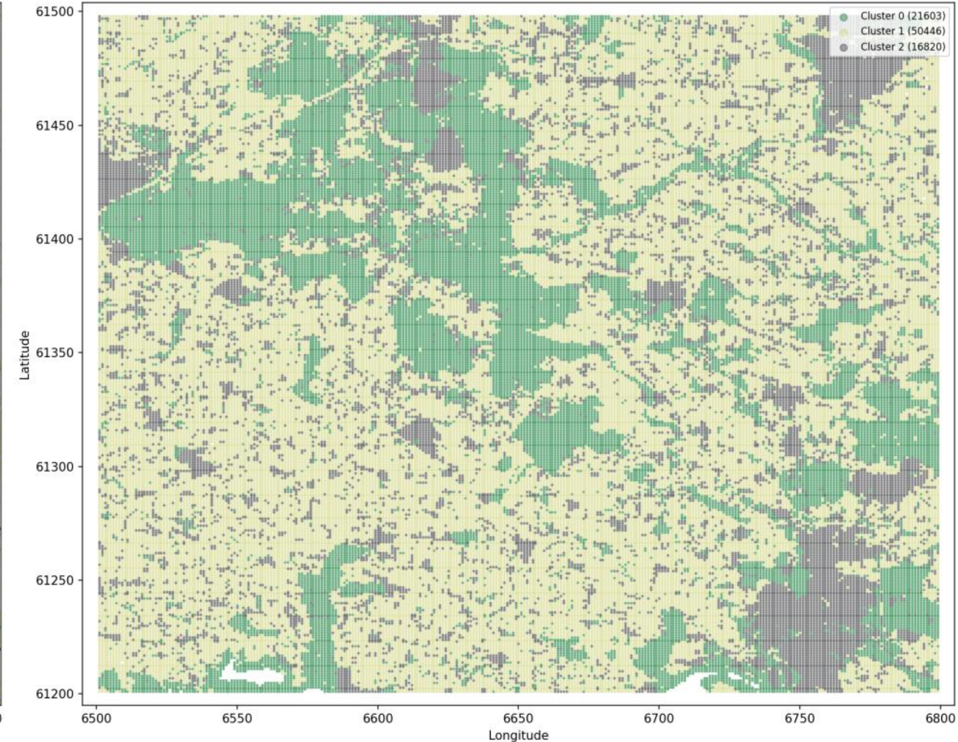
017211_100m_61203_6559_0

025021_100m_61316_6585_0



030067_100m_61296_6602_6

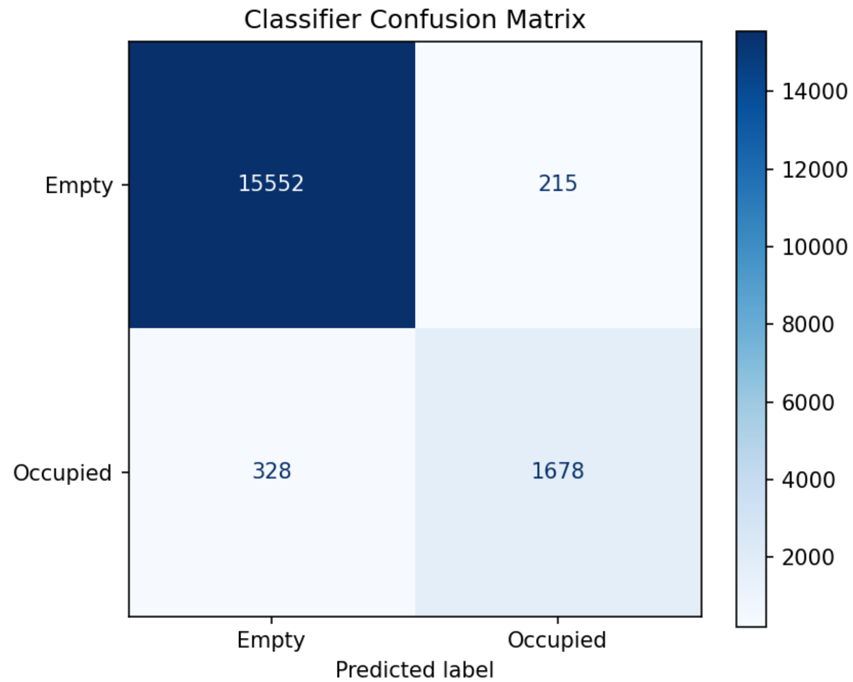
Clustering Effects



Clustering shows accurate separation into dense nature, rural fields and urban housing

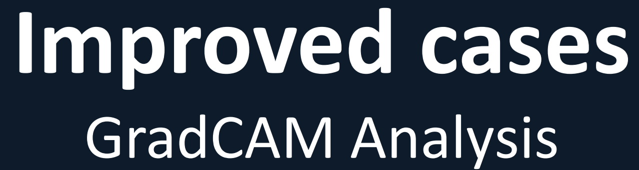
Populated vs Unpopulated

Results: Classification Full Dataset (100 x 100)



Classification Performance

MCC = 0.844
Accuracy = 0.973


Improved cases
GradCAM Analysis

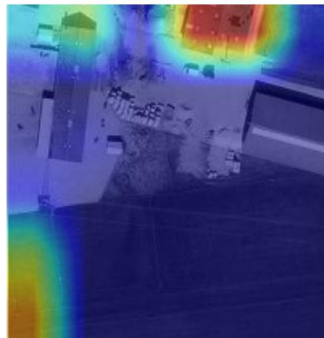
Gradcam: Classification successes

True Negative

Pred: Not Populated (p=0.12)

Input

Grad-CAM

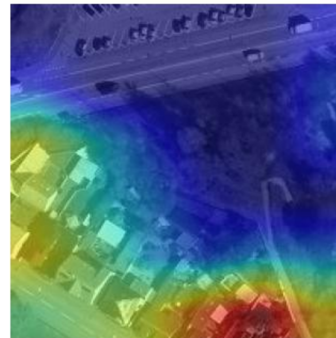


True Positive

Pred: Populated (p=0.97)

Input

Grad-CAM



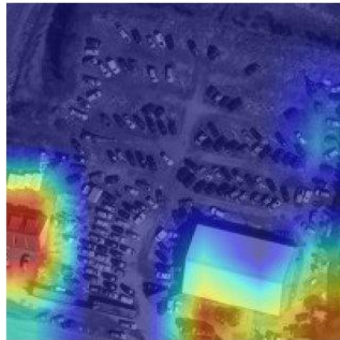
Gradcam: Classification failures

False Negative

Pred: Not Populated (p=0.20)

Input

Grad-CAM

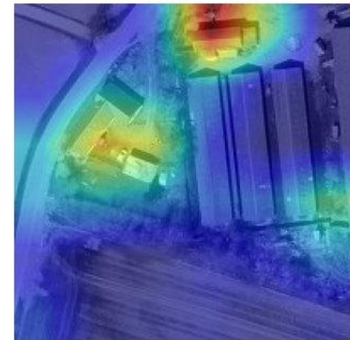


False Positive

Pred: Populated (p=0.94)

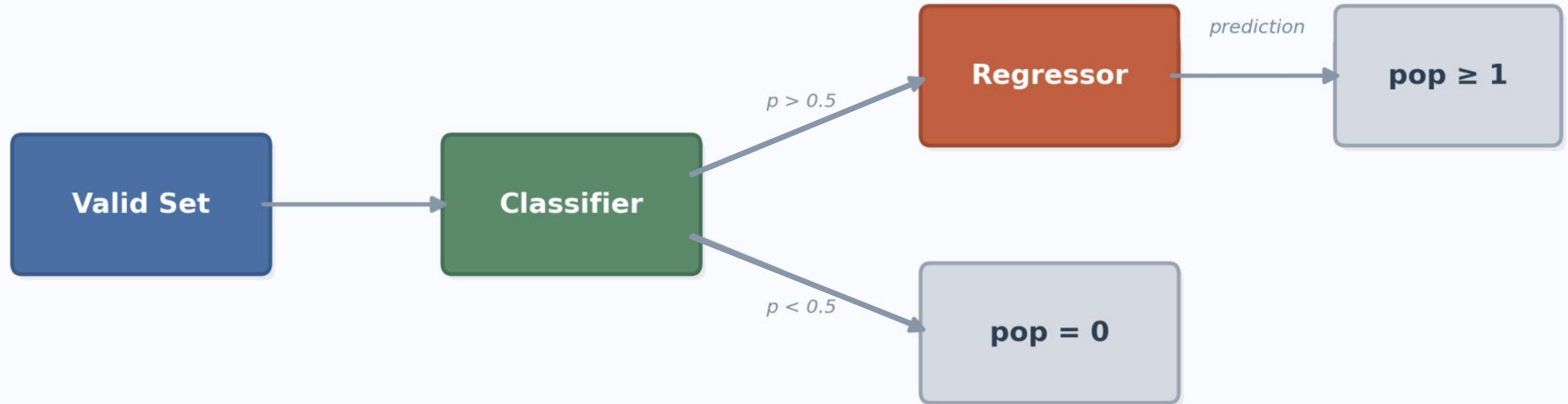
Input

Grad-CAM



Classifier + Regressor combined

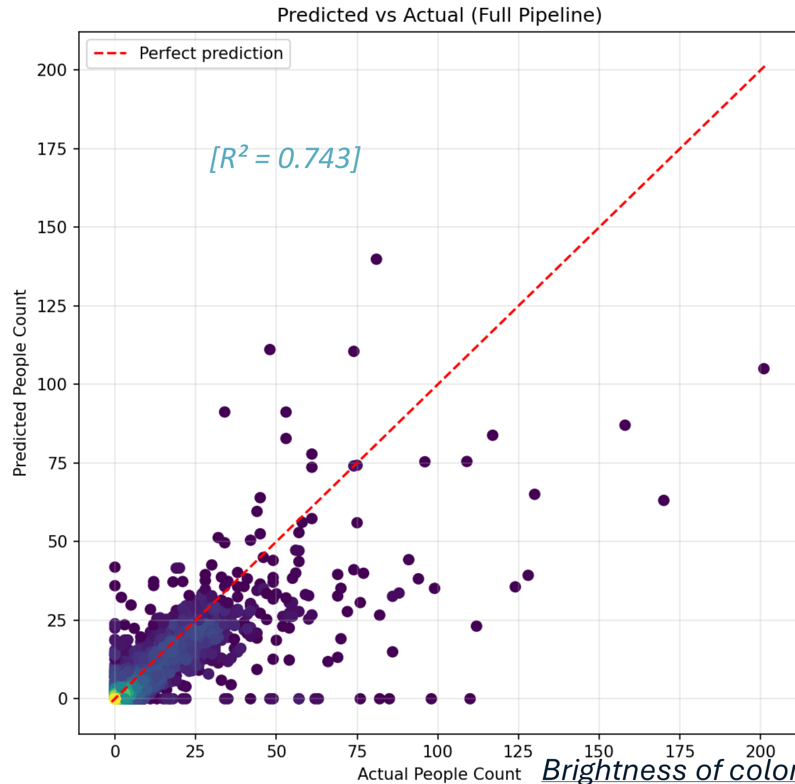
Full pipeline: Classification + Regressor combined

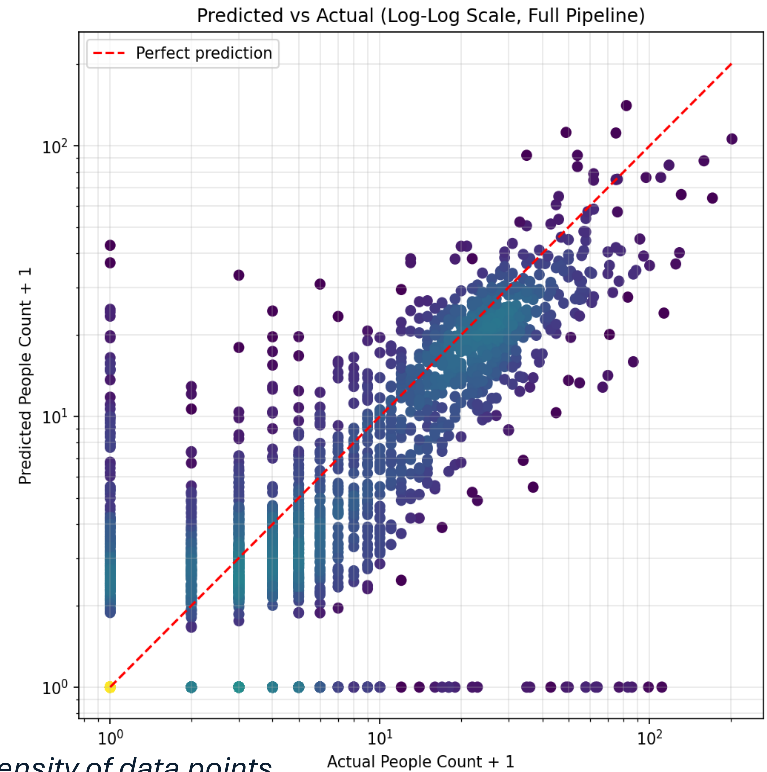


Regressor Models

Regressor Model	MSE	MAE	R^2
Model 1	11.9658	0.59	0.743
Model 2	12.6876	0.60	0.727

Classifier + Regressor Model predictions





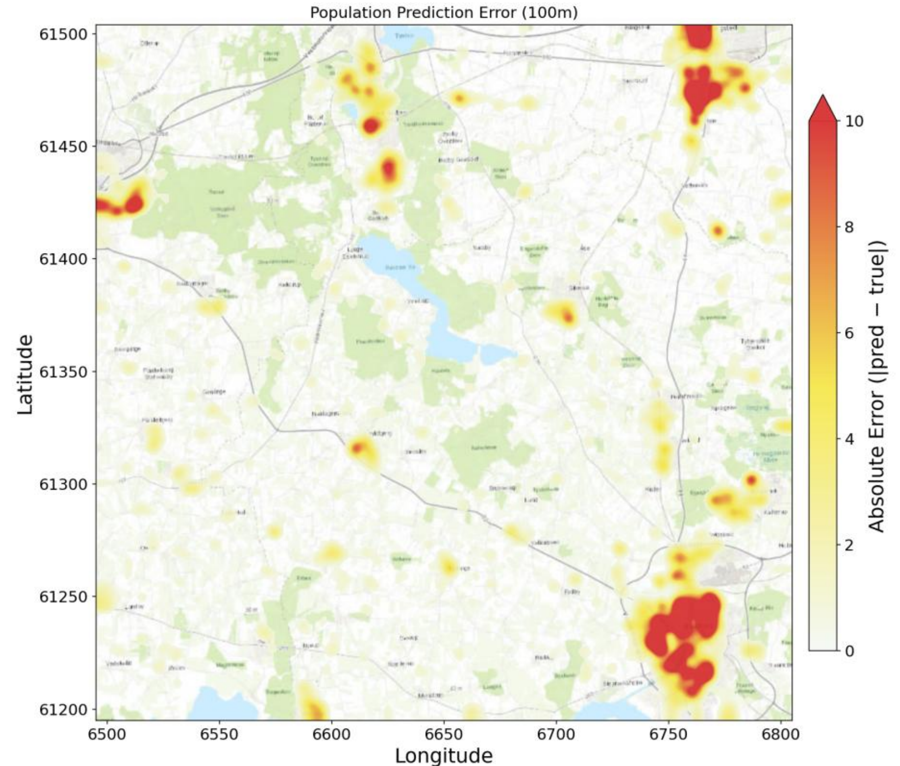
Brightness of color shows density of data points

Results: Error estimation map

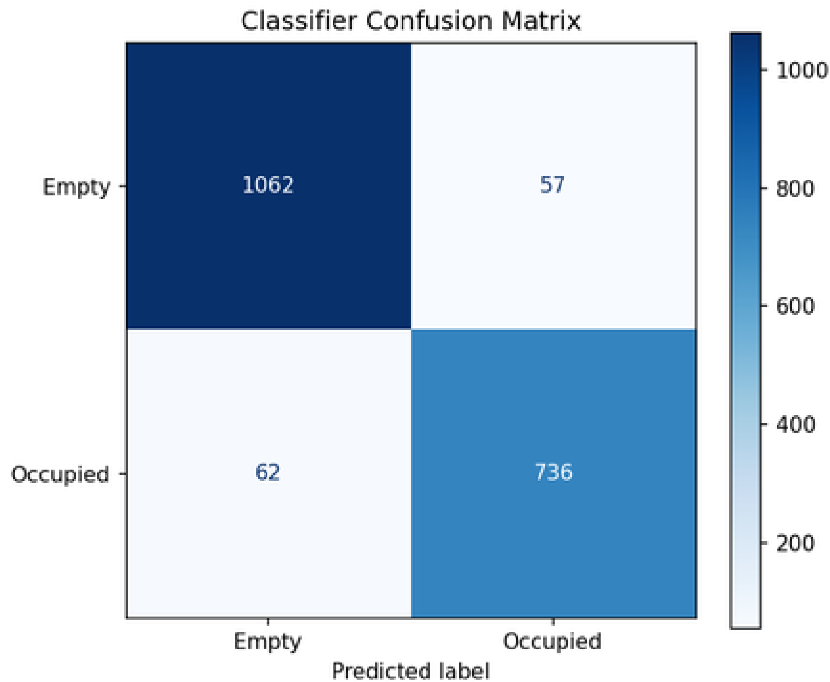
Largest errors in densely populated areas

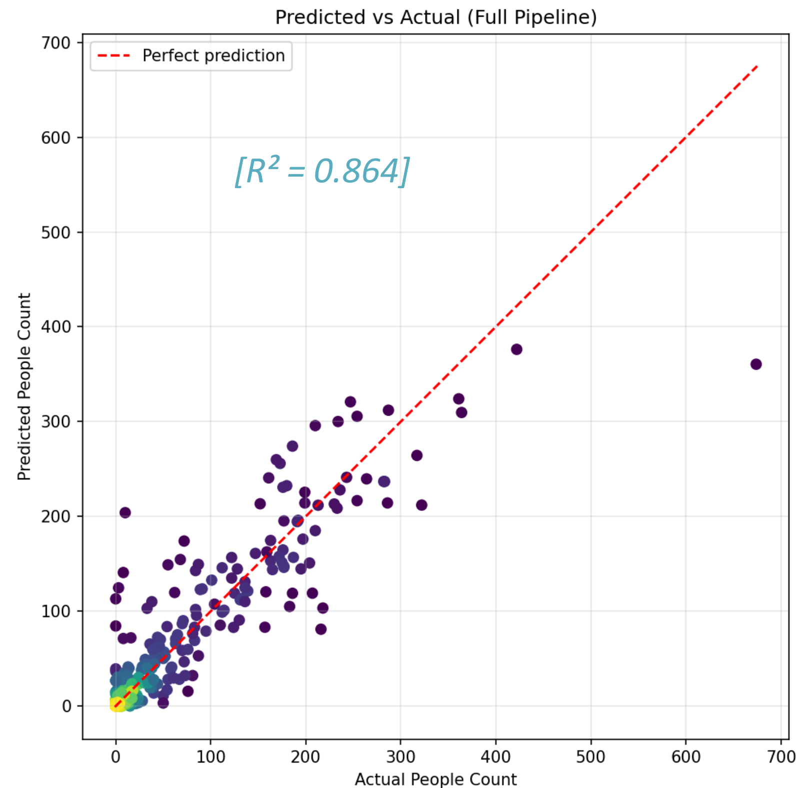
Makes sense, as these areas allow larger numerical errors

Few images from these areas compared to rural areas, so regressor is unprepared



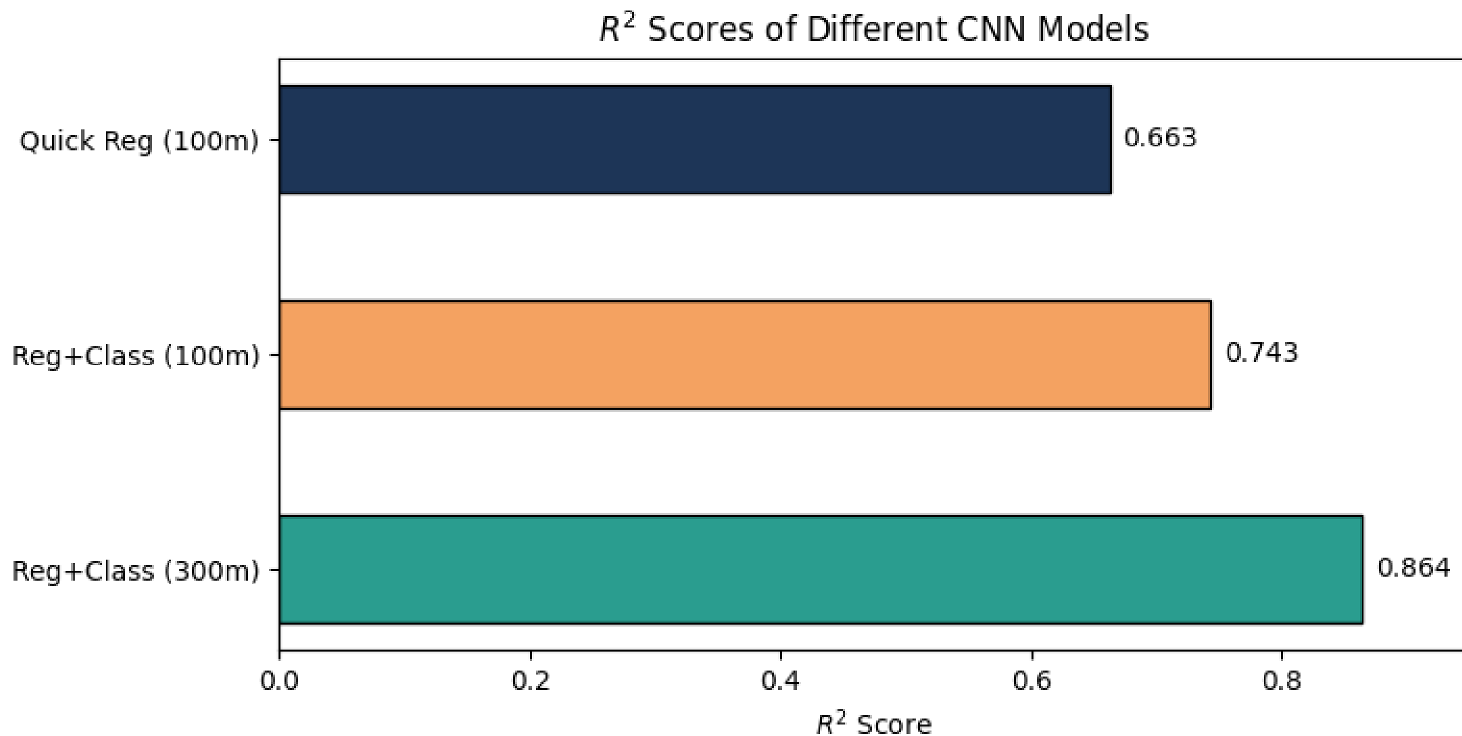
Results: Classifier + Regression (300 x 300)





Classification Performance (Accuracy = 0.938, MCC = 0.872)

Model Comparison



Note that the 3 models are trained and tested on sets of different sizes.

Transformer Model

n: Size of individual data points	Computational power required	Parallelization capabilities	Distance information travels in the model
Layer Type	**Complexity per Layer**	**Sequential Operations**	**Maximum Path Length**
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$

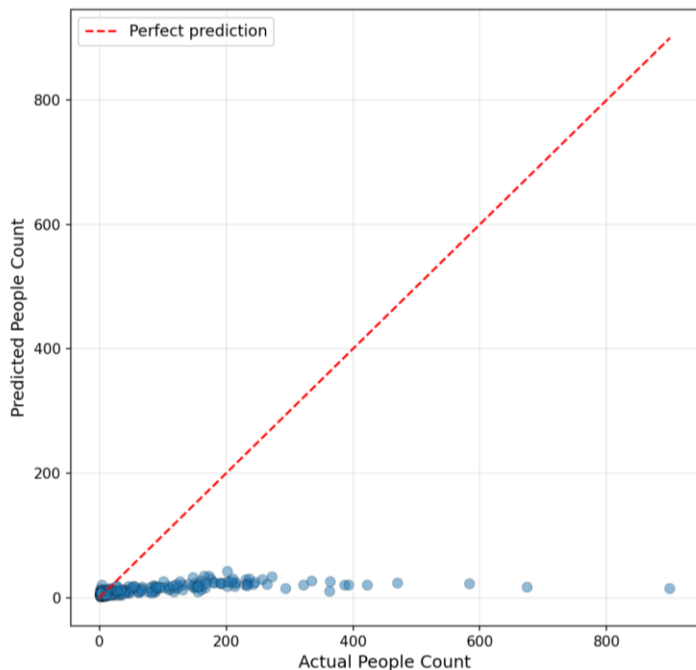
”Attention Is All You Need”. Ashish Vaswani et al. (2023).

Smallest risk of losing information!

Results: Transformer Model 300m×300m

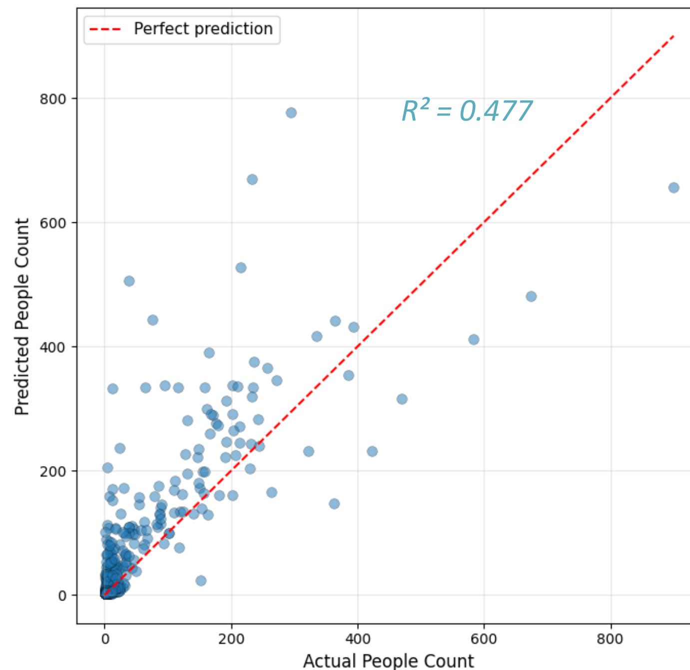
Model parameters: 2,919,937

Initial model



Unbalanced Dataset

Best model

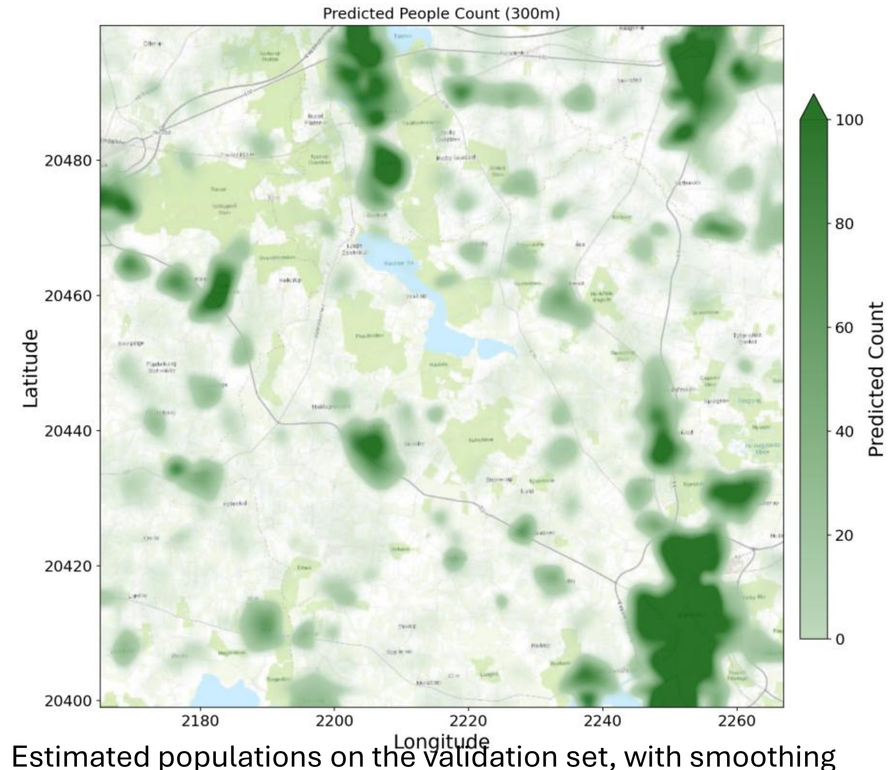


Weighted samples and population > 0

Conclusion

- Best model: Classification + Regression 300 m.
- Added spatial context gives more accurate results
- Largest predictive errors in populated areas.
- Clustering separates classes well.
- Initial transformer underperforms.

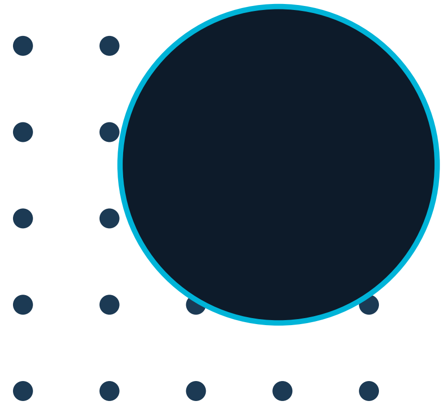
Total True Val. Pop.:	21 794
Total Estimated Val. Pop.:	21 923
Relative error:	**0.59% (!!)**



Estimated populations on the validation set, with smoothing

Applied Machine Learning

Appendix



Appendix – Data acquisition and detailed description

2015 Population numbers from DST distributed on the SquareNet(KvadratNet) were available from Institute of Geoscience, UCPH. Orthophotos taken by GeoDanmark and Klimadatastyrelsen and was also obtained through co-operation with GEO. Orthophotos were available in 10x10km .ecw files, with resolution of 10-12.5 cm per pixel. These were converted using open-source map program Q-GIS, to geographic .tif files, and then split-by-mask into the 100x100m squares used by KvadratNet with resolution 50 cm in another map program ArcGIS, and finally converted to .jpgs. For the full dataset of 88 869 images this process took approx. 100 hours of computing time.

These images were combined with DST population data and saved using the following naming scheme: RunningIndex_SquareSize_LatIndex_LongIndex_Population.jpg.

Ortophoto data was originally 5 band, but because of the heavy pre-processing necessary to convert to .jpgs, only one color band was kept, resulting in grayscale (0-255) images.

Appendix - 300m Dataset

The 300m dataset was created by conjoining 9 100x100m cells and adding together their populations. In a few places this was not possible, primarily along the coast or over a few points with missing data. In these places, no data were generated. This gave a total of 9587 images, of which 5673 where unpopulated and 3914 were populated.

We wanted to analyze this dataset to gain an understanding about how much spatial context is needed for our models, as this is often relevant information when used for geographic purposes. It also has the added benefit of a more balanced dataset, but with the drawback of larger images (600x600 px), which required more computing time for our CNNs and Transformer.

Appendix – 100x100 and 300x300 data set sizes

To each dataset we applied an 80/20 train/validation split which resulted in the below counts:

100x100 (40000 px per image):
Full data set: 88869 images (10001 populated)
Training set: 71096 images (7955 populated)
Validation set: 17773 images (2006 populated)

300x300 (360000 px per image):
Full data set: 9587 images (3914 populated)
Training set: 7670 images (3116 populated)
Validation set: 1917 images (798 populated)

Appendix – Model architecture 100x100

Classifier: 5 convolutional blocks, 256 channels, each convolution block evolves from a 32 to 256 feature map, MaxPooling is applied at the end to reduce spatial resolution and reduces position dependence of features. Dropout is applied with a 0.003 rate, which means hyper parameter optimization found few redundant weights (no over-fitting).

Regressor: Used our TunableCNN() function with parameters: convolutional blocks =5, base_filters = 64, filter_scale = 2.0, double_conv = True, fc_hidden = 256, use_extra_fc = True, learning rate = 9.18e-5, weight_decay = 0.0004, batch_size = 32, dropout = 0.27.

Final Classifier Parameter count: 4,483,873

Final Regressor Parameter count: 9,574,081

Appendix – Model architecture 300x300

Classifier: Same structure, but with dropout = 0.183. Updating this structure for 300x300 might be a possible place to improve our model.

Regressor: Used our TunableCNN() (see slide 32) function with parameters: convolutional blocks =4, base_filters = 32, filter_scale = 2.0, double_conv = True, fc_hidden = 256, use_extra_fc = False, learning rate = 3e-4, weight_decay = 1e-4, batch_size = 16, dropout = 0.3

Transformer: Embedding dimension of 192, 3 Attention Heads with 6 layers each - tuned to fit in GPU memory. Dropout of 0.3 and Learning rate of 10e-4 - standard values applied

Final Classifier Parameter count: 4,483,873

Final Regressor Parameter count: 1,239,649

Appendix – Model training and optimization

The training time for the classifier was 35 min on 300 x 300 images and 28 min on 100 x 100.

Classification Loss function: Binary Cross Entropy with logits (BCEwithlogits)

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

Classifier Hyper-Parameter Optimization: *Optuna* was applied to optimize the following parameters: learning rate, drop-out, Batch-size, weight decay and optimizer.

The time spent finding correct parameters was 26 mins in total with 40 trails and pruning.

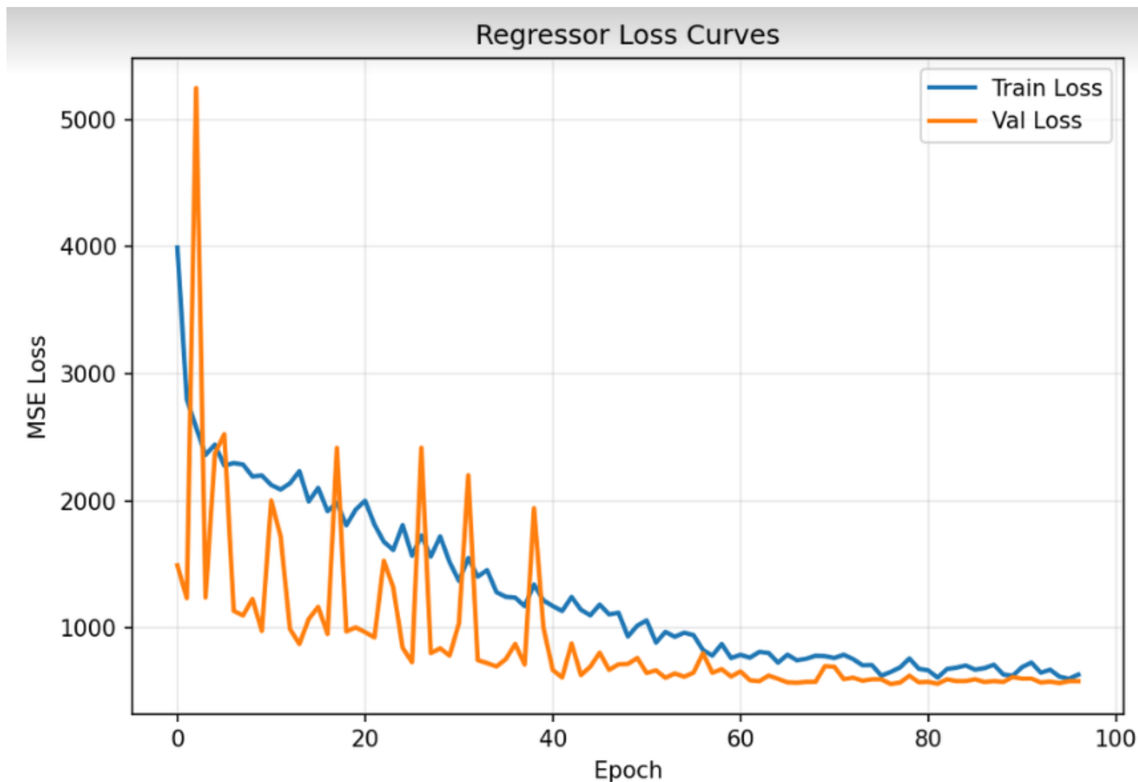
Appendix – Model training and optimization

Regressor Hyperparameter optimization: Entire CNN Architecture along with hyperparameters were tuned using Optuna and a tunable CNN function. For the 300m data Hyperparameter optimization took roughly 8 hours for 100 trials with 10 epochs each (with early pruning for bad loss or memory errors). The final model training lasted about an hour for 100 epochs, which was stopped early at 87 epochs. For the 100m data, HP optimization took roughly 4 hours with a final training time of half an hour. Training and HP optimization was performed using an MSE loss function.

```
class TunableCNN(nn.Module):
    def __init__(self, num_conv_blocks=4, base_filters=32, filter_scale=2.0,
                 double_conv=True, fc_hidden=128, use_extra_fc=False, dropout=0.3):
        super().__init__()
        def _block(in_c, out_c, double):
            layers = [
                nn.Conv2d(in_c, out_c, kernel_size=3, padding=1),
                nn.BatchNorm2d(out_c),
                nn.ReLU(inplace=True),
            ]
            if double:
                layers += [
                    nn.Conv2d(out_c, out_c, kernel_size=3, padding=1),
                    nn.BatchNorm2d(out_c),
                    nn.ReLU(inplace=True),
                ]
            layers.append(nn.MaxPool2d(2))
            return nn.Sequential(*layers)

        conv_blocks = []
        in_channels = 1
        out_channels = base_filters
        for _ in range(num_conv_blocks):
            out_channels = min(int(out_channels), 512)
            conv_blocks.append(_block(in_channels, out_channels, double_conv))
            in_channels = out_channels
            out_channels = int(out_channels * filter_scale)

        self.features = nn.Sequential(*conv_blocks)
        self.pool = nn.AdaptiveAvgPool2d(1)

        head_layers = [
            nn.Flatten(),
            nn.Linear(in_channels, fc_hidden),
            nn.ReLU(inplace=True),
            nn.Dropout(dropout),
        ]
        if use_extra_fc:
            head_layers += [
                nn.Linear(fc_hidden, fc_hidden // 2),
                nn.ReLU(inplace=True),
                nn.Dropout(dropout),
                nn.Linear(fc_hidden // 2, 1)
            ]
        else:
            head_layers.append(nn.Linear(fc_hidden, 1))

        self.head = nn.Sequential(*head_layers)

    def forward(self, x):
        return self.head(self.pool(self.features(x))).squeeze(-1)
```

Appendix – Example of Traning Curve

Example of loss curves for a final training session. Training was typically done for 50-100 epochs with early stopping implemented. We would use the model that produced the lowest validation loss.



Appendix - Regressor Models (as on slide 16)

Model 1:
Regressor trained on images where the **true** population is greater than 0.

Pro: Regressor only has to guess how many people live in an image. Not whether anyone does.

Con: Errors from classifier propagate into regression model.

Model 2:
Regressor trained on images where the classifier **predicted** the population is greater than 0.

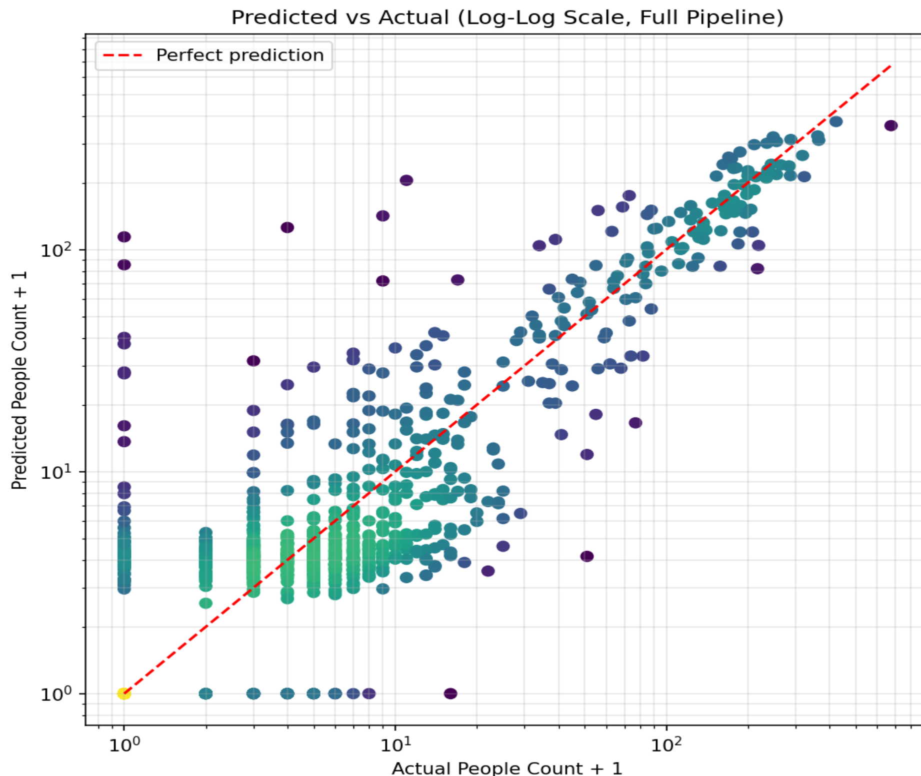
Pro: Adapted to Classifier errors.

Con: Regressor is bad at predicting population = 0. This is what we saw in our quick (and dirty) approach.

Appendix - 300m Data Loglogplot

Class+Reg model

Log-log plot shows more detail in low counts. We note that the classifier errors are less extreme in the 300m model, which highlights how the model has less room for these large errors when context is added.



Appendix – Grad-CAM

Gradient-Weighted Class Activation Map (Grad-CAM) is a tool for visualizing CNN decision-making by coupling pixels at the final feature layer of the CNN (8x8 px for our classifier) through the ordinary NN and finally onto the decision-making layer.

The importance of each pixel is calculated through backpropagation. This produces a SHAP-like metric, associating an importance to each pixel of reduced final-layer-image as though they were input features in an ordinary NN.

The final step is visualization, in which the reduced pixel-importance image is upscaled to the original image size (i.e. 8x8 -> 200x200). The importance numbers are smoothed out in this upcycling, which creates this “attention cloud” effect we see in our images on e.g slide 6 &13.

Introduced in: https://doi.org/10.48550/arXiv.1610.02391

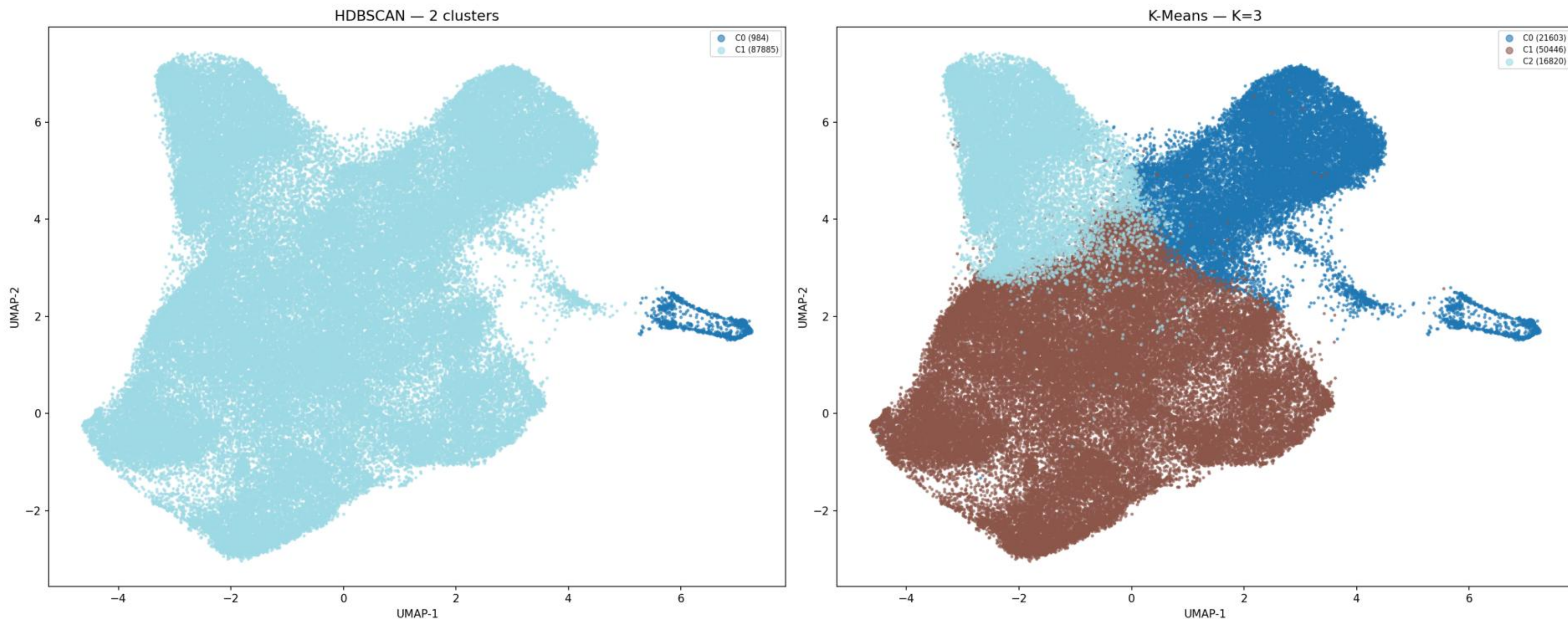
Appendix – Clustering Methods

For clustering, we used a pretrained network EfficientNet-B3 from torchvision, reduces images to 1536 features.

Dimensions reduced with PCA first, 1536->128, then with UMAP 128->20. Then clustered using HDBSCAN (min_cluster=500, min_sample=50) and K-means (k=3,4,5).

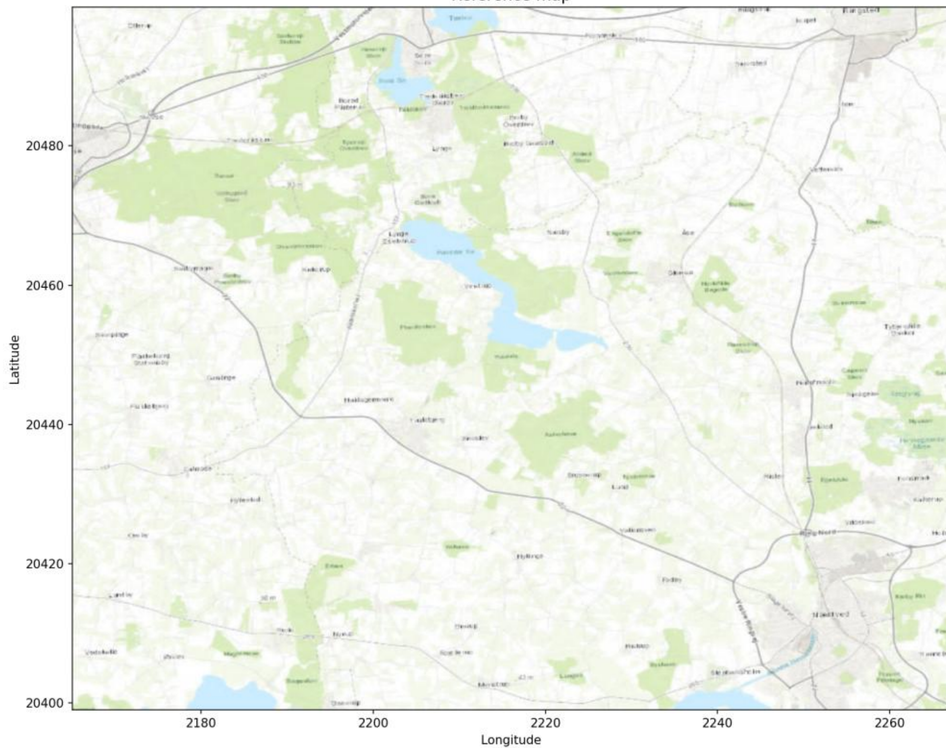
Compared with available map overlays, as in main presentation or in slide 38

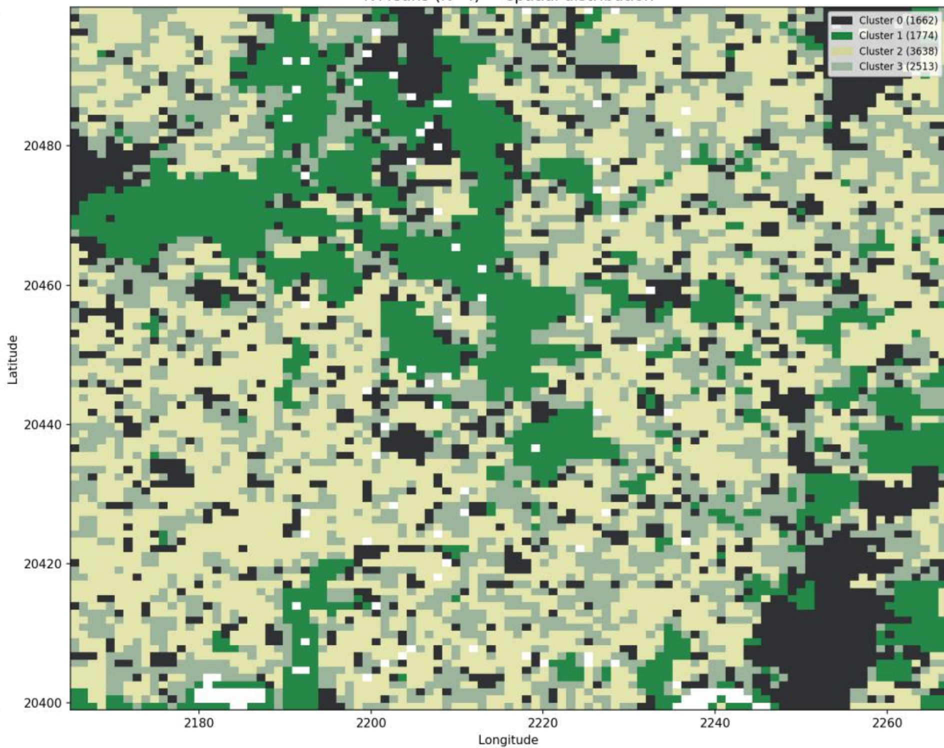
Appendix – Clustering separation in UMAP 2D



HDBScan finds outliers: These pictures are almost entirely monochromatic, features are distorted because of this. K-Means correctly place these within nature (forest+water) cluster. This is visualized in two UMAP dimensions (only used for this plot).

Appendix - Clustering





Clustering results at 300m scale. Structure is more coarse, but still comparable to reference

Appendix – Transformer model performance

Addressing Performance Issues

In the 300m dataset, after restricting the data to be pop > 0, there is on the order of 3k pictures, which is very little data for a Transformer model. This may explain the poor performance. The next step could be acquiring more data and/or "creating" new data by rotating/stretching existing images.

The training data were binned in equally large bins, where a weight was applied to each data-point, depending on which bin it fell into. Here, data points with large population values, were weighted a lot, and therefore contributed the most to updating the model parameters during training. The restrictions these weights gave are quite harsh and including them in a hyper-parameter search might lead to better performance.

Additionally, the model was very small with less than 3M parameters, which may not be enough to fully utilize the powerful properties of transformers.

Appendix – Transformer cont.

Trial runs with the Transformer model

We tried training on a pre-trained Vision Transformer trained on Image-net. But early on it didn't provide as good results as the self-trained model, so it was dropped. This is likely due to Image-net having a large variety of images, where ours are very similar in comparison. This was also limited to 224x224 pixel images, so we had to add black boxes to the 200x200 pixel data-set and the 600x600 pixel dataset would not run without compression or other type of manipulation.

Why is the Transformer Model interesting?

As seen in slide 22, we would expect the model to perform well on the 300m×300m dataset due to its ability to capture long-range dependencies.

Positional Encoding layer is also interesting in itself: Encoding images into a learnt vector space with many orthogonal directions (especially near-orthogonal) may encode vital and a large variation of information, that can be easily distinguished by the model. This would require vast amounts of good data though.

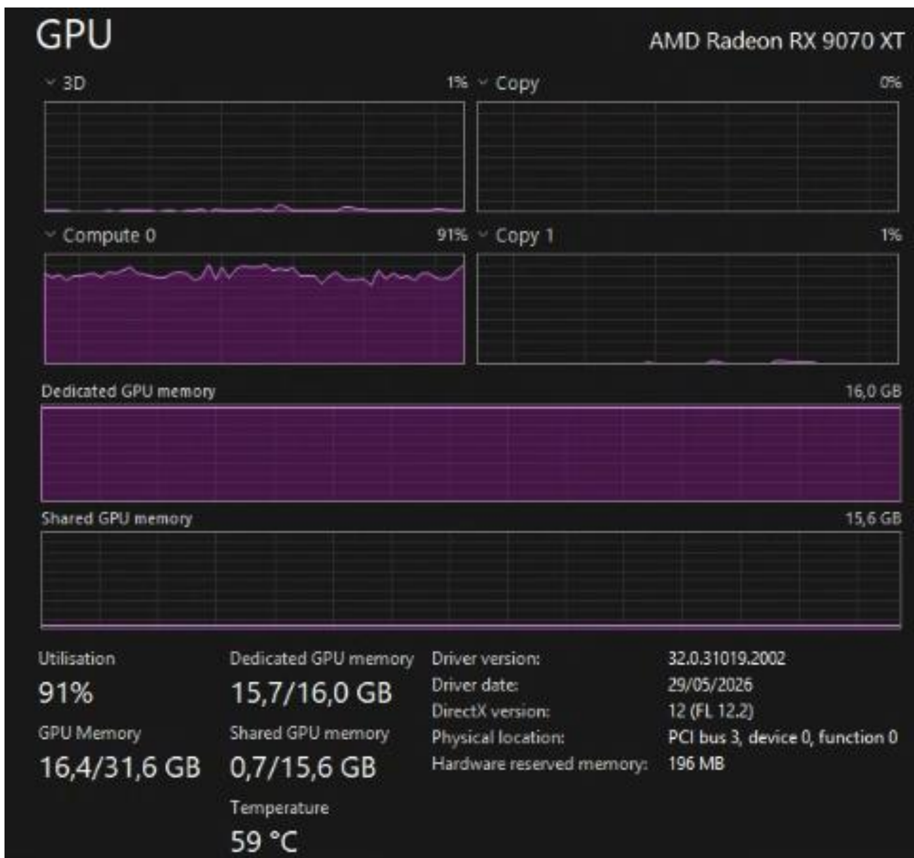
Appendix – Computation hardware & methods

To effectively train our models, access to GPU's was crucial. Therefore, we tested out the following options:

Colab Pro: NVIDIA A100 Tensor Core GPU with 80 GB of VRAM.
ERDA : DAG HPC GPU Notebook – A30 w/ MIG 1g.6gb profile
AMD RADEON RX9070xt: 16 GB VRAM
Nvidia GeForce RTX 5070: 12 GB VRAM

This allowed parallelization in Pytorch using CUDA or torch_directml for AMD. It resulted in substantial speed-up, as training was practically unfeasible on CPU only.

Appendix – GPU parallelization



Example of succesful use of an AMD GPU during training on batches of 16 300x300m images. Note the high Compute utilisation and proximity to the max dedicated GPU memory. We have effectively hit the ceiling for how many calculations the GPU can perform simultaneously.

Note: Pytorch is not able to access shared memory on AMD GPUs on Windows, which caused memory errors during HP optimization. See appendix slide 47.

Appendix – Discussion of 100m vs 300m results

For the 300 meter results the model is given more context, which results in fewer boundary errors with partial houses (note that the population data consists of points, and do not cover the full houses). We expected that this would make it easier for the regressor to guess the population, since the outlier edge cases in each grid will be 'averaged out', resulting in more consistent estimation.

Training and validation sets are also much more balanced for the 300m dataset, due to populated and unpopulated data samples getting grouped together, which is also why we see the accuracy drop for the classifier: There are suddenly less trivial cases, where the picture is obviously just taken over a lake or densely forested area. However, we observed that the MCC-score improved slightly, suggesting that greater context still improved classification on the more difficult cases from 100x100m.

Appendix – Lingering questions

Why not use infrared data as well?

- While this is available for large parts of Denmark (but not in our dataset), it is not always available, especially not on a global scale. We wanted to test the limits of CNN regressors on minimal information, as could be relevant in less developed regions.

Why use R^2 as a metric?

-This metric shows the correlation between true and predicted labels. It has the advantage of being more visually readable as well as allowing for easier comparison between datasets of different sizes and regressions on numbers in different scales (e.g. 100m vs 300m).

Why did you choose that specific region for analysis?

-Initially we wanted to look at all of Zealand, but this was unfeasible because of the heavy pre-processing needed to extract images. The selected region was chosen because of its square characteristics, mix of urban and rural communities, and because it did not have very densely, vertically populated regions.

Appendix – Lingering questions pt 2

Why use Matthews Correlation Coefficient as a metric (compared w. accuracy)?

-MCC is the go-to descriptor for unbalanced datasets, as a high score (close to 1) is only achieved when the classifier is performing well on all classes. This makes it more desirable compared to accuracy, since a highly unbalanced dataset can give a high accuracy to a classifier which only learns this underlying distribution. We use both descriptors for comparison with other models, and because our datasets are unbalanced to varying degrees.

Are orthophotos and DST population data compatible?

- The original orthophoto files were generated so that the edge of our 30x30 km region matches the DST SquareNet. This ensures that no information is double-counted or ignored. Unfortunately, our masking algorithm padded some files with empty pixels, which we had to remove after cutting. Secondly, both datasets were from 2015, with the photos being taken in spring. Thus, we argue that any potential discrepancies between datasets caused by moving, demolition and/or construction are minimal for this region and our project.

Appendix – Mistakes we made and corrected

- Classifier and regressor models were initially trained on different subsets of full data, leading to validation on training data in the joint model (data leakage).
- Forgetting to save model weights after training.
- Trying to utilize more RAM than available when loading data.
- Choosing Hyperparameters that would make the network training so large it could not fit on the GPU memory. Fixed by using optuna.TrialPruned().
- Unzipping 300 k files to a single folder on ERDA (This crashes the server)
- Pytorch num_workers for parallelization of data loading is bugged when running a jupyter notebook. It works when running .py files instead. This gave a massive speedup to training since data loading bottlenecked GPU-utilization.

Appendix – Future exploration

More things we could have explored:

- Training on more data and different data (different parts of Denmark, different countries etc.) could test model resilience and usability in other parts of the world.
- Training with ResNets could potentially amplify our results and serve as a check if our CNNs correctly get all features of our data.
- Optimize transformer e.g. by adding more data, larger model.
- Train with full set of RGB-values.
- Adding weights to data as alternative to the Class->Reg pipeline for simplification.
- Make predictions dependent on neighboring area predictions. Could potentially decrease outliers but may also be problematic in rural areas where a house might be ignored just because it is surrounded by nature.

Appendix – Project contributions

All participants contributed evenly to all parts of the projects.