Nikita Rudin
260809135

Classifier analysis

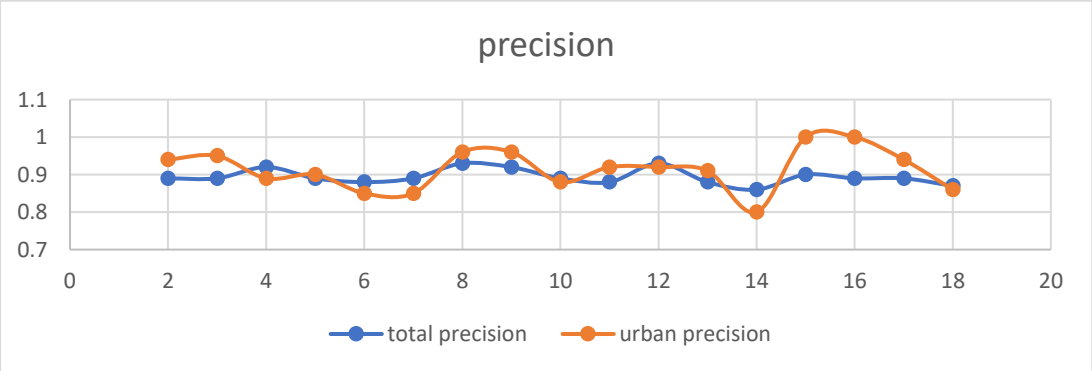
I am plotting the precision, recall and f1-score for the total and the urban tiles (since this is what interests us).

$$Precision = \frac{correctly\ classified\ as\ urban}{classified\ as\ urban}$$

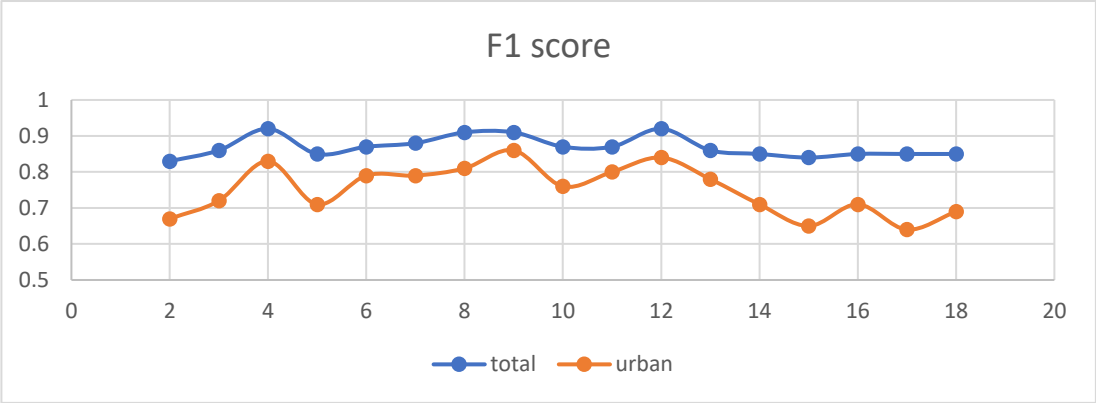
$$Recall = \frac{correctly\ classified\ as\ urban}{really\ \ urban}$$

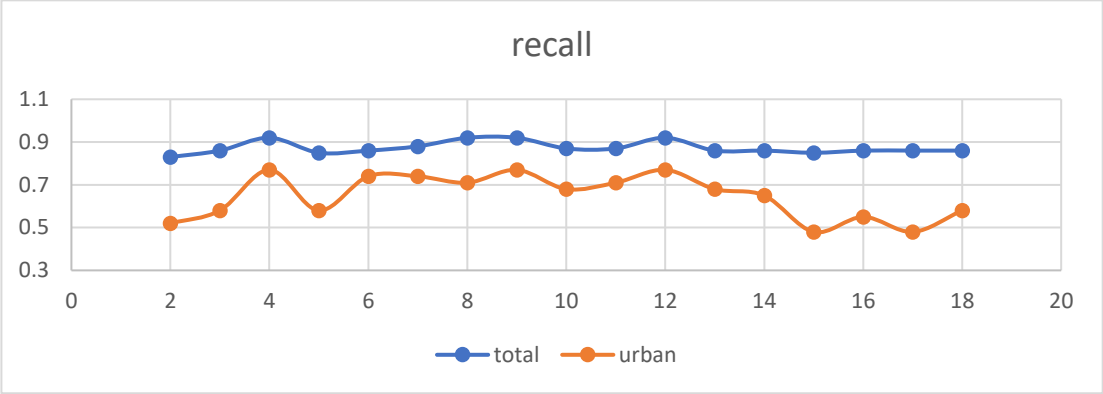
(The f1-score is weighted combination of recall and precision)
In our case recall of urban tiles is more important than precision, because a classifier that classifies very few tiles as urban can have a high precision but is not what we want.

After trial and error using 9 components provides good results. We can see that it is one of the highest f1-score on the plot for both total and urban.
Interestingly, at for 15 components the precision of urban tiles is very good. It would seem that it would produce good results. Unfortunately the opposite happens. My explanation is that for that number of components the classifier tends to put less tiles in the urban class. The ones that end-up being classified as urban are actually urban, but we missed a lot of other tiles that should have been in that category.
In general, we see that increasing the number of components doesn't increase the performance. I believe that this is due to the fact that for a higher number of components tiles of the same category start to appear different. Two urban tiles with different buildings will have further from each other. As a limit case, if we compare every pixel, all tiles will be very different and the classification will be pretty random. This is especially true, since tiles present some characteristics of other categories (example: trees in urban areas).



Nikita Rudin
260809135





General notes on the implementation:

1) Strangely, some tile images (always the same) fail to load on the map and stay as blurred even though the tiles are correctly classified. This might be linked to the fact that I am implementing this on windows.
2) I decided to neglect dynamics of the drone and allow a step in velocity, which involves an infinite acceleration. This means that the drone can make very sharp turns,s which is representative of a small drone compared to the scale of the map.
3) I have also tried to use a sort of "Roomba" algorithm, where the drone bounced of urban tiles, but it always ended up stuck between two urban tiles.

First Algorithm – Brownian motion

I implemented a drone moving in a Brownian motion in the physical sense. This means that at every step, the drone takes a step in a random direction. To make things simples the step has a fixed length. Algorithm:

- When the drone enters an unknown cell, it sends it to the classifier and stores the result in a list.
- If the cell is classified as urban, the drone takes the opposite step that brought him there.
- The drone chooses a random step.
- Before taking the next step the drone checks in the list of known tiles if the position he is going to is not urban.
- If it is urban, the drone chooses an other step and repeats.

Results (step of 20 px)

As expected, the results are not great, the drone spends a lot of time at the same place and doesn't explore enough. From physics the travelled distance should be proportional to $\sqrt{t}$. I capped the total distance to 25000 pixels.

Results – Brownian -start over river	
arable	26.25%
water	68.75%
urban	5%
desert	0.0%
undiscovered tiles	320
discovered tiles	80
total visited tiles	560
Ratio	14.28%

Results – Brownian -start over river	
arable	51.92%
water	13,46%
urban	34.61%
desert	0.0%
undiscovered tiles	348
discovered tiles	52
total visited tiles	330
Ratio	15.57%

Nikita Rudin
260809135

Nikita Rudin
260809135

2nd Algorithm "Curious drone"

This time, I implemented an algorithm that always looks for unknown cells. It is complete in the sense that it will always visit all possible tiles. It is not optimal, but I increase the performance with some tricks listed below. Algorithm:

- Find the closest unknown cell using a modified Breadth first search algorithm.
  I push the drone towards the bottom and right by adding these neighbors first in the queue
  If the neighbor is urban it is not added in the queue
- Stop the search once an empty cell is found and return the path
- The path contains the centers of the cells which need to be visited to arrive to the next empty one.
- The objective is set to the first cell in the path
- The drone goes in the direction of the objective
- Once the drone is close to the objective the next cell in the path is set as the objective
- Once the path is empty the search is run again
- If the drone enters an urban tile, the objective is set to the last non-urban cell and the path is emptied
- When no empty cell is found the simulation is stopped, the drone has visited all possible cells

Note: I am only using down,right,left,up (in that order) as neighbors. An interesting improvement would be to use the diagonal cells as well.

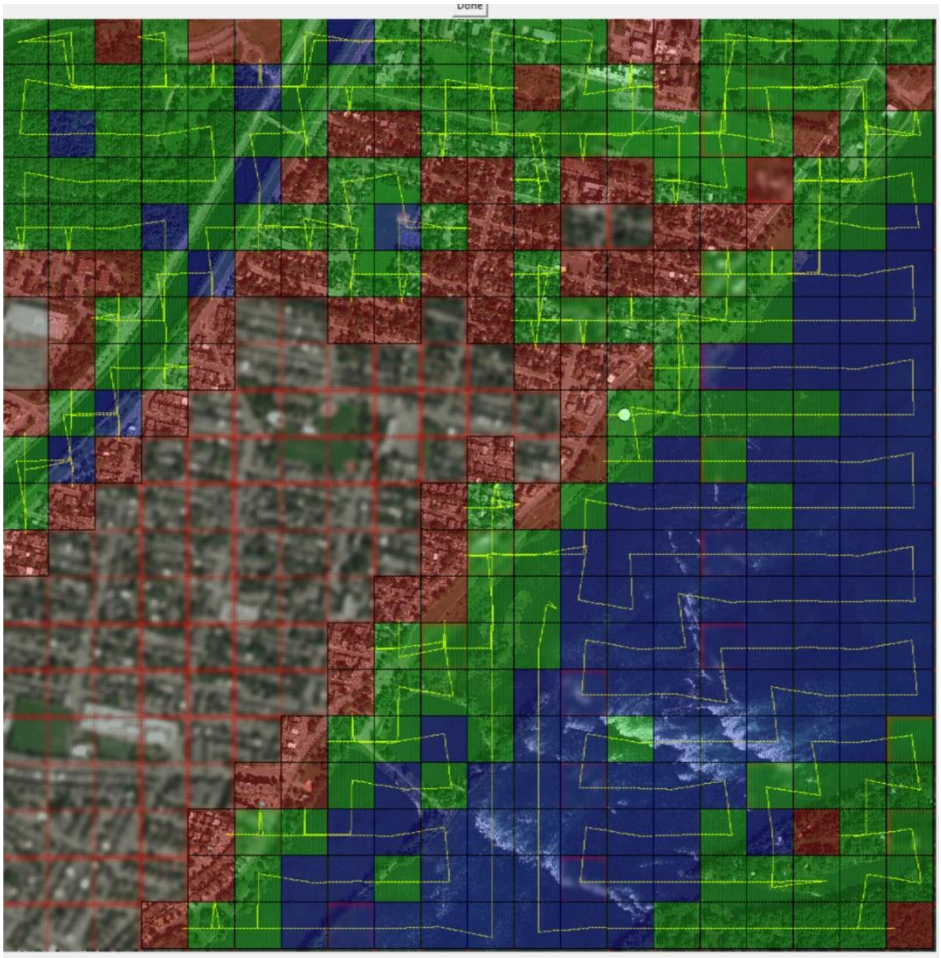
Results:

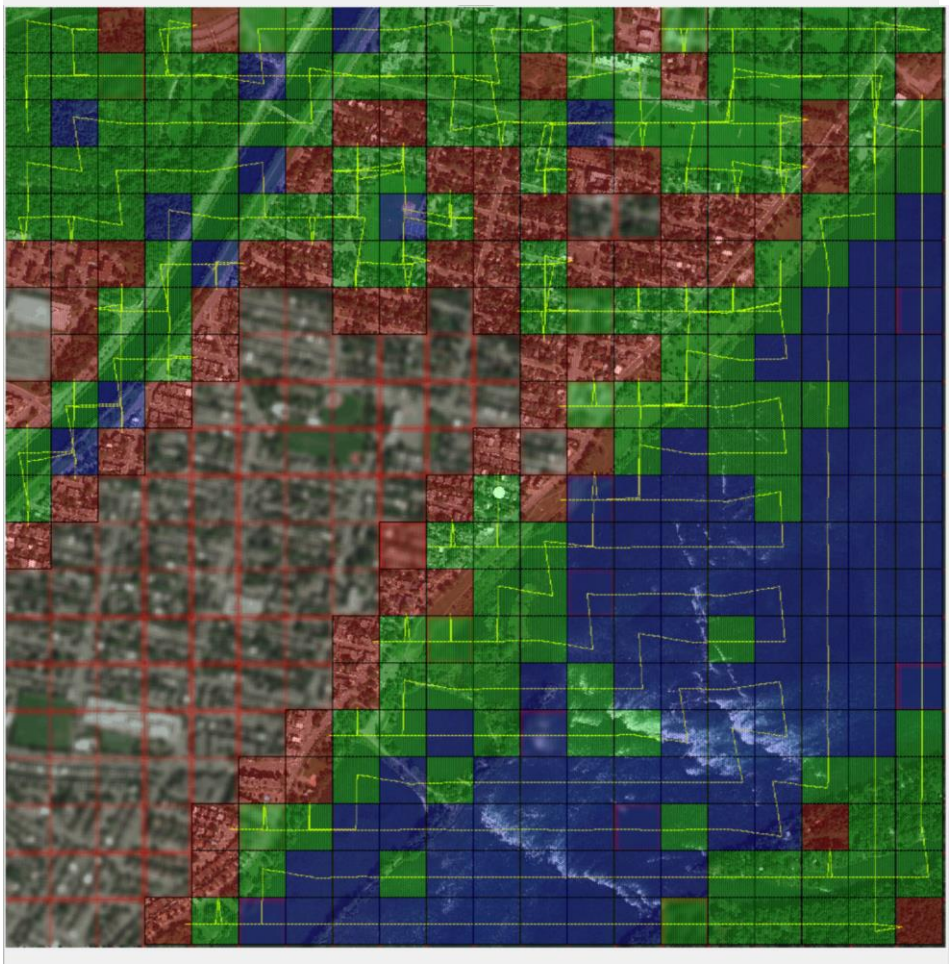
As expected the all possible cells are visited. We can see that the path is not optimal. The drone has to come back to visit some forgotten cells (there is no known way of finding a optimal solution).

Results – Brownian	start over river
arable	44.87%
water	34.29%
urban	20.83%
desert	0.0%
undiscovered tiles	88
discovered tiles	312
total visited tiles	468
Ratio	66.66%
Total distance	22120

Results – Brownian	start over canal
arable	44.87%
water	34.29%
urban	20.83%
desert	0.0%
undiscovered tiles	88
discovered tiles	312
total visited tiles	453
Ratio	68.87%
Total distance	21260

Nikita Rudin
260809135





Conclusion

It is clear that the second algorithm is much more efficient in in an exploration or search mission. I don't really see any pros of the Brownian motion except simplicity. It is true that the 2nd algorithm requires much more computation effort when the number of tiles increases. It's major advantage is that it will always visit all possible tiles. The path length is of course not optimal, but looking at the results it is acceptable.