Capstone report

Object detection on Satellite Images, by Roberto Jiménez Sánchez

Project Definition

The project is framed within the computer vision field, in the object detection domain. The applications of object detection are many, to provide just one example it could be used by self driving cars to detect what type of obstacle is in front of the car. This project in particular deals with object detection from satellite images. There is a growing number of satellites orbiting around Earth and in some cases, their mission is to take pictures of Earth's surface. The applications for algorithms of object detection for satellites are many; the example in particular that inspired this project comes from Ayush et al. (2020), that use satellite images to characterize poverty in a country by region. In another line of work, Biermann et al. (2020) use satellite images to detect plastic debris in the ocean.

Problem statement

To the best of the author's knowledge, object detection in itself does not seek to solve a particular problem. But many problems can be solved by an algorithm capable to perform object detection on satellite data. Going back to the example of poverty, as mentioned by Ayush *et al.* (2020), a common problem when dealing with poverty is that the data is very scarce. When it is available for a region, it might not be updated periodically. Thus, it is difficult to assess the performance of the policies that have been put in place to mitigate poverty. Ayush *et al.* (2020) propose using satellite Earth images to fill this gap. This project goes on the lines of the work of Ayush *et al.* (2020). However, the scope of the project is limited to obtaining objects from satellite images, the steps following to produce the poverty maps have been considered out of the scope.

Dataset

The dataset that has been used is the one provided by xView challenge. This dataset is composed of satellite images containing different types of objects at different regions in Earth and covers 1400km^2 of surface. 847 images are available to the public, with 0.3m resolution, containing over 1 million objects. The objects are distributed over 8 parent classes which are divided further into 60 child classes. As an example, a parent class is "Passenger Vehicle" and the associated child classes are "Bus" and "Small car". The topology of parent classes is provided here below:

- Fixed-Wing Aircraft
- Passenger Vehicle
- Truck
- Railway Vehicle

- Maritime Vessel
- Engineering Vehicle
- Building
- None

The “None” class contains classes with no common parent; there is no coherence between the objects of the “None” parent class, as opposed to for the other parent classes. The detail of the child classes is provided below, from Lam *et al.* (2018):

Fixed-Wing Aircraft	Passenger Vehicle	Truck	Railway Vehicle	Maritime Vessel	Engineering Vehicle	Building	None
Small Aircraft	Small Car	Pickup Truck	Passenger Car	Motoboat	Tower Crane	Hut/Tent	Helipad
Cargo Plane	Bus	Utility Truck	Cargo Car	Sailboat	Container Crane	Shed	Pylon
		Cargo Truck	Flat Car	Tugboat	Reach Stacker	Aircraft Hangar	Shipping Container
		Truck w/Box	Tank Car	Barge	Straddle Carrier	Damaged Building	Shipping Container Lot
		Truck Tractor	Locomotive	Fishing Vessel	Mobile Crane	Facility	Storage Tank
		Trailer		Ferry	Dump Truck		Vehicle Lot
		Truck w/Flatbed		Yacht	Haul Truck		Construction Site
		Truck w/Liquid		Container Ship	Scraper/Tractor		Tower Structure
				Oil Tanker	Front Loader		Helicopter
					Excavator		
					Cement Mixer		
					Ground Grader		
					Crane Truck		

The dataset images are from 2000pixels to 5000pixels per side. The objects vary in size from 3 meters to greater than 3000 meters (so 10 to 10000 pixels respectively). The dataset is split into train, validation and test sets. Only the train and validation sets are available to the public, and the validation set has no labels or bounding boxes provided to the public. An image example (cropped and resized to smaller size) is provided below.

Figure 1 Sample image from the dataset (the image has been cropped and reduced in size)

This dataset allows training a model to detect objects from satellite images, which can be used as a first step on the process to produce poverty maps, as detailed in the next section “Solution Statement”.

Practical aspects of the dataset

Training and validation images are provided in separated *.tar* files that can be found in the xView challenge website.

Training labels are provided in a *.geojson* file that contains information of the objects' bounding boxes, provided in coordinates of two opposite corners (the provided corners are the upper left and the bottom right ones), the class label, the image on which the bounding box appears, and the geospatial coordinates.

Additionally, a file is provided to map a label index to its class name, available in xView project repository.

Metrics

Precision and Recall

Precision and recall, as typically defined for classification problems, will be used here. We can calculate a class based score, or an average score for all classes:

$$Recall = TP/(TP + FN) \qquad Precision = TP/(TP + FP)$$

TP are true positives, FN false negatives, and FP false positives.

Mean Average Precision

The model will also be evaluated with the metrics provided at the xView challenge, which is the mAP (mean Average Precision). The mAP is the mean over classes of the AP for each class (Henderson and Ferrari, 2016). The AP of a class is given by the area under the precision/recall curve for the detections. An example of such a curve and how it is constructed is provided in Figure 2.

The steps provided following are obtained from Henderson and Ferrari, 2016, and completed with the Medium article "mAP (mean Average Precision) for Object Detection" by Jonathan Hui. To understand the curve, it is first necessary to define the overlap between two boundaries, which in this project will be defined by the IoU, the Intersection over Union. The IoU of two boundaries is simply defined as:

$$IoU = (area\ of\ intersection)/(area\ of\ union)$$

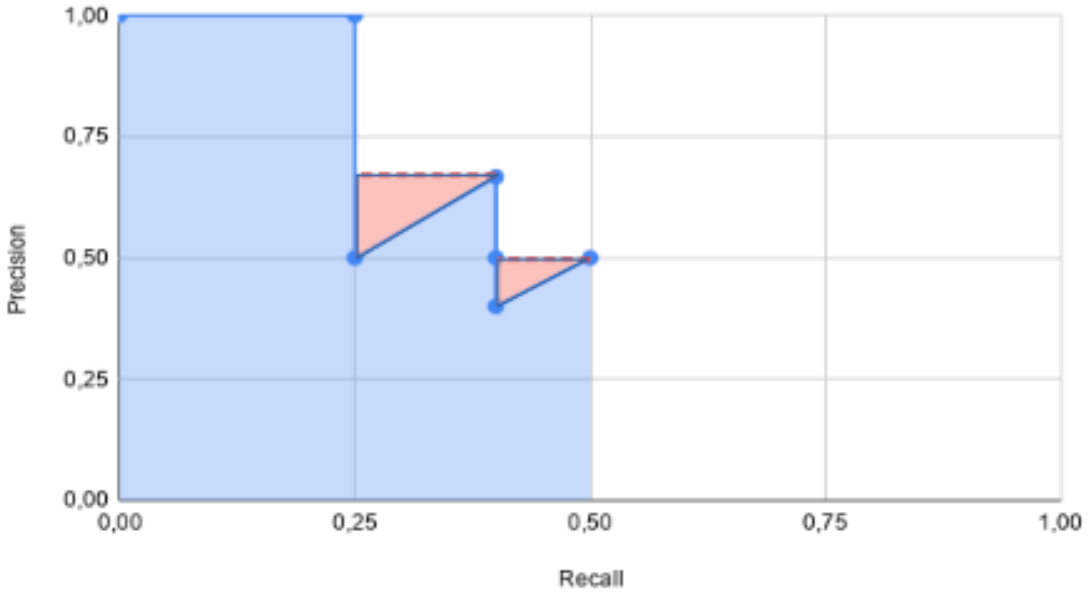

Figure 2 Example of precision/recall curve

To construct the PR curve the following steps need to be followed (we are working with a single class at this stage):

1. Map each detection provided by the algorithm to its most overlapping ground truth instance, as long as the overlap is larger than a threshold IoU value.
2. The highest scored detection mapped to a ground truth instance is considered as a true positive. All the other detections are false positives. Multiple detections of the same object are considered to be false detections. The ground truth objects that have not been detected are considered as false negatives.
3. The predictions are ranked from higher to lower confidence scores.
4. Recall and precision are computed progressively advancing through this rank of predictions, by recurring to their common definitions, except for the fact that only a subset of data will be used at each stage.

Below, the example that yields the precision/recall curve in Figure 2 is provided. In this example we assume that there are 6 ground truth instances of a given class. The algorithm has made 6 predictions, of which 3 are mapped to ground instances because their IoU is larger than the defined threshold. As a consequence, 3 predictions have no associated ground truth instances and FN=3 for all the steps.

Before calculating the area under recall precision curve, the jigsaw effect is smoothed, taking the highest precision value to the right for any given recall value. This is represented by the red triangles in Figure 2. The AP value is obtained then as the blue area plus the red area depicted in Figure 2.

Two mAP values are provided in the results. The first mAP[0.5] is the mAP value calculated for an IoU threshold of 50%. The second value, denoted mAP[0.5-0.95] (Ren *et al.*, 2016) is the average mAP over different IoU thresholds, from 50% to 95% with steps of 5%.

Rank	Correct?	Cumulated TP	Cumulated FP	**Recall**	**Precision**
1	TP	1	0	**0,25**	**1**
2	FP	1	1	**0,25**	**0,5**
3	TP	2	1	**0,4**	**0,67**
4	FP	2	2	**0,4**	**0,5**
5	FP	2	3	**0,4**	**0,4**
6	TP	3	3	**0,5**	**0,5**

Analysis of the problem

In this section, we begin by reporting the benchmarks found associated with this dataset. Then, an exploratory data analysis is presented, looking at features such as image sizes, bounding boxes sizes, and distributions across the classes. To conclude, we move onto briefly discussing a couple algorithms that have been used to complete this project.

Benchmarks

Several benchmarks have been found associated with this data, they are reported here. xView challenge provides a baseline model based on TensorFlow, which details can be found in Lam *et al.* (2018). The model uses the Single Shot Multibox Detector meta-architecture (SSD). The model is evaluated recurring to the whole set of images. To train the model, the images are fed in smaller sizes; three approaches are used: (i) dividing the images into images of $300px^2$ (Vanilla dataset); (ii) using a multi-resolution dataset, which creates image chips of different sizes ($300px^2$, $400px^2$ and $500px^2$, Multires dataset) and (iii) an augmented dataset, which uses the multi-resolution dataset and adds image augmentation (shifting, rotation, noise and blurring) (Augmented dataset).

The results are evaluated using 30% of the data by recurring to the mAP metric (which is detailed in the evaluation section below):

	Vanilla dataset	Multires dataset	Augmented dataset
mAP	0.1456	0.2590	0.1549

Another benchmark model has been found in Medium, by Fong, 2018. In this case, the model used is YOLOv3, and the mAP score is 0.085. For this second benchmark only the train images were used to train and evaluate the model, as will be the case for this capstone project, and it might be more appropriate for comparison.

Finally, Ayush *et al.* (2020), using the parent classes and three child classes, report a mAP of 0.248, using YOLOv3.

Exploratory Data Analysis

We begin showing a sample image (cropped) with some of its bounding boxes drawn:

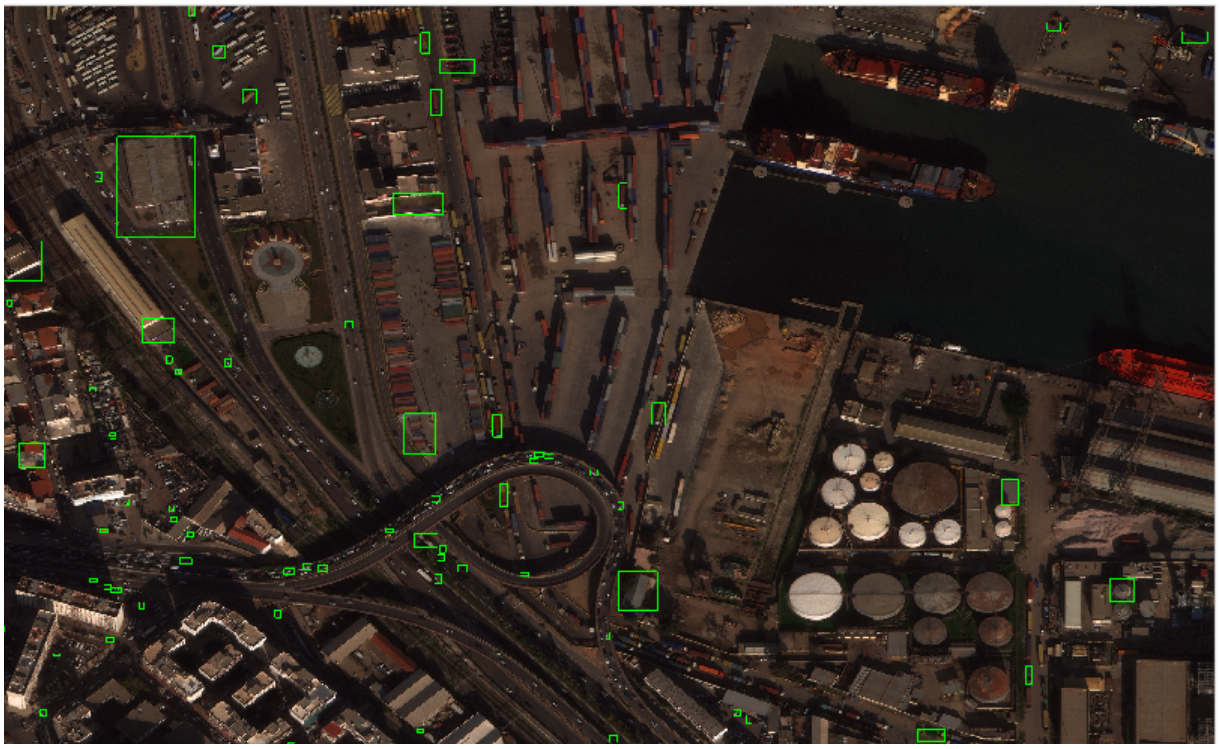

Figure 3 Cropped satellite image with some bounding boxes drawn

We can see already looking at this image that there are important differences in size for the bounding boxes of the different objects.

The analysis of the data has focused on:

1. Assessing issues that might need cleaning, related for example with missing images, or bounding boxes that make no sense
2. Looking at the bounding boxes in detail, distribution of sizes and shapes. All of them together and the distributions across classes.
3. Deciding on whether the child classes, the parent or the parent alternative classes should be chosen.

Identifying data problems

Missing data

The data has been verified for missing values. Two checks have been carried out:

- If all images listed in the file containing the bounding boxes coordinates and labels are available,

- If there is no missing information associated with the bounding boxes.

Once the **images** were extracted from the *tar* file, the image width and height associated to an image id were obtained by using the bash command *file*. The image ids obtained from the *tar* were compared to the image ids available in the bounding boxes file, and it was observed that one of the image IDs in the bounding boxes file was not available as image.

The **bounding boxes** file was also checked for missing (NaN) values and none have been found in the columns used for our model (coordinates relative to image ID, labels and image ids). The world coordinates have not been checked.

Invalid data

Images sizes

The shape of the images has been checked to assess if there were outliers:

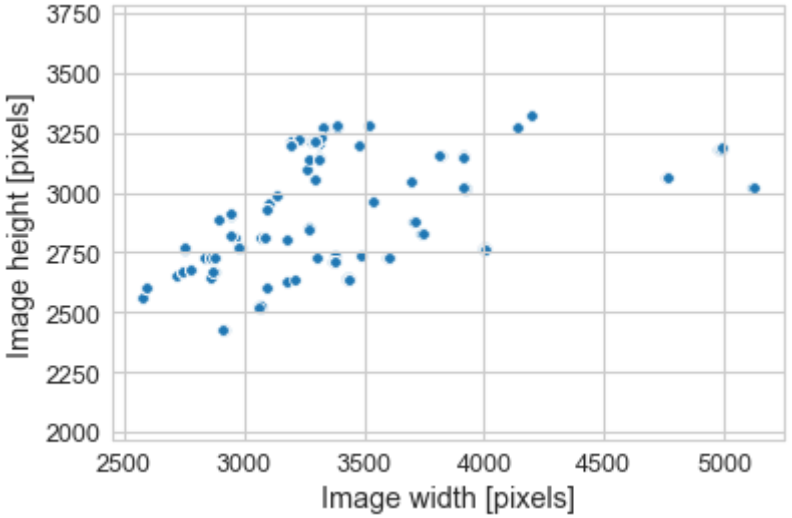

Figure 4 Height and width of the images in the dataset

There seems to be some images exceptionally large in width (43 images are above 4500 pixels in width, the scatter plot is a bit misleading as some points overlap). Some of those have been visually inspected and nothing has been found indicating a quality issue with them. So it has been decided to keep them. The images sizes are too large to be sent as-is to an object detection model. They will need to be either resized or cropped into smaller images that can be sent to the algorithm for training and making predictions.

Also, their size in Mb has been checked. Images' sizes range from 19.8Mb to 47.8Mb. Larger images in Mb are also larger images in pixels, so no inconsistencies have been found associated with the images.

Bounding boxes

A bounding box is defined by its upper left and bottom right corners, as below:

Figure 5 Bounding box definition

The bounding boxes have been verified for the following quality and consistency issues.

- **No size or wrong corners** order: from the definition of a bounding box, x1 and y1 should always have strictly larger values than x0 and y0. There were 9 bounding boxes that did not fulfill this condition.
- **Outside of image**: it has been verified if the coordinates were completely or partially outside the limits of the corresponding image. For this step it was important obtaining the image sizes first. There are many more bounding boxes that do not fulfill this condition, more on this issue below.
- Bounding boxes **labels**: it has been checked, as the numbers were float numbers (and the classes are integers) that there were no decimals in any annotations. It has also been verified that all assigned labels had a matching name in the mapping file, which was not the case. 79 bounding boxes had annotations with no name assigned.

Regarding the bounding boxes that were out of the image, 23 of them were completely out of the corresponding image. The bounding boxes that were partially out of their image have been assessed in more detail. There are a larger number of them, around 10600. Some of those bounding boxes have been plotted to observe if they contained an object or not, and if the object is coherent with the label assigned. Below there is a couple examples:

Figure 6 Examples of bounding boxes which coordinates surpass the image limits

All bounding boxes that were looked at contained an object that seemed correctly labelled. It was decided to keep these bounding boxes, correcting the coordinate(s) that were outside the image limits.

Analysis of data features and characteristics

The data has been analysed looking mainly at two things:

- The distribution of bounding boxes across the different labels available,
- The sizes of the bounding boxes.

The first point is important to understand if there is any class imbalance in the dataset, this could guide the choices for things such as training the data. The second point can guide the choice of the model to use for object detection and the adjustment of any associated hyperparameters.

Analysis of distribution of bounding boxes across labels

Part of the analysis presented here focuses on choosing which classification of the labels should be used to train the model. As indicated in the description of the dataset, the bounding boxes can be classified according to:

- Child classes, of which there are 60,
- Parent classes, of which there are 7 classes (or 8 if None parent class is considered as an actual class),
- Another classification possibility has been assessed, which will be referred to as “Parent Alternative” classes. This classification considers the Parent class, for the objects where there is one, and the child class when there is no Parent class (so for all the child classes grouped under the “None” category). This yields a total of 16 classes.

For the purpose of creating the poverty maps, which is the ultimate goal of the model started in this project, all three classification systems can be useful. Using the Parent Classification system and including None as a class has been discarded, as the objects grouped under the None label are too different between them and it seems that a model would struggle to identify objects belonging to this “class”.

The Figure 7 below shows the count of bounding boxes for each class, as provided directly by the labels. As can be observed in the figure, there are huge differences in the number of occurrences of child classes, class imbalance is quite important. The most found class is **Building**, with almost 320k occurrences, and the least represented is **Railway Vehicle**, with only 17 occurrences.

It shall be noted that both Building and Railway Vehicle are parent classes actually, and not child classes. This is not an issue with the data, it was done on purpose by the creators of the dataset. According to Lam *et al.* (2018) this is due to the way the annotations are done, when the labeler is unable to make a confident determination from the child classes, the object will fall back to the more general parent category. The two most common classes, building and small car, represent roughly 87% of the objects found in the dataset. There is 9 classes with less than 100 examples for the child classification.

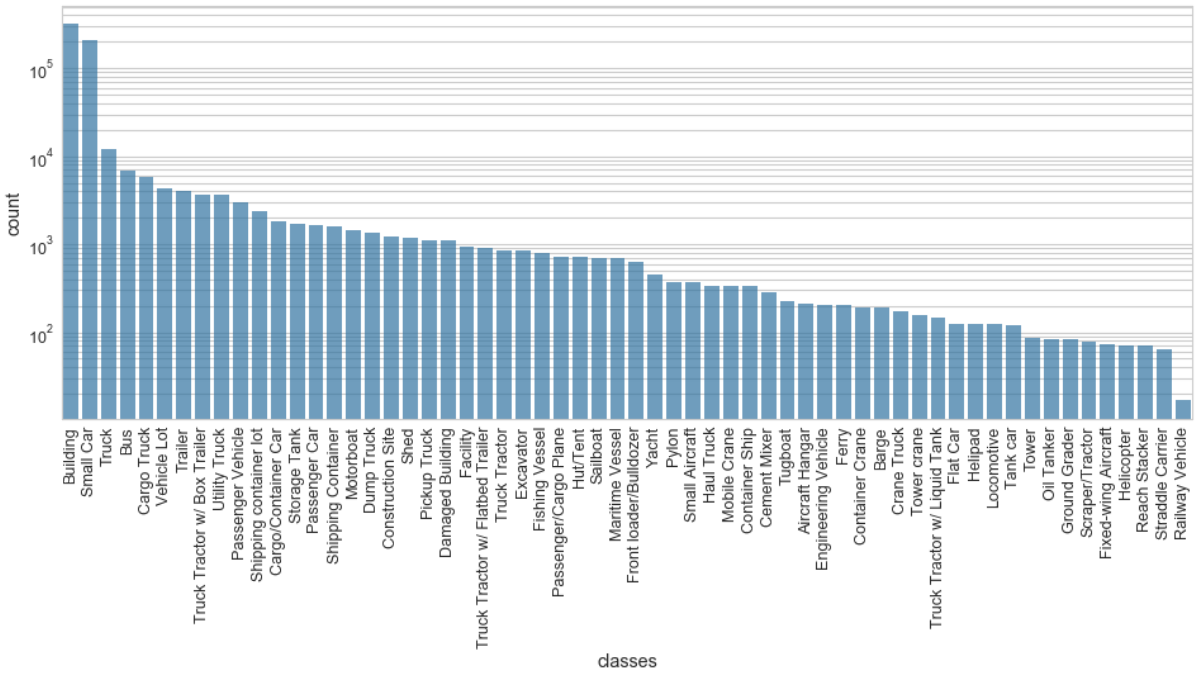

Figure 7 Count of bounding boxes per child class

We can now look at the same distribution for the Parent Alternative classification:

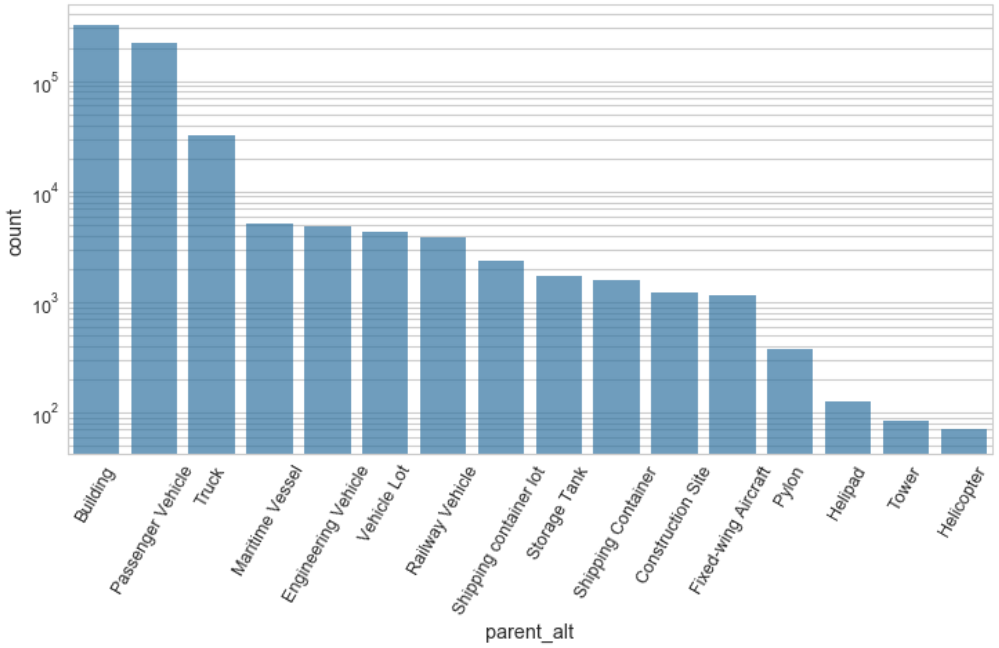

Figure 8 Count of bounding boxes per Parent Alternative class

In this case we go from 71 occurrences for Helicopter to 320811 for Building. The ratio between the lowest and the highest is slightly smaller than in the previous case. There is still two classes with less than 100 occurrences.

The fact that there are so few occurrences for some objects is quite problematic: it will be hard for the model to recognize those objects, and also it will be hard properly splitting the data so the classes have the same distributions across the splits than in the original dataset.

For this reason, algorithms to properly perform the split were studied. In the *Algorithms* section the solution chosen to deal with this problem is provided.

Below, the images (note that we talk about images here and not bounding boxes as in Figure 7) in which a given child class appears is plotted. This figure is plotted because the train-test split has to be performed on images, it cannot be performed efficiently over bounding boxes.

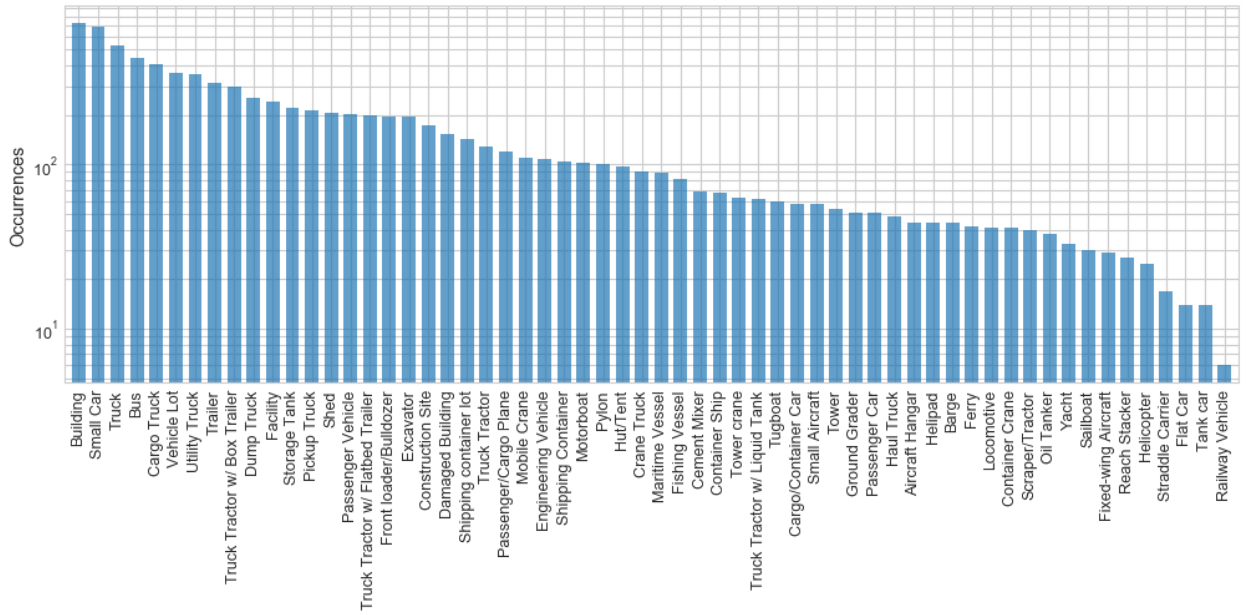

Figure 9 Number of images for a given child class

The number of images on which Railway Vehicle appears is extremely low, only 6. It shall be noted that two splits need to be performed over the data. One to select a subset of data that can be handled by the available resources. And another one, performed over this subset of data, to provide the train-evaluation-test subsets. Having classes that appear in such low numbers might translate in not having any occurrences in some of the split sets. In the case of the parent alternative classes, the class which appears in less images is Helicopter, which appears in 25 images (image not plotted).

Finally, in line with the previous graphs, we can look at the number of bounding boxes per image. As shown below, it goes from 1 to 7600. This is important as the algorithm should be able to work with cluttered images as these ones. This is also what makes the split between sets so complicated. For one image, i.e. one row of data we can have up to 7000 objects (in different classes the count goes up to 33 classes for one image). This kind of problem is referred to as multi-label classification problem (as one “row” of data has several labels associated).

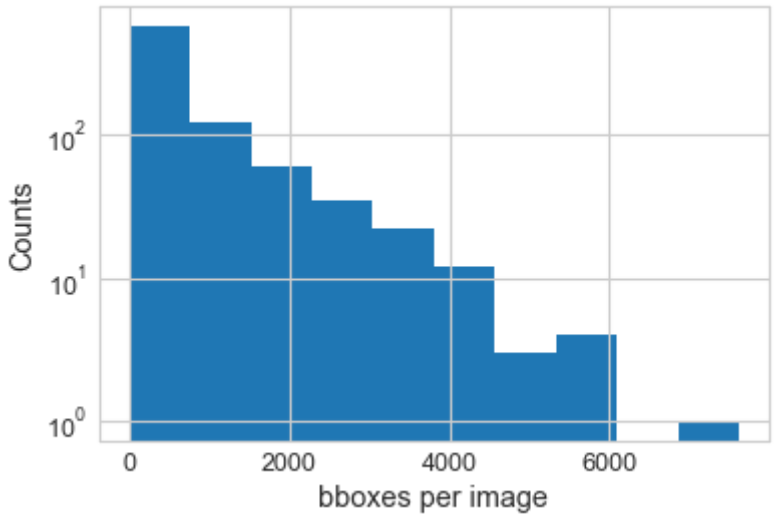

Figure 10 Distribution of image counts with the number of bounding boxes inside the image

Analysis of bounding boxes sizes

Following, we look at the bounding boxes distribution of sizes for each of the child classes (ordered by median value, note the vertical logarithmic scale):

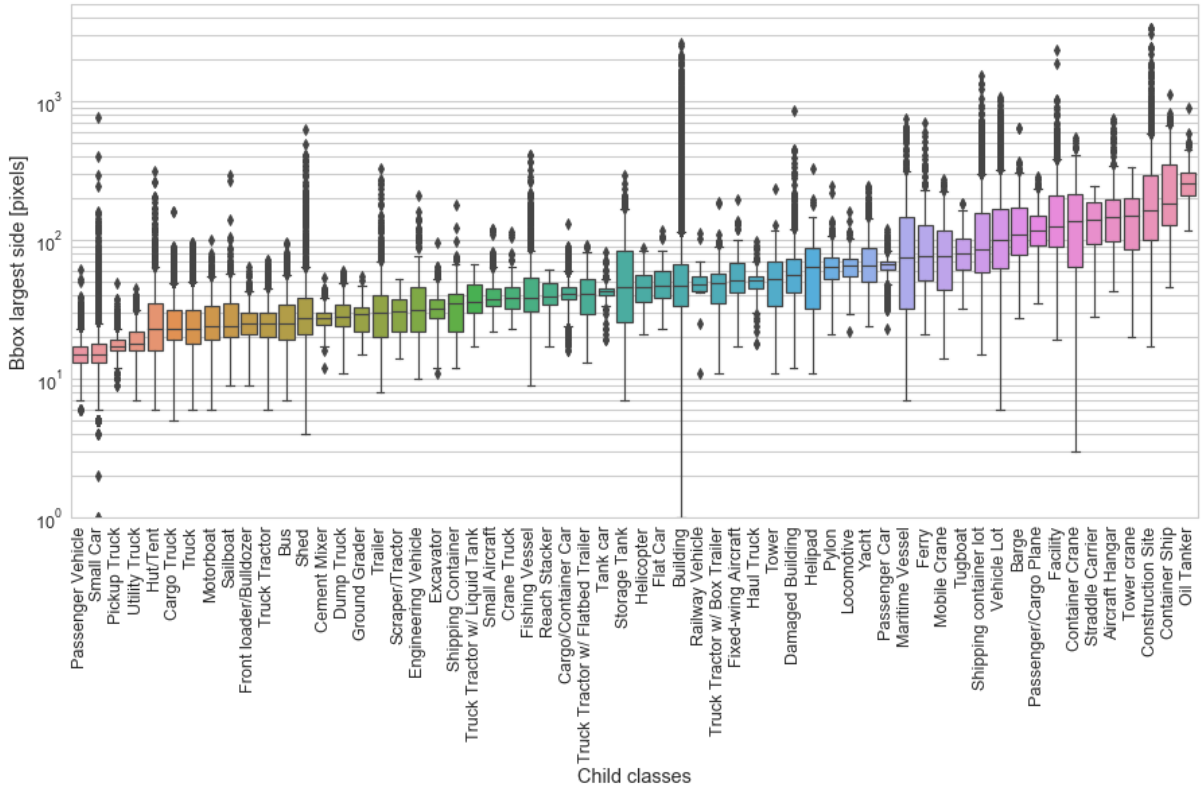

Figure 11 Distribution of bounding box largest side with the child classes

The main observation from this graph is that for some classes, the range of sizes within the class is huge: classes like Building go from a few pixels to around a 1000 for some occurrences. The algorithm that performs the object detection should ideally be able to deal with this.

Conclusions of the EDA

- There is a huge class imbalance, both for child classes and parent classes. Also, there are some classes that appear a extremely low number of times, some of them appear only in 6 images.
- There are large differences between the objects, with sizes that can vary from 10px to 2000px, all classes comprised. For objects of the same class there is still large differences.

Both problems are encountered no matter the classification system chosen, using the child or the parent alternative classes. Some of the child classes appear in very little images (as low as 6), which can difficult the split of the dataset and obtaining acceptable scores for that particular class. For these reasons, it has been preferred to **work with the Parent Alternative** classification system for this project.

As there are classes that are typically too small, it has been preferred to crop the raw images rather than reducing them in size. Reducing the images in size could have resulted in making some objects disappear, as the required reduction coefficient would be from 5 to 10 (going from 2000 - 5000 pixels to maximum 600 pixels commonly accepted by object detection algorithms).

The decided size to crop the images (from hereon referred to as chip size) is of 224. For all of the parent alternative classes but 1, more than 75% of bounding boxes is below this size. This is depicted in the figure below. This chip size would imply that a given object is only split in two pieces, except for those larger than 200 pixels. Construction Site class will have more objects split into more than 2 pieces. We will see if their performance is significantly worse.

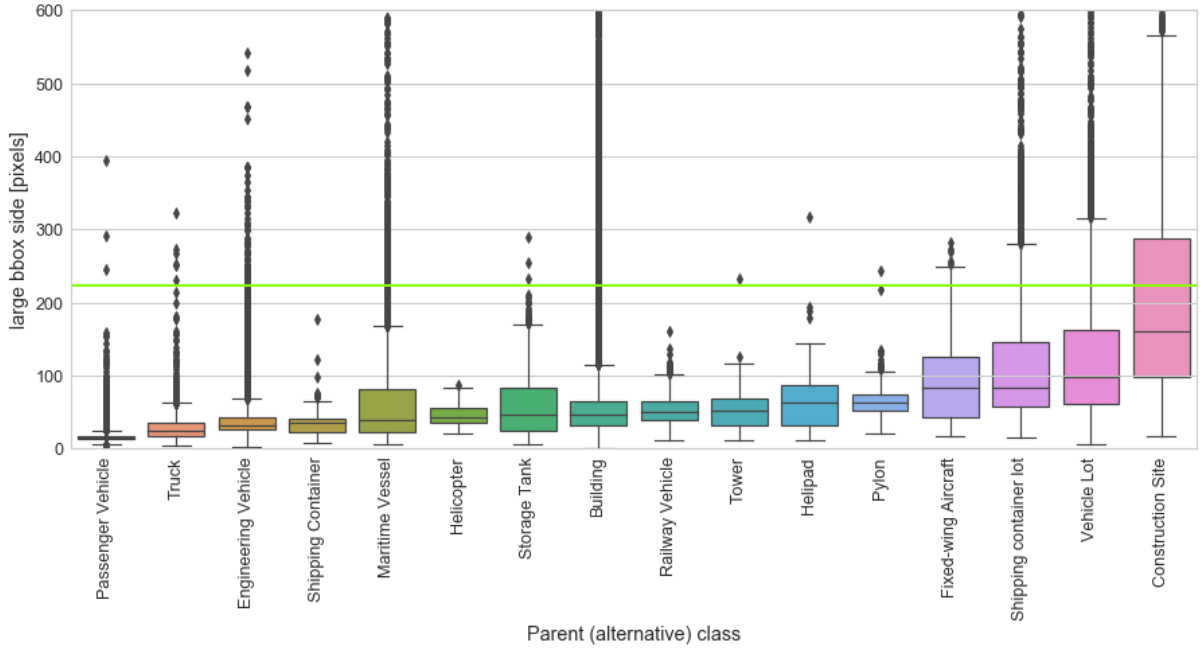

Figure 12 Distribution of bounding box largest side with the parent alternative classes

Algorithms

Two algorithms have been necessary to carry out this project. YOLOv5, to detect the objects, and iterative train test split to obtain splits of the datasets that maintain the distribution of labels across the sets better than a random train-test split.

YOLO

The main reason to choose YOLOv5 for this project is that in the tests performed during the project, the algorithm worked straight out of the box once the data was provided as necessary. The SSD from De Groot (repository), which applicability was studied as well for this project, could not be put to work quickly. YOLOv5 is also fast and it reaches quite acceptable mAP values on the COCO dataset, as reported in the repository. Plus, Ayush *et al.* use YOLOv3 for the same xView dataset that we are dealing with here.

The YOLO (You Only Look Once) family of algorithms was introduced by Joseph Redmond in 2016. The idea of this object detection algorithms is to perform detection of objects and classification of an object in a single pass (or as these algorithms are called also, in One Shot) over the image, instead of in two passes as it was performed previously.

Fong (2018) provides a succinct and quite clear explanation of the working principle of YOLO, and it is provided following: “YOLO takes an initial input image of a certain size (say 416 x 416) and then divides it up into a grid by downsampling the image by half, 5 times, resulting in a grid of 13x13. We call this the proposal grid because for each grid cell, we have some number of boxes (the paper uses 5 for the VOC dataset) of predetermined size centered on grid cell. The boxes serve as initial proposals for the locations and sizes of objects of interest in the image. What the YOLO network then does is to transform these boxes into ones that actually wrap around these objects of interest and classifies them. There are usually many redundant overlapping boxes which are then removed using post processing (more specifically, non maximum suppression).” The full pipeline is depicted below:

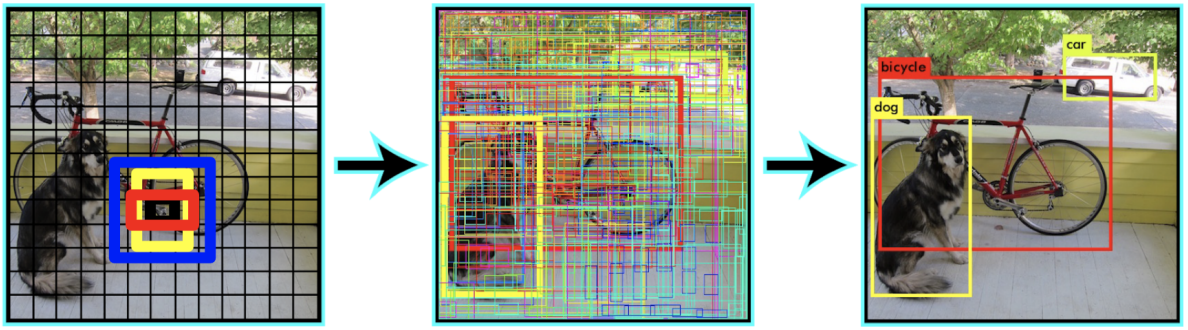

Figure 13 *(From Fong 2018)*. Image divided into 13x13 cells with a number of anchors proposed for one of the cells (left). The network transforms these anchors to boxes that wrap them around objects (center). Redundant and low probability proposals are removed (right).

Many improvements have been introduced since the first YOLO version and the author of this project is a beginner in this field to just understand them, let alone explain them. A good explanation of the versions up to YOLOv3 is provided by Mansoury 2019. Additionally, no paper has been published yet in YOLOv5 and to properly understand the way it works would

require going deeply into the code. To name a few of those improvements as examples: the anchors that are proposed for a given cell are optimized previously by a dimensional *k-means* clustering with a Genetic algorithm. At the time this project was started, it was possible to use images to 640 pixels by side, but this has gone up to 1280 pixels. From YOLOv3, feature pyramid networks are used, which allow to properly predict the same object with different sizes, even if not all sizes have been seen during training. This latter feature is quite useful for our data, given the spread in sizes shown by the classes (see Figure 12).

Several models are provided in the YOLOv5 repository (Jocher 2021), the table below, from the repository, summarizes them:

Model	**size** (pixels)	**mAP**^{val} 0.5:0.95	**mAP**^{test} 0.5:0.95	**mAP**^{val} 0.5	**Speed** V100 (ms)		**params** (M)	**FLOPS** 640 (B)
YOLOv5s	640	36.7	36.7	55.4	**2.0**		7.3	17.0
YOLOv5m	640	44.5	44.5	63.1	2.7		21.4	51.3
YOLOv5l	640	48.2	48.2	66.9	3.8		47.0	115.4
YOLOv5x	640	**50.4**	**50.4**	**68.8**	6.1		87.7	218.8

It shall be noted that this project uses the models from release v4.0 from January 2021. For this project, there is no need for real-time or quick predictions, so we can go with one of the heavier YOLOv5 versions with better scores. However, given the limited resources for training the **YOLOv5l** has been preferred over YOLOv5x to be able to train faster.

Iterative train test split

As mentioned in the analysis of the data, we are dealing with a multi label problem. Contrarily for example to an image classification problem where only one class is present in an image, in object detection we might encounter several classes in the same image, and each class might appear several times.

In multi-label data, groups could be formed based on the different combinations of labels (labelsets) that characterize the training examples. And then one possible solution to perform the split is to perform Stratified K fold based on the combination of labels. The problem for this particular dataset is that there are way too many combinations to produce a successful split. The table below provides a very simple dataset example with labelsets with just 4 possible labels for the data. In the table example below, with only 4 classes, we obtain 5 labelsets.

	Label				
Image	A	B	C	D	Labelset
1	1	0	0	0	1
2	1	1	0	0	2
3	1	0	1	0	3
4	0	1	1	0	4
5	0	0	1	1	5
6	1	1	0	0	2

The actual maximal combination of labels is of *min*(2^q, m), where *q* is the number of labels and *m* is the number of examples in the dataset (Sechidis *et al.* 2011). So typically the maximal combination of labels equals *m*. That is considering that a label is either in an image or not, regardless of the number of times it appears for an image. In our case, for the two classification tasks considered (either working with child classes or with parent alternative classes) we would have:

Classification	labels	images	combinations
Child class	60	847	1,15E+18
Parent alternative	16	847	65536

In both cases we will most likely end up with a number of labelsets equal to the number of images, making the split by labelset unfeasible. For this reason, other ways to perform the split and keep a proportion of classes which are close to the original proportion have been explored.

The chosen solution has been to use Iterative Stratification, as introduced by Sechidis *et al.* 2011. Roughly, iterative stratification works this way:

1. The desired number of examples of each label for each subset is calculated.
2. The label with the fewest remaining examples in the unassigned data is selected.
3. For each example of this label, the algorithm selects an appropriate subset for distribution. To assign the subset, priority is given to the subset that currently needs more examples of this label. If there are several subsets in this situation, the one which needs more examples all labels counted is selected. If still there are ties, a subset is chosen randomly.
4. The example (row of data) is added to the subset. The desired number of examples for all labels for this subset is updated and the process keeps going.

The algorithm finishes when the dataset is empty. The authors of this split algorithm report good performance measures of the splits performed with this method, and thus it has been chosen as improvement over a random split. In any case, below we briefly look at the results obtained when we splitting in this way. The iterative train test split used for this project is taken from the package *scikit multilearn*.

Performance Analysis

To assess the convenience of the iterative train test split, the frequency of occurrence of a given class in an subset of data compared to the frequency of that class in all the dataset was studied. In the case of performing a perfect split, if our dataset contains just two classes, for example Buildings and Cars, appearing in 63% and 37% of the bounding boxes respectively for the whole data, a perfect split would achieve the same proportions in the subsets of the data. Thus, dividing the classes frequencies in the subsets by the original classes frequencies would yield 1 for all classes. Below we plot this ratio (frequency of a class in a given split / frequency of that class in full set of data), for all parent alternative classes:

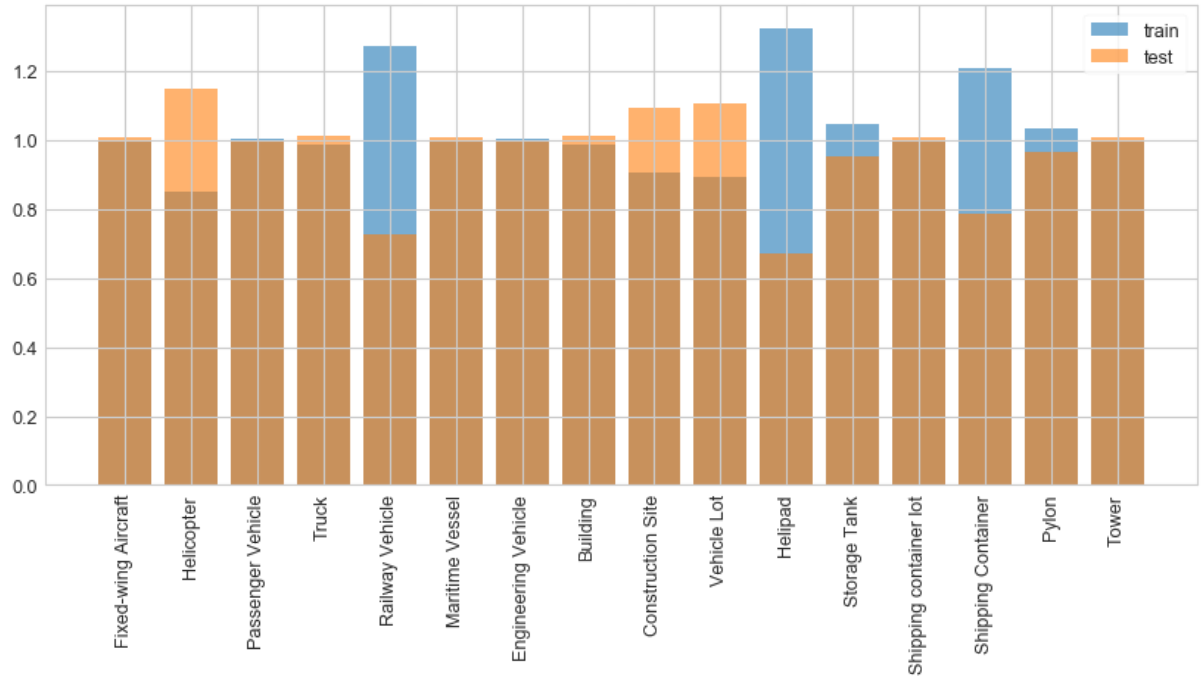

Figure 14 Ratio of the frequency of a class in a subset of data divided by the frequency of that class in the full dataset.

As can be observed, there are some classes (Fixed wing Aircraft for example) for which a near perfect split is obtained, with both the training and the test set having a very similar proportion of a class to the one in the full dataset. For other classes, Helipad for example, this is not the case.

The algorithm allows choosing if the split is performed considering binary information on the classes (i.e. just if a class is present in an image or not), or integrating a larger order into the problem, taking into account the total occurrences of a class instead of just their presence or not in the data.

To assess the quality of the split, both the average and the standard deviation of the ratio of frequencies for all classes are calculated. A perfect split would have average 1 and standard deviation of 0, so values closer to those are better. The results are provided below, comparing the performance of a random split with the iterative train test split.

	Presence or not in images		Total occurrences	
Split method	Average	Standard	Average	Standard
Random train test split	1.078	0.18	1.136	0.37
Iterative train test split	1.015	0.10	0.88	0.23

Iterative train test split shows better results also for our particular case. Ideally many tests would need to be performed to validate these conclusions, but the paper of (Sechidis *et al.* 2011) already did that. This analysis was done as validation of the algorithm and to tune the hyperparameters of the iterative train test split algorithm to obtain an optimal split.

Methodology

Data preprocessing

Based on the data analysis, the following cleaning steps have been taken:

1. The bounding boxes with a label that has no identification / mapping have been removed. There were 79 bounding boxes representing 0.01% of the total.
2. The bounding boxes that had a 0 length side were removed. There were only 9
3. The bounding boxes that were completely outside of an image have been removed. There were 23 of them. To be able to check for this, the image sizes had to be extracted from the images themselves. To obtain image sizes quickly, the bash command file was used. The idea was to avoid having to load images in python.
4. The bounding boxes that had one of the corners (but not both) out of the image have been cropped so all bounding boxes are completely inside the image.
5. Transforming data to its correct type. Basically we had to transform some columns from float numbers to integers. Before it was checked that no decimals were there. This could indicate some type of erroneous data, notably for the class labels.

On top of this cleaning efforts, some features were added. Two additional columns were added to the data: the parent and the parent alternative labels. And some columns to assess the sizes and the shape: diagonal, longest side, aspect ratio of the bounding box.

Implementation

Technical means

Two major problems have been encountered for implementing this project:

- The size of 24Gb of the dataset once decompressed.
- The fact that a powerful GPU is required to train the object detection algorithm.

Different solutions were explored to more or less extent: Udacity workspace, Kaggle notebook, personal laptop and using the AWS account. Below is a synthesis from these tests:

- **Kaggle notebook**: it proved useful for the possibility to have all the data accessible from the Notebook (by uploading it as a dataset). GPUs are available for training with a daily limit. The problem however came with the fact that the notebooks are fungible, and that no easy way was found to download/upload local data created during the notebook execution, for example the weights of the trained model, or the predicted labels.
- **Udacity workspace** was not of much use given the limitations in hard drive space. It did not allow loading even a reasonable portion of the data.
- **Google Colab**: it is also a fungible notebook but in this case the dataset has to be hosted elsewhere and loaded each time the notebook session is restarted. It provides free use of GPU for a few hours per day.
- **AWS** account with Udacity: following advice provided in the student forum, this solution to train the model was discarded as the credits would burn quickly. However, it proved quite useful for storing the data in a S3 bucket. It was possible to connect to this bucket to download the data from Google Colab by using AWS client.

At the end, the solution chosen was to keep the dataset in a S3 bucket and to use Google Colab for the GPU calculation. The dataset was also uploaded to Kaggle as a backup in case of expiration of AWS credits, and it might be made available publicly. In order to work with the data in Google Colab, at the beginning it was downloaded from S3 every time. However, for the final runs it was saved from S3 to Google Drive which can be integrated in Google Colab as a local drive. The data does not need to be downloaded every time, but Google Drive needs to be mounted to Google Colab everytime.

The exploratory data analysis and the split of the dataset were performed in the personal computer, to reduce as maximum as possible the use of Cloud resources to keep them for training and testing the model.

For the first tests of the model, about 100 images of the total were used for training and testing. For the final version of the model, used to produce the results in this report, a 25% of the data in total has been used. Roughly half of it (109 images) were assigned to the training set and the other half (106 images) were assigned to the test set by the iterative train test split algorithm.

Coding

YOLOv5, the chosen algorithm to perform object detection, includes a Colab notebook which links to YOLOv5 repository and shows examples of the necessary functions: installation of the requirements, training, testing and performing inference. Additionally the *test* function provides several metrics, including the mAP used for this project.

The main issue with Google Colab notebook is that the dataset and all configuration files had to be downloaded each time the calculation session had to be restarted. A GPU session with Colab free account lasts around 3 hours. As a consequence, to reduce human errors and to optimize time, as much automation as possible has been sought to be able to get to training as quickly as possible.

YOLOv5 requires a configuration file specific to the dataset (and in our case, it is also dependent on the image size chosen to split the larger images), and if we want to resume training from a previous training session we also need to provide the last weights. For the first versions of the Notebook these files were loaded into the notebook manually, but for the final versions they have been generated or loaded automatically by the notebook.

Finally, the images and labels have to be prepared to be fed into the model. The image below, from YOLOv5 repository, shows the coordinate system that YOLOv5 employs: a bounding box is provided by its center and its width and height, instead of the two corners coordinates that are provided in the xView *.geojson* file; and YOLOv5 requires a fractional measure from 0 to 1 (measure relative to image sizes) instead of the measure in pixels that was provided with the data.

Figure 15 Coordinate system that is used by YOLOv5 (Darknet format)

The process below was followed:

- The images were split into chips of a defined chip size, and the used chip coordinates were stored.
- Out of all the bounding boxes of a given image, find those that intersect a given chip,
- Readjust the bounding box coordinates to those relative to the chip in pixels. Trim the bounding box if it surpasses the boundaries of the chip. Transform pixel corners coordinates to YOLO coordinates.
- Save the labels and the images in the appropriate folders and output formats for YOLOv5 training script to be able to find them and interpret them.

Refinement

For the first tests with YOLO:

- The child classes were used,
- Random train test split was used to obtain first a subset of the data (100 images) to work with and again to obtain the train and test subsets.

However, the test dataset did not contain samples of all classes. After some training, the mAP values that were attained were of ~0.05, and that without using all the classes for testing.

Because of this, other methods to split the data were investigated and the iterative train test split solution was implemented. The algorithm was used both to extract the 25% of data to work with, and at a second step to produce the train and test sets (with 50% of the data for each).

It was also decided to use the parent alternative classes, this should facilitate splitting the data, and augmented the likelihood that during training the model had the chance to be trained on all types of classes.

A final refinement step taken was to remove some images containing only background (that is images with no objects in them). Although all the original images contained objects, when images were split into smaller chips, this was no longer the case. It was verified that around 50% of the image chips at the selected size did not contain any objects. According to the author of YOLOv5, the optimal value is between 0 to 10%; feeding some background images helps to reduce False Positives. It was decided that 5% of images used for training would be background images. For testing, all background images were left.

Results

The model has been trained for 80 epochs. It shall be noted that commonly for this type of problem the model should be trained much more, but given the available resources this has been the maximum possible.

The following table shows the metrics calculated over the test data set:

Class	Images	Labels	P	R	mAP[0.5]	mAP[0.5-0.95]
all	19826	105203	0.372	0.237	0.215	0.0969
Fixed-wing Aircraft	19826	242	0.64	0.508	0.453	0.205
Helicopter	19826	5	0	0	0	0
Passenger Vehicle	19826	31311	0.722	0.645	0.636	0.239
Truck	19826	5722	0.428	0.29	0.25	0.118
Railway Vehicle	19826	513	0.741	0.43	0.473	0.242
Maritime Vessel	19826	1028	0.589	0.314	0.299	0.133
Engineering Vehicle	19826	696	0.241	0.283	0.17	0.0813
Building	19826	63316	0.647	0.497	0.495	0.218
Construction Site	19826	666	0.253	0.0447	0.0357	0.0157
Vehicle Lot	19826	1216	0.37	0.0609	0.0602	0.0186
Helipad	19826	10	0	0	0	0
Storage Tank	19826	170	0.541	0.224	0.239	0.108
Shipping container lot	19826	176	0.265	0.182	0.0862	0.0279
Shipping Container	19826	86	0.255	0.14	0.0798	0.0464
Pylon	19826	33	0.266	0.182	0.155	0.097
Tower	19826	13	0	0	0	0

A first remark is that there are some classes that obtain 0 for all their metrics scores. This is most likely due to insufficient training of the model over those classes. It can be observed that those classes are the ones with the lowest occurrence of labels over the test dataset, and looking back at Figure 8 it can be observed that those are the classes with the fewer counts. Usually it seems to appear that the larger the labels the larger the metric values, but there are some interesting cases: Fixed-Wing Aircraft for example obtains relatively large metrics despite not having many labels.

In general, for most classes except for Engineering Vehicle, Precision is larger than Recall. This means that the false positives FP are smaller than the FN, for most classes, i.e. there are much more objects that are not being found than objects or background that are incorrectly considered belonging to the class. This is especially remarkable for the classes: Construction Site and Vehicle Lot, with a ratio P/R of 5 or 6. The confusion matrix below can shed some light into this and other issues.

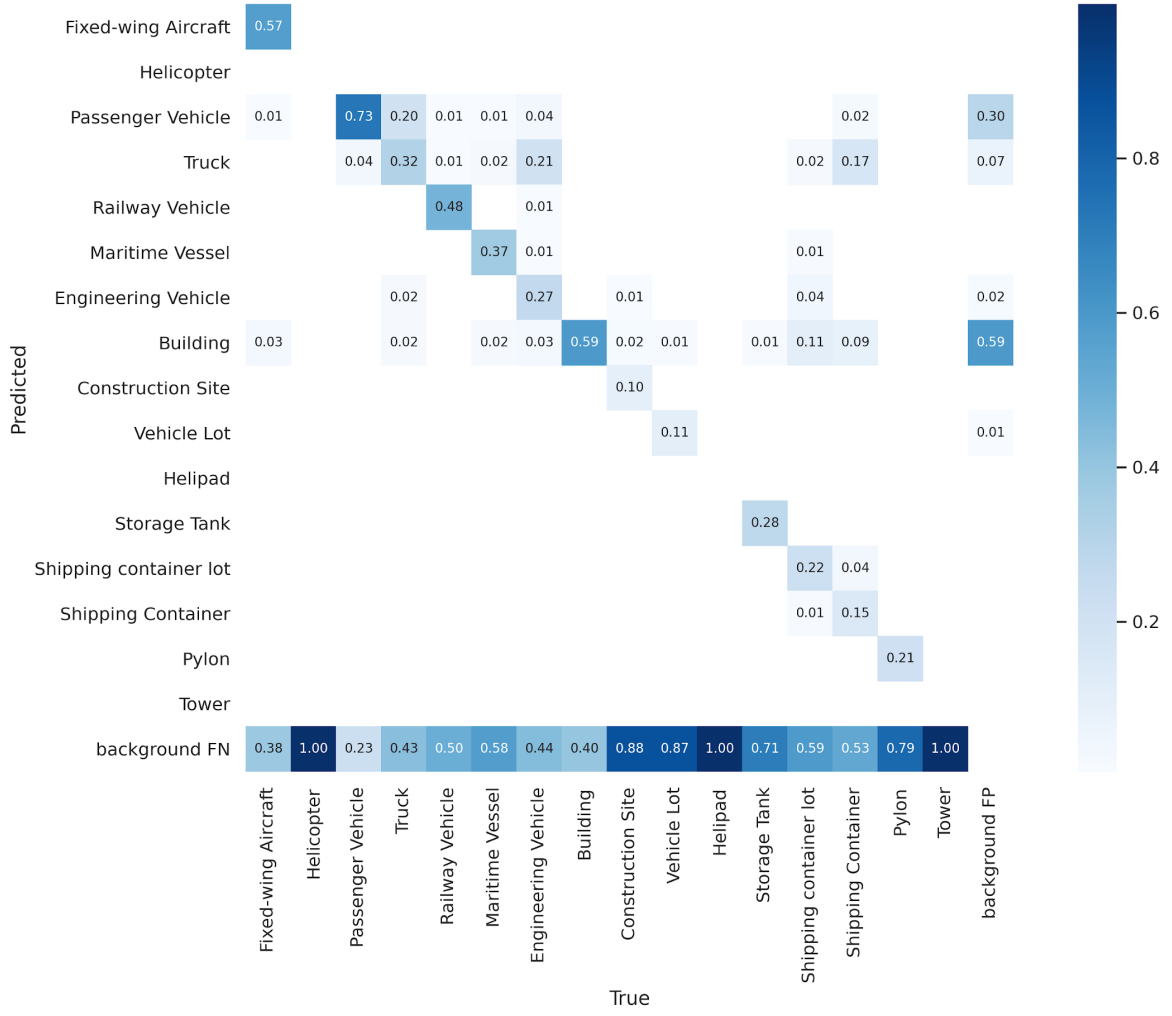

Figure 16 Confusion matrix: rows are predicted classes and columns are true classes

The confusion matrix contains an additional row, background FN, and an additional column, background FP. The first provides the objects that are not labelled in any way (i.e they are considered as background), and the column provides when background is considered as an object of a given class. For every column the sum is 1. The diagonal is the proportion of True Positives.

Looking at the confusion matrix, we can understand what happened to the classes that had 0 metric values, no object whatsoever was detected for those classes (background FN=1 For Helicopter, Helipad and Tower).

We can see some other interesting facts, for example many Trucks are being predicted as Passenger Vehicle, but not so much the other way around. And Shipping Container and Engineering Vehicle are predicted as Trucks; there are actually more Shipping Containers predicted as Trucks than as Shipping Containers. Building and Passenger Vehicle contain the largest amount of True Positives but also of False Positives. Also, Building and Passenger Vehicle seem to have the larger proportion of False Positives from other classes. It shall be noted that both these classes are the most common, so training more on these classes might have biased the algorithm.

Comparison to benchmarks

Compared to the results found in the literature (Fong 2018 or Lam *et al.* 2018) that use the child classes, this model provides a higher mAP (0.215) than the model of Fong but lower than Lam *et al.* when they used the multi-resolution approach. Again, in the case of Lam *et al.* 2018 they could train their model on the whole train data (for a week time), and here we use only 100 images (with around 8 hours total training time) and quite likely not all images have been seen by the model since we only completed 80 epochs.

Compared to the model of Ayush *et al.* (2020), which use a mix of Parent Classes and child classes (for a total of 10 classes, as opposed to here where we used the Parent Alternative classes for a total of 16 classes) we obtain a slightly lower score, 0.215 against 0.248. However, using the same 10 classes than Ayush *et al.* (2020) the mAP of this model would be 0.287, better than theirs.

Given the little time spent in training, the results seem quite acceptable. The improvements introduced by Jocher (2021) in YOLOv5 seem definitely to help in making predictions, at least for this problem in particular. Further training should allow obtaining better results in identifying objects, to then move onto the next steps for producing the poverty maps.

References

Papers

Ayush, K.; Uzkent, B.; Burke, M.; Lobell, D.; Ermon, S. (2020). Generating Interpretable Poverty Maps using Object Detection in Satellite Images. arXiv preprint arXiv:2002.01612

Biermann, L.; Clewley, D.; Martinez-Vicente, V. *et al.* (2020). Finding Plastic Patches in Coastal Waters using Optical Satellite Data. *Sci Rep* 10**,** art. no. 5364. https://doi.org/10.1038/s41598-020-62298-z

Henderson P.; Ferrari V. (2017). End-to-End Training of Object Class Detectors for Mean Average Precision. In: Lai SH., Lepetit V., Nishino K., Sato Y. (eds) Computer Vision – ACCV 2016. ACCV 2016. Lecture Notes in Computer Science, vol 10115. Springer, Cham. https://doi.org/10.1007/978-3-319-54193-8_13

Sechidis K.; Tsoumakas G.; Vlahavas I. (2011). On the Stratification of Multi-label Data. In: Gunopulos D., Hofmann T., Malerba D., Vazirgiannis M. (eds) Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2011. Lecture Notes in Computer Science, vol 6913, 145-158. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23808-6_10

Lam, D.; Kuzma, R.; McGee, K.; Dooley, S.; Laielli, M.; Klaric, M.; Bulatov, Y.; McCord, B. (2018). xView: Objects in Context in Overhead Imagery. arXiv preprint arXiv:1802.07856

Ren, S.; He, K.; Girshick, R.; Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. arXiv:1506.01497

Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91

Szymański, P.; Kajdanowicz, T. (2017). A Network Perspective on Stratification of Multi-Label Data. Proceedings of the First International Workshop on Learning with Imbalanced Domains: Theory and Applications, in *PMLR* 74, 22-35

Websites

Fong, R. (2018). The XView Dataset and Baseline Results. Medium article. Consulted 03/03/2021.
https://medium.com/picterra/the-xview-dataset-and-baseline-results-5ab4a1d0f47f

Hui, J. (2018). mAP (mean Average Precision) for Object Detection. Medium article. Consulted 03/03/2021.
https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173

Mansouri, I. (2019). Computer Vision Part 5: Object Detection, when Image Classification just doesn't cut it.
https://medium.com/@ilias_mansouri/part-5-object-detection-when-image-classification-just-doesnt-cut-it-b4072fb1a03d

The scikit-multilearn Team: scikit-multilearn API Reference.
http://scikit.ml/api/skmultilearn.html

Jocher, G., Stoken, A.; Borovec. J. et al. (2021). Ultralytics/yolov5: v4.0 - nn.SiLU() activations, Weights & Biases logging, PyTorch Hub integration.
https://zenodo.org/record/4418161#.YJf2xybtZDB

DIUx xView 2018 Detection Challenge. http://xviewdataset.org/

GitHub repositories

xView1 baseline: https://github.com/DIUx-xView/xView1_baseline

Glenn Jocher, YOLOv5: https://github.com/ultralytics/yolov5/tree/v4.0

Single Shot Detector implementation:
https://github.com/amdegroot/ssd.pytorch