Chapter 4

Research Methodology

This chapter lists and discusses the specific steps and activities that will be performed by the proponents to accomplish the project. The discussion covers the activities from pre-proposal to the final thesis writing.

The development of the study follows a 7-stage pipeline namely image collection, preparing data for crowdsourcing, crowdsourcing platform development, crowdsourcing, sidewalk width collection, accessibility score modeling and validation, and improving the pipeline of Atlas (See Figure 4.1). Each stage of the pipeline will be discussed thoroughly in the following sections.

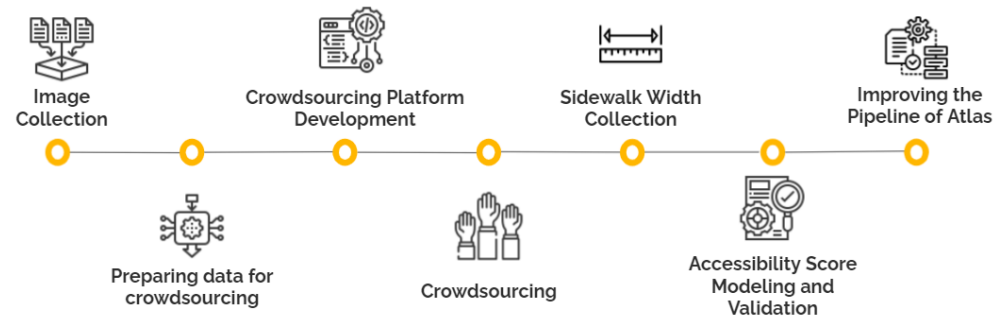


Figure 4.1: 7-stage project pipeline of the study

4.1 Image Collection

An initial collection of Google Street View (GSV) images is necessary to proceed with other stages of our study such as the running the YOLOv5 and OCR (ResNet-

101) model on the GSV images and preparing it for deployment to the website.

For this task, we utilized PostgreSQL, PostGIS, OpenStreetMap (OSM), and Google Street View to implement a semi-automatic method of collecting images. The following steps were taken during the research process for the Google Street View and satellite image data collection.

4.1.1 Retrieving the Road Network

Since Google Street View requires the longitude and the latitude of a specific location, we decided to utilize publicly available geospatial data to get the entire road network of Metro Manila. We were able to download the entire region of Metro Manila from Planet OSM[1] and loaded this into a PostgreSQL database using osm2pgsql[2]. The Planet OSM extract provides multiple tables but for the road networks, we focused on extracting the data from `planet-osm-line`. This specific table provides all the `LINESTRING` geometries, such as roads. A `LINESTRING` is composed of a series of longitude and latitude pairs and these values may be extracted from the database using the PostGIS functions provided.

However, the longitude and latitude pairs were too close to each other to make a difference calling the Google Street View API. In order to reduce redundant points, we rounded off the latitude and longitude pairs to the fourth decimal place and selected unique values based on the decimal degree precision values[3]. For the entire Metro Manila region, we were able to extract a total of 203,949 latitude and longitude pairs.

4.1.2 Google Street View API for Street-Level Imagery

After gathering list of coordinates, we were able to use the Google Street View API to get static images of roads. We created a script to loop through our coordinate list. For every coordinate, we collected four (4) images representing the north, south, east, and west perspectives. We then pitched the images slightly downwards to mimic an eye-level perspective. This totals to 815,796 static road images. Keep in mind, with our limitations, some of these images are unusable due to some factors such as Google Street View's privacy policy with regards to private roads. We also limited our study to collecting street view images only in

[1] `https://planet.openstreetmap.org/`
[2] `https://osm2pgsql.org/`
[3] `https://en.wikipedia.org/wiki/Decimal_degrees#Precision`

Makati and Manila. With that, we were able to collect 109,832 street view images and stored them in the Azure Database. Figure 4.2 is a sample Google Street View image that is stored in our Azure Database.

Figure 4.2: Sample Google Street View image of the road with the sidewalk present in frame

4.2 Preparing Data for Crowdsourcing

Before the Google Street View images are uploaded to Atlas for crowdsourcing, we first wanted to train our version of YOLOv5 to be able to detect objects that are situated on Philippine sidewalks. This is to reduce the need for our crowdworkers to locate obstructions and draw a bounding box when using Atlas, our crowdsourcing platform. We used YOLOv5 to detect these obstructions, so that our crowdworkers will have an easier task of only having to select which bounding boxes on the streetview image are the ones that block the sidewalk's pathway.

4.2.1 Detecting Objects on Sidewalks

With the streetview images we collected, we used YOLOv5 to detect objects found on the images. Since YOLOv5 is trained after the Common Objects in Contect (COCO) 2017 train/val/test dataset which contains a list of 91 objects as seen in

Appendix C, we expect it to detect objects that do not affect the accessibility of the sidewalk. Examples of these include the objects under the categories of:

- *animal*
- *accessory*
- *sports*
- *kitchen*
- *food*
- *furniture*
- *electronic*
- *indoor*

We utilized the existing COCO 2017 Dataset and filtered its classes to only detect objects located on sidewalks seen in Table 4.1. We also added to our list other objects that are commonly found on Philippine sidewalks. It is important to note that the objects we have added still need a number of instances for the YOLOv5 model to be able to detect them.

Objects from the COCO 2017 Dataset	**Additional Objects Added**
Bicycle	Tricycle
Car	Street Vendor Stand
Motorcycle	Street Sign
Traffic light	Cracked Pavement
Fire hydrant	Construction Materials
Stop sign	Utility Post
Parking meter	Lamp Post
Bench	Trees
	Curb Ramp

Table 4.1: List of classes for our YOLOv5 model to detect

4.2.2 Running YOLOv5 Inference on the Google Street View Images

We identified objects from the COCO 2017 Dataset that are relevant in affecting the accessibility of sidewalks. These objects include: *bicycles*, *cars*, *motorcycles*,

traffic light, *fire hydrant*, *stop sign*, *parking meter*, and *bench*. We cloned YOLOv5 from the main GitHub repository and it was able to detect these objects out of the box. In Figure 4.3, we can see that our current YOLOv5 model is able to detect cars, motorcycles, and a parking meter in the GSV image that we have collected. However, there are instances when the YOLOv5 model would mislabel some objects, such as the parking meter label in Figure 4.3 for it is actually security guard, not a parking meter. Since the YOLOv5 model is not entirely perfect in detecting the objects, we decided to download the COCO 2017 Dataset for re-training the YOLOv5 model. The COCO Train/Val dataset consists of 118,287 and 5,000 images respectively, while also containing the annotations for each specific image. We then filtered the annotations to only include: *bicycles*, *cars*, *motorcycles*, *traffic light*, *fire hydrant*, *stop sign*, *parking meter*, and *bench*. After filtering, we were left with 23,829 images for train, and 1,037 images for valid that contains the instances of the objects specified. In Figure 4.4, we can see a sample image from the COCO 2017 Dataset together with its corresponding .xml annotation file. The images from the COCO 2017 dataset consist of pictures taken from the internet and labeled according to the objects found in the image. In the figure, we can see two annotations for the two bikes seen in the image.

Figure 4.3: An example of the YOLOv5 model mislabeling a security guard as a parking meter

4.2.3 Crowdsourcing Manual Annotations to Train YOLOv5

Some of the objects that we included in our list, but are not part of the COCO 2017 dataset, still need instances for the YOLOv5 model to be able to recognize

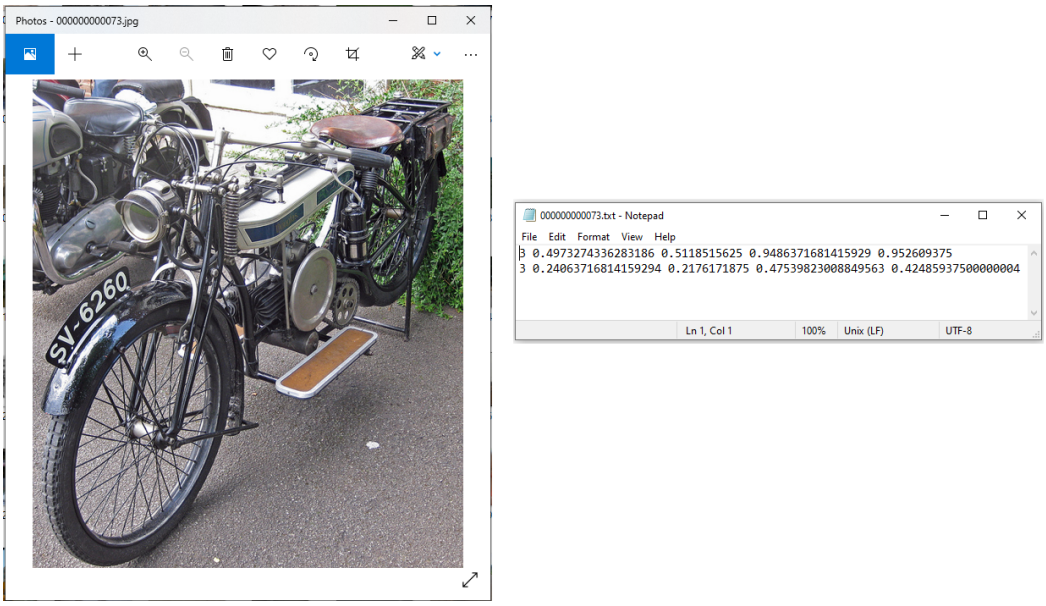


Figure 4.4: Sample image downloaded from the COCO 2017 Dataset with its corresponding .xml annotation file

them. As previously stated in Section 4.2.1, those objects are: *tricycle*, *street vendor stand*, *street sign*, *cracked pavement*, *construction materials*, *utility post*, *lamp post*, *trees*, and *curb ramps*. With that, we looked for volunteers who were willing to participate in the annotation process.

We created a user guide (See Appendix E), which informed users how to use Makesense.ai, a free online image labeling tool which could export labels as usable files for training YOLOv5. Within the manual, we included tips on how to properly label the objects, such as not overlapping the bounding box too much around the object, so that irrelevant pixels would not be included in the label. We then divided all the images we collected into sets with 70 images each. We chose 70 images so that each set will take approximately one hour to accomplish. Afterwards, we asked our volunteers to indicate how many sets they could annotate. By spreading awareness of our project, we were able to gather 14,120 manually annotated images with an original goal of 15,000.

We also took this opportunity to clean our dataset of images by asking the participants to submit a list of irrelevant images. Each participant was asked to note down of the filename of the specific irrelevant image, as we have images in our dataset that do not contain any sidewalks (e.g. images facing walls or located on highways). They compiled the filenames in a text file and included it in their submission. In total, we were able to identify 1,748 images among the submitted 14,120 images as irrelevant, and we removed these from our dataset leaving us

with 12,372 annotated images.

4.2.4 Computing Total Number of Instances per Class

We computed for the total number of instances we per class in order to see which objects have enough instances when training the YOLOv5 model. In Table 4.2, we can see the total number of instances from the COCO Dataset, while the total number that our crowdworkers have labeled can be seen on Table 4.3. According to Jocher (2021c) who is the main author and developer of YOLOv5, the recommended number of instances (labeled objects) per class should be more than 10,000 instances. Since we had limited time to manually look for images from the internet and label them according to the class object, it is a limitation of our study to have classes with less than 10,000 instances. This could affect the accuracy of our version of YOLOv5 to properly learn these objects during training. However, we still did our best to increase the number of instances for some of the classes, such as the street vendor stand which originally only had 892 instances. We obtained images from Google and annotated street vendor stands in order to increase number of instances to 1,291. With this, we hope that it would to some extent help our YOLOv5 model in learning how to detect the said object.

Class	**Total Number of Instances**
Bicycle	7,433
Car	45,883
Motorcycle	9,145
Traffic light	13,565
Fire hydrant	1,970
Stop sign	2,061
Parking meter	1,345
Bench	10,252

Table 4.2: Total Number of Instances Per Class from the COCO Dataset

4.2.5 Combining COCO 2017 Annotations and Manual Annotations

Our initial plan was to train our version of YOLOv5 to detect the list of objects on the COCO 2017 annotations and manual annotations before uploading the labeled images to Atlas for crowdsourcing. Unfortunately, we encountered resource

Class	Total Number of Instances
Tricycle	3,450
Street vendor stand	1,291
Street sign	3,346
Cracked pavement	2,609
Construction materials	1,136
Utility post	13,686
Lamp post	6,878
Tree	16,507
Curb ramp	1,130

Table 4.3: Total Number of Instances Per Class from Crowdworkers

problems earlier on when trying to train YOLOv5. Since we wanted to keep up with our timeline for the project, we instead decided to combine the 12,372 manual annotations from our volunteers and the annotations of the YOLOv5 trained on the COCO 2017 annotations in preparation for the crowdsourcing in Atlas. We combined them by running a Python script to create a new XML file if a specific streetview image contains both an annotation from COCO 2017 and the manual annotation. In Figure 4.5, we can see a sample labeled image after combining both the COCO 2017 annotations and manual annotations. All the labels of *car*, *traffic light*, and *fire hydrant* were from the COCO 2017, while the labels of *tree*, *lamp post*, and *curb ramp* were the manual annotations of our volunteers.

The only limitation of this output is that all the manual annotation labels were annotated by our volunteers, so there is no precise way to measure how accurate their annotations were. However, when training a computer vision model to detect objects, the labels are usually manually annotated by human beings. We also went over the images labeled by our volunteers to check for errors, but we saw that most of their annotations were correct. With the annotation manual (see Appendix E) that we provided, we believe that they were able to understand how to properly label the objects. We gave out specific instructions in the manual such as not overlapping the bounding box too much around the object, so that irrelevant pixels would not be included within the label.

4.3 Crowdsourcing Platform Development

Atlas is an online crowdsourcing platform we developed to facilitate the crowdsourcing of accessibility scores by presenting pre-labeled sidewalk images to vol-

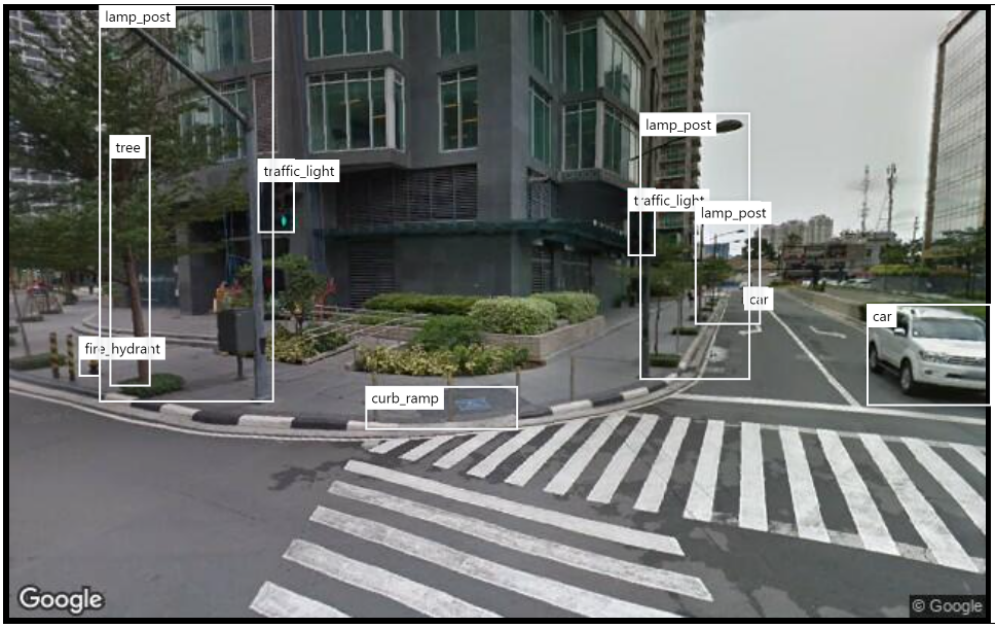


Figure 4.5: Sample labeled image using the combination of COCO 2017 and Manual Annotation as shown on Atlas

unteers. The output of Atlas will be a model that can predict an accessibility score of a given sidewalk image based from the crowdsourced accessibility scores. Atlas could also be used to create a dataset of accessibility scores that could be mapped out.

By developing Atlas, we were able to have a crowdsourcing platform wherein registered users could label, score, and further annotate sidewalks. We uploaded 12,372 preliminary streetview images during our data preparation to the crowdworkers. Atlas was created using Next.js as our primary web development framework. We also utilized the Tailwind CSS framework to design our web pages to fit the theme of our research project. The whole development process took around five months and it was deployed and hosted on our Vercel website, Vercel is an open serverless platform for static and hybrid applications. In Section 5.1, we discussed more completely the entire system architecture of Atlas.

4.4 Crowdsourcing on Atlas

The crowdsourcing platform was disseminated through online community forums, schools, and urban planning groups. The researchers aimed to incentivize crowdworkers by offering raffle draws and cash prizes to the users with the most number

of annotations. We disseminated this competition through a graphic posted in the same social media groups mentioned above, as seen in appendix F.

In registering to the platform, we asked for the users relevant demographic information such as: their city of residence, age, use of mobility aids, and commute frequency. They were requested to identify and label sidewalk obstructions, rate sidewalk accessibility, and determine the surface type of the sidewalk.

4.5 Sidewalk Width Collection

In addition to collecting sidewalk accessibility scores, annotations, and surface types, we collected the widths of the annotated sidewalks due to its effect on sidewalk accessibility. As stated in Sections 3.1 and 3.1.2, the width of the sidewalk is a feature that contributes to its overall accessibility. The collection of sidewalk widths was done through the use of the measure distance feature on Google Earth. As seen in Figure 4.6, the feature allows users to click and drag on a top-view satellite street view image and will output the width in meters. We used the latitude and longitude of the street view images that were annotated in order to locate it on Google Earth. By measuring the width of the sidewalks, we add another feature on sidewalk accessibility that will be used as a predictor for the model.

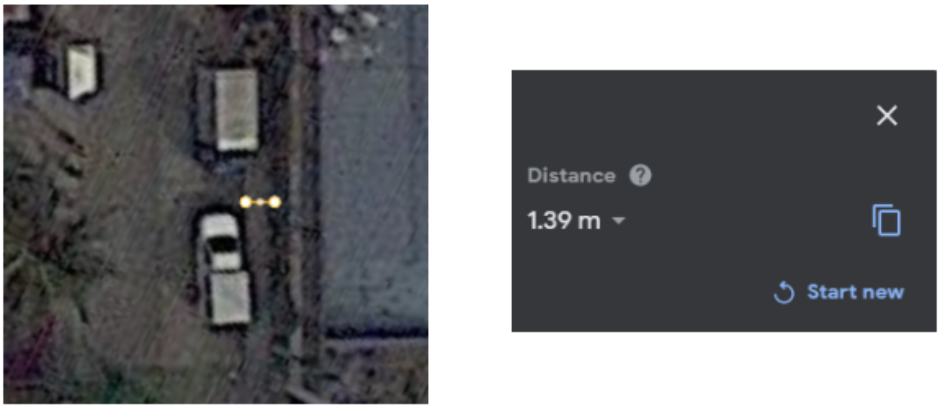


Figure 4.6: Measuring sidewalk width of a satellite image using the measure distance feature of Google Earth

4.6 Model Creation and Validation

After collecting accessibility scores, annotations, surface types, and sidewalk widths, we begin with modeling sidewalk accessibility. With the use of XGBoost, we created a regression model in order to see which features affect the accessibility of a sidewalk. The model was also used to predict the accessibility score of an entry given the features such as surface type, sidewalk width, and obstructions.

4.6.1 Pre-processing

Data pre-processing was done in order for us to be able to quantify our data for modeling and exploration. Initially, the annotation data we receive from our crowdworkers consists of the following columns: *id*, *date*, *username*, *imageID*, *accessibilityRating*, *pavementType*, *selectedObjectsID*, *newObjects*. In order to process this better, we needed to transform *pavementType*, *selectedObjectsID*, and *newObjects* in a more measurable data type. For the objects, we mapped the *selectedObjectsID* and *newObjects* column to their respective counts per class. As such, we generated the following columns: *objectType_total*, *objectType_obstruction*, *objecType_non-obstruction* for each of the 18 object types the crowdsourcing platform has. In addition, the *pavementType* column was converted to four numerical columns: *isRough*, *isSmooth*, *isSlippery*, *isNoSidewalk*. We would like to state that not all the user annotations were used due to our limited time and our manual process of acquiring the sidewalk width. Because of this, annotations without a corresponding sidewalk width were dropped. After this, we normalized the data by converting the counts of obstructions and non-obstructions into percentages by dividing them with the total counts of objects in an image. We also averaged the sidewalk widths of annotations on the same image in order to have a consistent width among images with multiple annotations.

4.6.2 Model Development

In developing our accessibility score model, there are two objectives: predicting the accessibility score of a sidewalk given several features and determining the features with the biggest impact on the prediction. To do this, we ran a regression analysis on the data through the use of XGBoost regressor. XGBoost is among the highest performing learning models due to its use of an collection of learning models to provide accurate results (Chen & Guestrin, 2016). For the predictors of the model seen in Table 4.4, we identified 40 features from our data. These are

the sidewalk width, three surface types, 18 sidewalk obstructions types, and 18 object types not tagged as obstructions.

Features			
width	parking_meter	curb_ramp	tricycle_obstruction
isRough	bench	others	street_vendor_obstruction
isSmooth	tricycle	bicycle_obstruction	street_sign_obstruction
isSlippery	street_vendor_stand	car_obstruction	cracked_pavement_obstruction
bicycle	street_sign	motorcycle_obstruction	construction_materials_obstruction
car	cracked_pavement	traffic_light_obstruction	utility_post_obstruction
motorycycle	construction_materials	fire_hydrant_obstruction	lamp_post_obstruction
traffic_light	utility_post	stop_sign_obstruction	tree_obstruction
fire_hydrant	lamp_post	parking_meter_obstruction	curb_ramp_obstruction
stop_sign	tree	bench_obstruction	others_obstruction

Table 4.4: Table of Predictors for the Sidewalk Accessibility Model

In order to better understand the impact of each feature, we use SHAP values from the SHAP package to interpret them. SHAP values represent the contributions of each value to the output of the sidewalk accessibility model. Through SHAP values, we can see which features have a positive and a negative effect on the output of the model. In this case, the output of the model would be the predicted accessibility score.

4.6.3 Training and Validation

In order to train and validate the model, we split the data into train and test sets with a percentage of 80% and 20% respectively. We then look at modelling with a classifier (XGBClassifier) or a regressor (Linear Regression, XGBRegressor) and choose the model that performs best based of their scores on the dataset, 5-fold cross validation score, and other peformance measures such as the accuracy and root mean squared error for the classifier and regressor respectively. We then tune the parameters of the models to increase its performance. In order to determine the best parameters to be tuned, we use `GridSearchCV`, a function from sklearn's model selection package, to fit multiple configurations of parameters in our model. By running multiple configurations of parameters, GridSearchCV gets to see which configurations performs the best on the train set. The configurations of parameters that were tuned for the classifier are the following: `learning_rate` , `max_depth` , `min_child_weight` , `n_estimators`. The configurations of parameters that were tuned for the regressor are the following: `colsample_bytree`, `learning_rate` , `max_depth` , `min_child_weight` , `n_estimators`, and `subsample`. These parameters were chosen due to their relevance on influencing the model's ability to generalize and reduce overfitting. We also use ridge as the regularization method for the models since we do not want to remove features when reducing the complexity of our model. After getting the best configurations of parameters for the XGBoost regressor and classifier, we ran the 5-fold cross validation with original default parameters and tuned parameters and compare their accuracy scores. Besides this, F1-score was used to measure the performance of the classifier models while the root mean squared error (RMSE) was used for the regression models.

4.7 Improving the Pipeline of Atlas

We initially collected a total of 109,832 street view images from the Google Street View API as mentioned in Section 4.1.2. However, we encountered resource problems earlier on when training YOLOv5 as mentioned in Section 4.2.5. With this,

we used the 12,372 images we collected from manual annotations and combined it with the annotations from the YOLOv5 trained on the COCO 2017 annotations. We proceeded with our crowdsourcing on Atlas by only using the 12,372 street view images. However, training YOLOv5 would be helpful so that we can utilize all the 109,832 street view images that we initially collected. This section documents the process of how we were eventually able to train the YOLOv5 model. We also used MMSegmentation to filter out irrelevant images, as we have images in our dataset that do not contain any sidewalks (e.g. images facing walls or located on highways). Afterwards, we uploaded these new images to Atlas.

4.7.1 Training of YOLOv5 with COCO 2017 Annotations and Manual Annotations

Training the YOLOv5 model would improve our study as we would be able to accurately measure the correctness of the annotations in the street view images. When training the model, we first shuffled the data to ensure that the all the different classes of the annotations get randomized among the train, test, and validation sets. We then partitioned the data by following a split ratio of 70% train, 15% valid, and 15% test. Our train data included 26,793 images, while our test and valid both contained 5,743 images. In Table 4.5, we also created a python script to compute for the total number of instances within the Train, Test, and Validation sets to ensure that the instances of the classes are more or less equally divided among the three sets according to their split ratio. Afterwards, we followed the guide instructions on YOLOv5's main GitHub repository on how to train on a custom dataset (Jocher, 2021a). Using the cloud server of College of Computer Studies (CCS) Jupyterhub, we were able to utilize their NVIDIA Tesla V100 graphics card for training the YOLOv5 model. We trained on 300 epochs and a batch size of 64. It was recommended by the author of YOLOv5 (Jocher, 2021c) that the training should start on 300 epochs, and the batch size to be set to the largest that our hardware allows for. We also used YOLOv5x, the largest and most accurate model of YOLOv5 for images with 640 pixels in size. All our 109,832 street view images were saved in a resolution of 640x400, hence we used the YOLOv5x model. After the training was completed, we saved the best weights of our model, and used it to run inference on our dataset of 109,832 street view images. Figure 4.7 is a sample output of the inference using the trained YOLOv5 model.

Class	Total Instances	Instances on Train Set	Instances on Test Set	Instances on Valid Set
Tricycle	3,450	2,416	540	494
Bicycle	7,433	5,115	1,152	1,166
Car	45,883	31,947	7,215	6,721
Motorcycle	9,145	6,495	1,257	1,393
Street vendor stand	1,291	861	189	241
Street sign	3,346	2,291	507	548
Cracked pavement	2,609	1,838	376	395
Construction Materials	1,136	785	165	186
Utility post	13,686	9,438	2,104	2,144
Traffic light	13,565	9,533	1,971	2,061
Fire hydrant	1,970	1,366	298	306
Stop sign	2,061	1,413	326	322
Parking meter	1,345	902	244	199
Bench	10,252	7,181	1,663	1,408
Lamp post	6,878	4,820	1.002	1,056
Tree	16,507	11,555	2,299	2,653
Curb Ramp	1,130	768	186	176

Table 4.5: Total instances of all classes within Train/Test/Valid sets for YOLOv5 Training

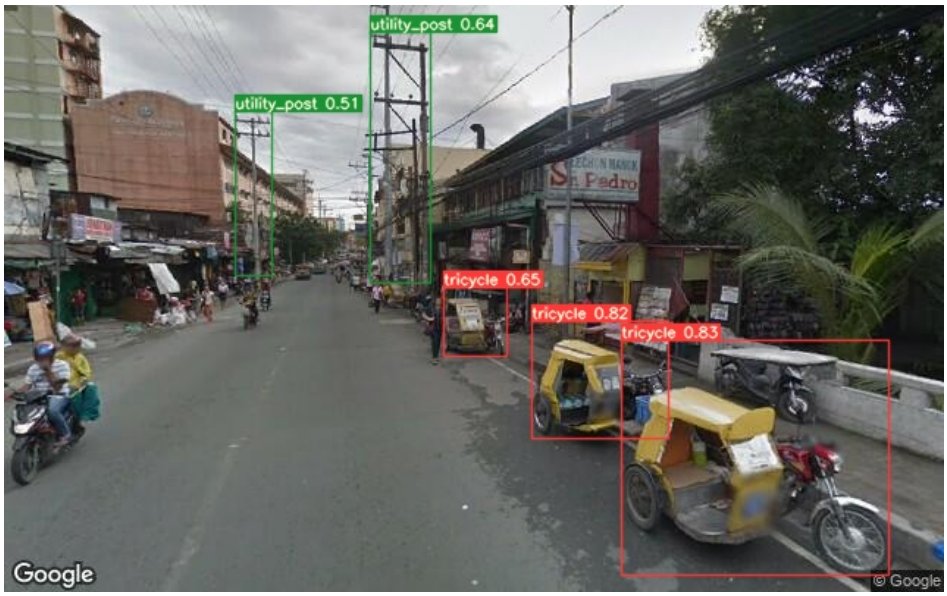


Figure 4.7: Sample result of the inference using the trained YOLOv5 model

4.7.2 Filtering Sidewalks Using Semantic Segmentation

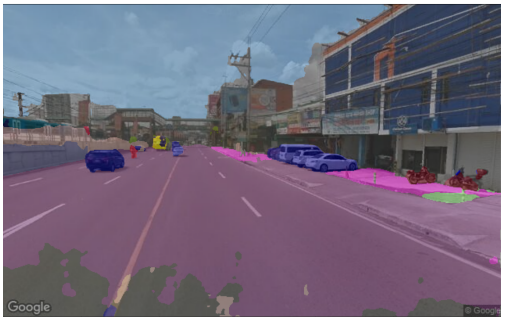
We also attempted to improve the results of our crowdsourcing phase by filtering sidewalks from images using semantic segmentation. Semantic segmentation is defined as the process of classifying each pixel belonging to a particular label (Matcha, 2020). In our research project, we aimed to classify and label sidewalks themselves in order to identify relevant objects in determining accessibility, i.e. objects found on the actual sidewalk. We used OCR (ResNet-101) semantic segmentation model through MMSegmentation, an open source semantic segmentation toolbox from OpenMMLab based on the PyTorch machine learning framework.

In total, we ran inference on 109,832 Google Streetview images which we previously collected. Figure 4.8 compares three images of sidewalks which we ran inference on. In Figure 4.8(a) you can see that the entirety of the sidewalk is categorized properly and segmented with pink tint. This is most likely due to the lack of interference from other objects in the image and the whole sidewalk being under the same type of lighting. Figure 4.8(b) on the other hand, shows an image without a sidewalk but had regions segmented as sidewalk through the pink tint. The mislabeled sidewalk is a part of the road that has a shadow casted on it which made the region look like it was different from the road. Looking at Figure 4.8(c), we see an inaccurate result through the incomplete segmentation of a sidewalk. Only the upper part of the sidewalk was labeled as sidewalk and the lower part was labeled as part of the road. The inaccuracy of the model led us to discontinue our

plan to apply it on images during the crowdsourcing phase. Although the group considered morphological transformation methods such as dilation and erosion to complete the sidewalk area created by semantic segmentation, it would be difficult to handle the cases of mislabeled sidewalks as seen on Figure 4.8(b). Because of this, we did not push through with using these images for crowdsourcing since defining a generalized threshold would not guarantee accurate results for images with mislabeled sidewalks. In order to replicate the effect of semantic segmentation, we introduced a feature on Atlas where users can select a *"No sidewalk"* option on images with no sidewalks. This will help us exclude these images and annotations from our model.

(a) Accurate and Complete Sidewalk

(b) Mislabeled Sidewalk



(c) Incomplete Sidewalk

Figure 4.8: Image quality and color dynamics affect the accuracy of OpenMMLab's semantic segmentation toolbox