BRAILS

Release 2.0

Charles Wang

May 26, 2021

USER MANUAL

CHAPTER

ONE

WHAT IS BRAILS

The SimCenter tool - Building Recognition using AI at Large-Scale (BRAILS) is an AI-enabled software to assist regional-scale simulations. BRAILS is a prototype development that utilizes machine learning (ML), deep learning (DL), and computer vision (CV) to extract information from satellite and street view images for being used in computational modeling and risk assessment of the built environment. It also provides the architecture, engineering, and construction professionals the insight and tools to more efficiently plan, design, construct, and manage buildings and infrastructure systems.

The released v2.0 is re-structured with modules for performing specific analyses of images. The expanded module library enables BRAILS' capability of predicting a broader spectrum of building attributes including occupancy class, roof type, foundation elevation, year built, soft-story.

The new release also features a streamlined workflow, CityBuilder, for automatic creation of regional-scale building inventories by fusing multiple sources of data, including OpenStreetMap, Microsoft Footprint Data, Google Maps, and extracting information from them using the modules.

Examples of BRAILS' application in natural hazard engineering include: The identification of roof shapes, occupancy type, number of stories, construction year, and foundation elevation to improve the damage and loss calculations for the hurricane workflow; The identification of soft-story buildings to improve models in earthquake workflows.

1.1 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1612843. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

1.2 About

This is an open-source research application, BRAILS Github page, released under a **BSD clause 2** license, Section 1.8. *BRAILS* focuses on harvesting and analyzing regional building information based on data from multiple sources to assist decision makings in various sections, such as urban plan, risk management, etc.

In SimCenter, one of the successful applications of *BRAILS* is in assisting the assessment of natural hazards and their imparts on the built environment. The increase in the intensity of natural hazards is magnifying the impact of such events on our society. In order to quantify and mitigate the risk due to the hazards and to prepare for the potential impacts in a region, it is necessary to analyze existing buildings that are pertinent to natural hazard analysis and risk management. However, gathering the building information in a region- or city-scale and analyzing them is a laborious and expensive undertaking.

BRAILS provides a framework for building information generation/gathering to support regional hazard analysis. In this framework, different types of data are acquired from multiple sources and are fused to semantically profile each

building in a region. Specifically, deep learning technique is employed to analyze street or satellite images. Pretrained convolutional neural networks shipped with *BRAILS* are capable of analyzing images and detecting building properties that indicate vulnerabilities to natural hazards. A novel data mining tool is integrated to overcome the data scarcity issue, quantify the uncertainty and enrich the data repository. With this framework, building inventories of cities can be analyzed and the results provide the insights for further disaster and risk management planning and simulations.

1.3 Installation

BRAILS is developed in Python3. You can install BRAILS using pip3 (you can use pip is your default pip is pip3):

```
pip3 install -U BRAILS
```

Windows users may experience difficulties because of two dependencies required: GDAL and Fiona.

If you have difficulties installing BRAILS using the above command, please check the *Troubleshooting* section.

1.4 User Guide

BRAILS provides a streamlined workflow for creating building inventories.

The workflow calls functions from multiple modules, which are capable of extracting information from images.

Most modules can be used as standalone functions.

1.4.1 1.x User Guide (Legacy)

Prepare and fuse data

Get google map api key

Obtain the API key from https://developers.google.com/maps/documentation/embed/get-api-key.

Define api key in src/keys.py file like this:

```
GoogleMapAPIKey = "replace this with your key"
```

Prepare a list of building addresses in csv format

For example, Atlantic_Cities_Addrs.csv looks like Fig. 1.4.1.

Noticing that, in addition to addresses, there are extra columns containing basic building information (e.g., stories, occupancy, etc.). These basic information are scraped from tax websites. It should be noted that, for a portion of these buildings, some building information may be missing from the websites. In this case, just leave them blank in csv file.

Once Atlantic_Cities_Addrs.csv is prepared, define the path of it in src/confiugre.py like this:

```
cleanedBIMFileName = dataDir+"/Atlantic_Cities_Addrs.csv"
```

address	stories	yearBuilt	occupancy	structureType	buildingDescription	city
13 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1965	Residential	Frame	1SF	MARGATE CITY CITY
11 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1969	Residential	Frame	1SF1G	MARGATE CITY CITY
9 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1969	Residential	Frame	1SF	MARGATE CITY CITY
7 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1965	Residential	Frame	1SF	MARGATE CITY CITY
5 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1966	Residential	Frame	1SF	MARGATE CITY CITY
3 E GILMAR CIRCLE, MARGATE CITY,NJ	1	1965	Residential	Frame	1SF	MARGATE CITY CITY
1 E GILMAR CIRCLE, MARGATE CITY,NJ	2	1965	Residential	Frame	2SF	MARGATE CITY CITY
8093 FULTON AVE, MARGATE CITY,NJ	1.5	1957	Residential	Frame	1.5SF2G	MARGATE CITY CITY
8095 FULTON AVE, MARGATE CITY,NJ	2	1962	Residential	Frame	2SF3G	MARGATE CITY CITY
8097 FULTON AVE, MARGATE CITY,NJ	2	1959	Residential	Frame	2SF	MARGATE CITY CITY
8099 FULTON AVE, MARGATE CITY,NJ	1.5	1964	Residential	Frame	1.5SF	MARGATE CITY CITY
8101 FULTON AVE, MARGATE CITY,NJ	2	1959	Residential	Frame	2SF3G	MARGATE CITY CITY
8103 FULTON AVE, MARGATE CITY,NJ	1	1957	Residential	Frame	1SF1G	MARGATE CITY CITY
8105 FULTON AVE, MARGATE CITY,NJ	2	1952	Residential	Frame	2SF	MARGATE CITY CITY
8107 FULTON AVE, MARGATE CITY,NJ	2.5	2004	Residential	Stucco	2.5SS	MARGATE CITY CITY

Fig. 1.4.1: Address list

Prepare a boundary file of the region of interest in geojson format

Define the path in src/configure.py like this:

```
RegionBoundaryFileName = dataDir+"/AtlanticCoastalCities_Boundary.geojson"
```

For this demo, we have prepared this boundary file for you, download from here.

Prepare building footprints in geojson format

AI generated building footprints database -> USBuildingFootprints.

We have prepared a cleaned version that just contains Atlantic coastal cities, download from here.

Define the path of this footprints file in src/configure.py like this:

```
BuildingFootPrintsFileName = dataDir+"/AtlanticCoastalCities_Footprints.geojson"
```

Geocode buildings and create a basic BIM file for this region.

Define the file path to store BIM for all buildings in src/configure.py like this:

```
resultBIMFileName = dataDir+"/Atlantic_Cities_BIM.geojson"
```

Then run the following command from src/preparedata (This will cost $ because it calls Google API. To avoid this, download the Atlantic geocoding file and unzip it in your data/preparedata dir. The code will first look into this dir for geocoding information, if it was not there, the code will call Google API.)

```
python geocoding_addr.py
```

This will create a BIM file Atlantic_Cities_BIM.geojson containing basic building information within the interested region. The generated BIM file can be visulized in a GIS software, such as QGIS.

```
{
    "type": "FeatureCollection",
    "features": [{
                    "type": "Feature",
```

(continues on next page)

(continued from previous page)


```
                "id": "8460",
                "properties": {
                        "id": "8460",
                        "lat": 39.371879,
                        "lon": -74.456126,
                        "address": "1970 W RIVERSIDE DR, ATLANTIC CITY,NJ",
                        "stories": 2,
                        "yearBuilt": 2006,
                        "occupancy": "Residential",
                        "structureType": "Frame",
                        "buildingDescription": "2SF",
                        "city": "ATLANTIC CITY CITY"
                },
                "geometry": {
                        "type": "Polygon",
                        "coordinates": [
                                [
                                        [-74.45606, 39.371837],
                                        [-74.455935, 39.371934],
                                        [-74.456037, 39.372013],
                                        [-74.456162, 39.371916],
                                        [-74.45606, 39.371837]
                                ]
                        ]
                }
        },
        {
                "type": "Feature",
                "id": "8461",
                "properties": {
                        "id": "8461",
                        "lat": 39.3716807,
                        "lon": -74.4513949,
                        "address": "1619 COLUMBIA AVE, ATLANTIC CITY,NJ",
                        "stories": 2,
                        "yearBuilt": 1979,
                        "occupancy": "Residential",
                        "structureType": "Frame",
                        "buildingDescription": "2SF",
                        "city": "ATLANTIC CITY CITY"
                },
                "geometry": {
                        "type": "Polygon",
                        "coordinates": [
                                [
                                        [-74.451353, 39.371717],
                                        [-74.451493, 39.371755],
                                        [-74.451526, 39.37168],
                                        [-74.451386, 39.371643],
                                        [-74.451353, 39.371717]
                                ]
                        ]
                }
        }
    ]
}
```

Train a model

There are pretrained ConvNets released with BRAILS that can be used out of the box.

However, if the user is interested in training his / her own ConvNets, the following is an example demonstrating how to train a classifier (for roof shape), and how to use the trained model in the BRAILS application.

Train

The image data set for training has been prepared by SimCenter and can be download from here:

Charles Wang. (2019). Random satellite images of buildings (Version v1.0) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.3521067.

To train on the downloaded data, run the following command in the terminal

```
cd src/training/
python train_classifier.py --img_dir <IMAGE_DIRECTORY> --model_dir <MODEL_DIRECTORY>
```

IMAGE_DIRECTORY is the directory where you have your images for training

```
IMAGE_DIRECTORY
├── class_1
│       └── *.png
├── class_2
|       └── *.png
├── ...
|
└── class_n
        └── *.png
```

MODEL_DIRECTORY is the directory where the trained model will be saved.

It is better to run the above code on a GPU machine.

Predict

Now we use the trained model to predict roof types based on satellite images.

Firstly we need to download those images by calling Google API (will cost $).

```
cd src/predicting
python downloadRoofImages.py
```

To save $, instead of running the above command, you can just download them from here.

Now predictions can be made using the trained model:

```
cd src/predicting
python predict.py --image_dir <IMAGE_DIRECTORY> --model_path <MODEL_PATH>
```

IMAGE_DIRECTORY is the directory of your images. MODEL_PATH is the folder where you have your trained model saved.

Make predictions

Predict

Now we can use the trained model to predict on new images.

We show a example here: predict roof type based on satellite images.

Firstly we need to download those satellite images by calling Google API (will cost $).

```
cd src/predicting
python downloadRoofImages.py
```

To save $, instead of running the above command, you can just get them from here. that are prepared by SimCenter.

Predictions can now be made using the trained model:

```
cd src/predicting
python predict.py --image_dir <IMAGE_DIRECTORY> --model_path <MODEL_PATH>
```

IMAGE_DIRECTORY is the directory of your images. MODEL_PATH is the folder where you have your trained model saved.

Enhance the data

Fig. 1.4.2: Data enhancement

To enhance the initial database, use SURF to predict missing building information.

For more information about *SURF*, read its documentation.

To install *SURF*, run the following command in the terminal:

```
pip3 install pySURF
```

To enhance the initial database, run the following command in the terminal:

```
python3 SURF.ET-AI.py
# the SURF.ET-AI.py file can be found in github
# https://github.com/charlesxwang/SURF
```

After the running of the enhance script above, a new geojson file containing BIMs for the city will be generated. The missing values in the initial database are now filled with predicted values.

The following figures show the prediction errors for four building properties: year built, number of stories, structure type, occupancy.

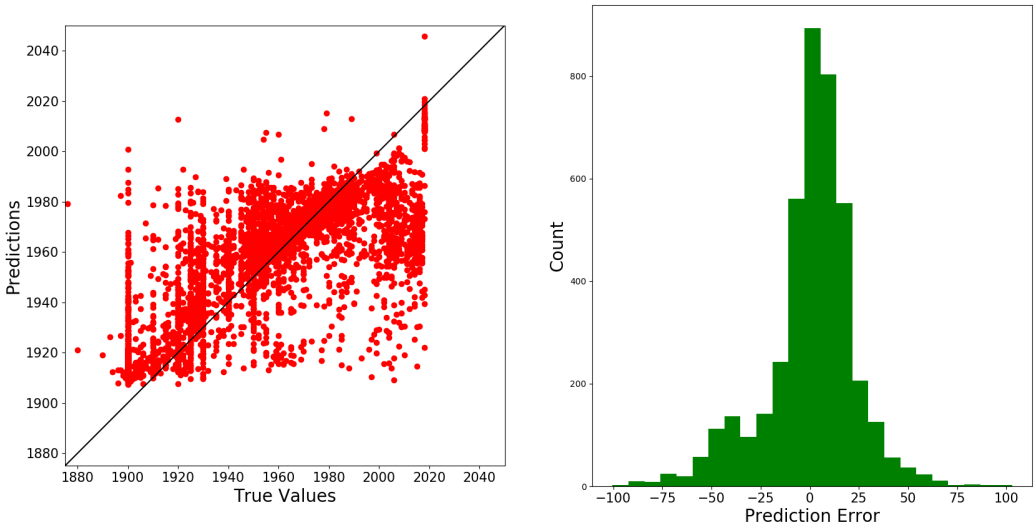


Fig. 1.4.3: Prediction error of year built

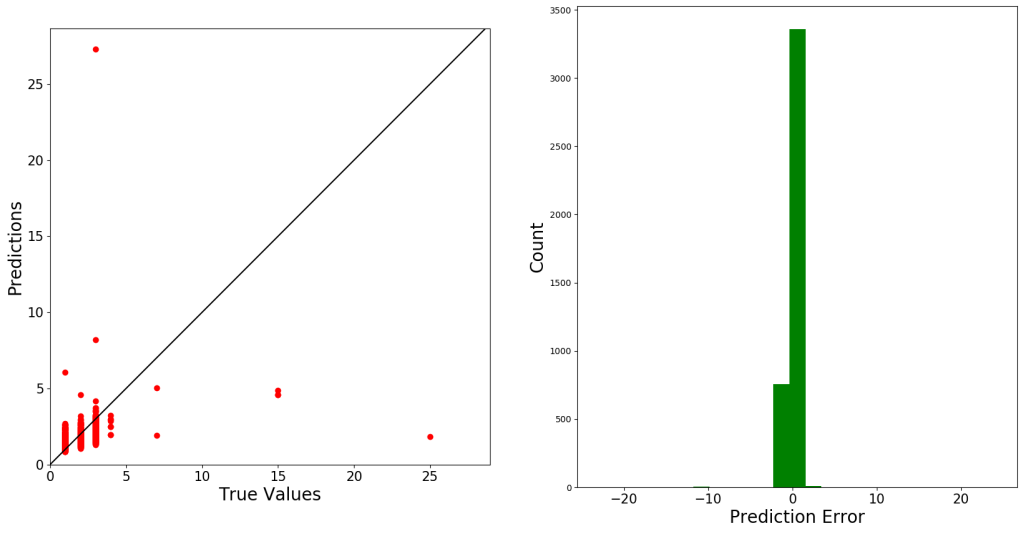


Fig. 1.4.4: Prediction error of number of stories

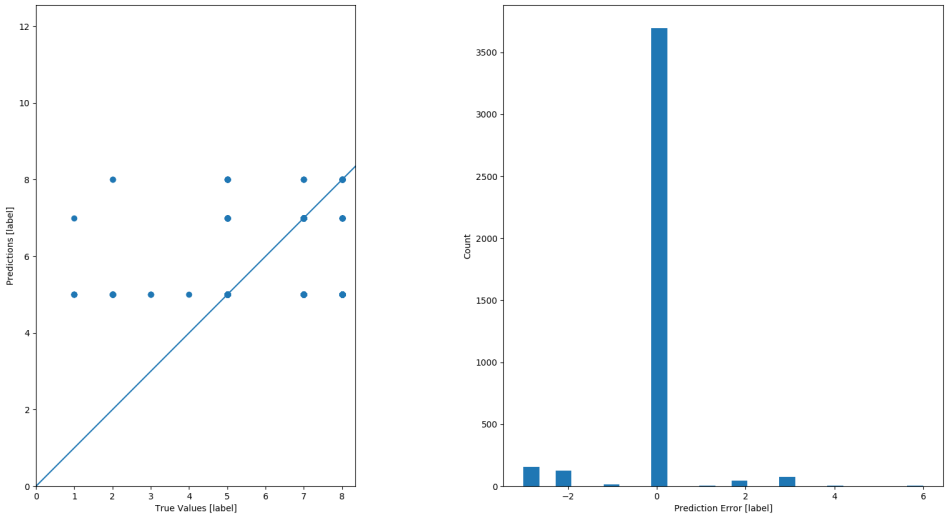


Fig. 1.4.5: Prediction error of structure type

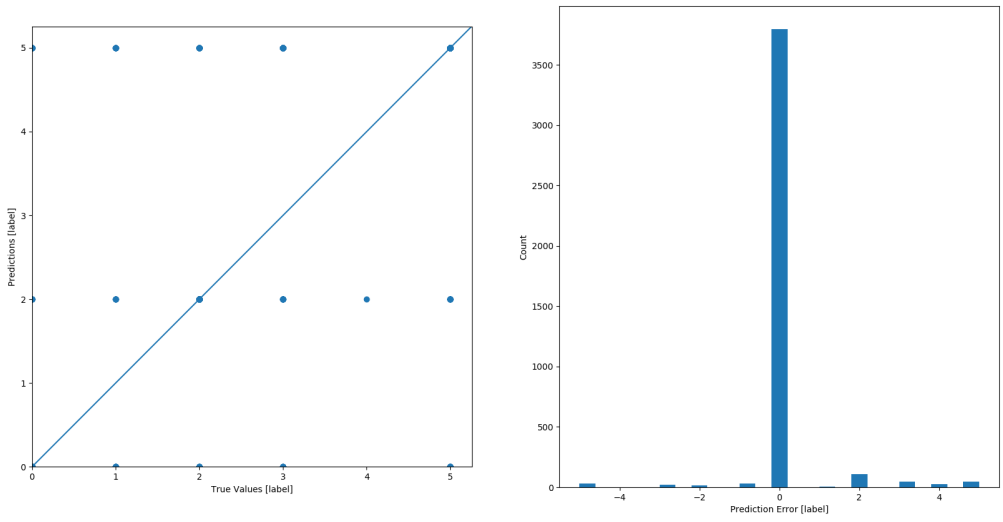


Fig. 1.4.6: Prediction error of occupancy

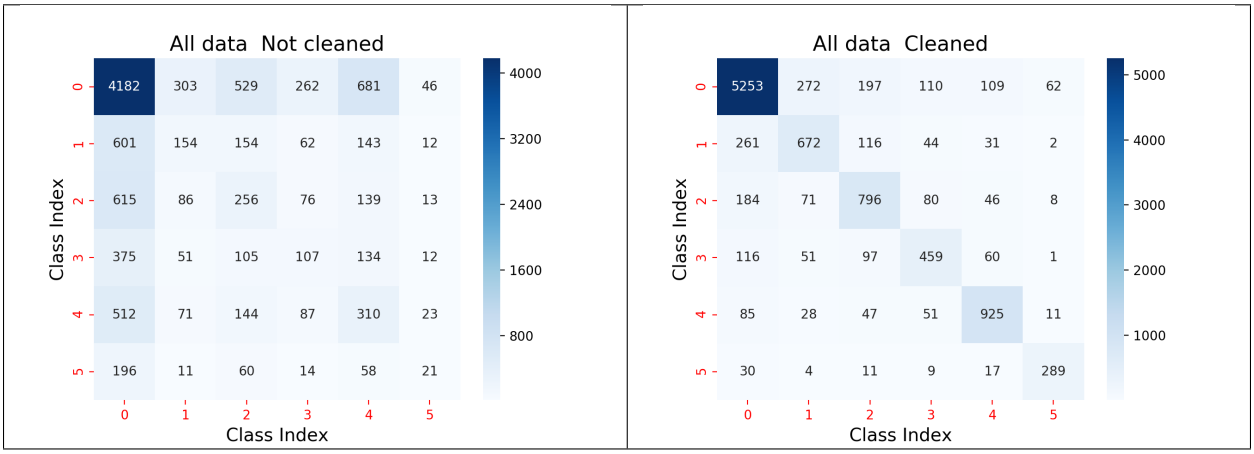
1.4.2 Data Preperation for Training and Inference

Training Data

The accuracy of a deep learning model is highly dependent on the dataset used to train it. If the training data contains minimal noise and includes a sufficient level of features necessary for model development, high model prediction accuracies are attainable. In image-based applications, data noise can be in many forms, such as incorrect image labels and imagery that lack the features sought by the model (e.g., occluded features, etc.).

In developing the pre-trained models in BRAILS, training images are subjected to rigorous prescreening to ensure sufficient visibility of target buildings in each image in the training set. Without this screening step, creating models with reasonable confidence levels becomes difficult, with the training accuracies being inversely proportional to the extent of noise. Table **??** shows an example of one of our observations of this condition throughout SimCenter's model development efforts. The confusion matrices in both figures are for models generated using identical model architecture and hyperparameters. The first confusion matrix in Table **??** is for the model trained on a dataset of images that contained 20% noisy data, while the second matrix is for the model trained on a dataset of images that were fully prescreened before model training. After 100 epochs, the former model attained an F1-score of 47.43%; the latter achieved an F1-score of 79.15% for the same validation set.

Table 1.4.1: Effect of high noise levels on training performance



`noisyImages` shows a sample set of images removed after prescreening.

Table 1.4.2: Sample noisy images removed by the prescreening algorithm

`cleanImages` shows a sample set of images that were deemed suitable for model consumption by the prescreening algorithm.

Table 1.4.3: Sample clean images retained by prescreening

Data Suitable for Inference

1.4.3 Modules

The BRAILS framework consists of a series of modules. The modules can be called and executed in a workflow. Each module can also be used as a standalone function.

Generic Image Classifier

The Generic Image Classifier is a module that can be used for creating user defined classifier.

The user provides categorized images to this module.

An image classifier will be built automatically based on the images provided.

The classifier is then trained and saved locally.

The trained classifier can readily be used for inference readily and can be shared with other users.

During the inference stage, the classifier takes a list of images as the input, and predicts the classes of the images.

The following is an example, in which a classifier is created and trained.

The image dataset for this example contains street view images categorized according to construction materials.

The dataset can be downloaded here.

When unzipped, the file gives the 'building_materials' which is a directory that contains the images for training:

```
building_materials
├── class_1
│       └── *.png
├── class_2
|       └── *.png
├── ...
|
└── class_n
        └── *.png
```

Construct the image classifier

```
# import the module
from brails.modules import ImageClassifier

# initialize the classifier, give it a name
materialClassifier = ImageClassifier(modelName='materialClassifierV0.1')

# load data
materialClassifier.loadData('building_materials')
```

Train the model

```
# train the base model for 50 epochs and then fine tune for 200 epochs
materialClassifier.train(baseModel='InceptionV3', initial_epochs=50,fine_tune_
↪epochs=200)
```

It is recommended to run the above example on a GPU machine.

The following ML model training options are available for selection as the baseModel key:

```
'Xception',
'VGG16',
'VGG19',
'ResNet50',
'ResNet101',
'ResNet152',
'ResNet50V2',
'ResNet101V2',
'ResNet152V2',
'InceptionV3',
'InceptionResNetV2',
'MobileNet',
'MobileNetV2',
'DenseNet121',
'DenseNet169',
'DenseNet201',
'NASNetMobile',
'NASNetLarge',
'EfficientNetB0',
'EfficientNetB1',
'EfficientNetB2',
'EfficientNetB3',
'EfficientNetB4',
'EfficientNetB5',
'EfficientNetB6',
'EfficientNetB7'
```

Use the model

Now you can use the trained model to predict the (building materials) class for a given image.

```
# If you are running the inference from another place, you need to initialize the␣
↪classifier firstly:
from brails.GenericImageClassifier import ImageClassifier
materialClassifier = ImageClassifier(modelName='materialClassifierV0.1')

# define the paths of images in a list
imgs = ['building_materials/concrete/469 VAN BUREN AVE Oakland2.jpg',
        'building_materials/masonry/101 FAIRMOUNT AVE Oakland2.jpg',
        'building_materials/wood/41 MOSS AVE Oakland2.jpg']

# use the model to predict
predictions = materialClassifier.predict(imgs)
```

The predictions will be written in preds.csv under the current directory.

Note: The generic image classifier is intended to illustrate the overall process of model training and prediction. The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Roof Shape Classifier

The Roof Shape Classifier is a module built upon the *Generic Image Classifier* module.

The module is shipped with BRAILS, so you don't have to install it standalone if you've installed BRAILS following the *Installation* instruction.

It takes a list of satellite images of roof tops as the input, and classify the roof types into three categories: gabled, hipped, and flat.

Use the module

A pretrained model is shipped with BRAILS. So you can use it directly without training your own model.

The first time you initialize this model, it will download the model from the internet to your local computer.

The images used in the example can be downloaded from here.

```
# import the module
from brails.modules import RoofClassifier

# initialize a roof classifier
roofModel = RoofClassifier()

# define the paths of images in a list
imgs = ['image_examples/Roof/gabled/76.png',
        'image_examples/Roof/hipped/54.png',
        'image_examples/Roof/flat/94.png']

# use the model to predict
predictions = roofModel.predict(imgs)
```

The predictions look like this:

```
Image :  image_examples/Roof/gabled/76.png     Class : gabled (83.21%)
Image :  image_examples/Roof/hipped/54.png     Class : hipped (100.0%)
Image :  image_examples/Roof/flat/94.png     Class : flat (97.68%)
Results written in file roofType_preds.csv
```

Sample images used in this example are:

Fig. 1.4.7: image_examples/Roof/gabled/76.png Gabled

Fig. 1.4.8: image_examples/Roof/hipped/54.png Hipped

Fig. 1.4.9: image_examples/Roof/flat/94.png Flat

Note: The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Retrain the model

You can retrain the existing model with your own data.

```
# Load images from a folder
roofModel.loadData('folder-of-images')

# Re-train it for only 1 epoch for this demo. You can increase it.
roofModel.retrain(initial_epochs=1)

# Test the re-trained model
predictions = roofModel.predict(imgs)

# Save the re-trained model
roofModel.save('myCoolNewModelv0.1')
```

Occupancy Classifier

The Occupancy Classifier is a module built upon the *Generic Image Classifier* module.

The module is shipped with BRAILS, so you don't have to install it standalone if you've installed BRAILS following the *Installation* instruction.

It takes a list of street view images of residential buildings as the input, and classify the buildings into three categories: RES1 (single family building), RES3 (multi-family building), COM(Commercial building).

Use the module

A pretrained model is shipped with BRAILS. So you can use it directly without training your own model.

The first time you initialize this model, it will download the model from the internet to your local computer.

The images used in the example can be downloaded from here.

```
# import the module
from brails.modules import OccupancyClassifier

# initialize an occupancy classifier
occupancyModel = OccupancyClassifier()

# define the paths of images in a list
imgs = ['image_examples/Occupancy/RES1/36887.jpg',
    'image_examples/Occupancy/RES3/37902.jpg',
    'image_examples/Occupancy/COM/42915.jpg']

# use the model to predict
predictions = occupancyModel.predict(imgs)
```

The predictions look like this:

```
Image :  image_examples/Occupancy/RES1/36887.jpg     Class : RES1 (100.0%)
Image :  image_examples/Occupancy/RES3/37902.jpg     Class : RES3 (100.0%)
Image :  image_examples/Occupancy/COM/42915.jpg     Class : COM (100.0%)
Results written in file tmp/occupancy_preds.csv
```

Sample images used in this example are:

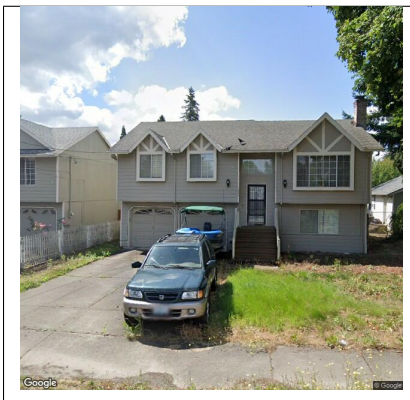


Fig. 1.4.10: Predicted as Single-family Building

Fig. 1.4.11: Predicted as Multi-family Building

Fig. 1.4.12: Predicted as Commercial Building

Note: The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Retrain the model

You can retrain the existing model with your own data.

```
# Load images from a folder
occupancyModel.loadData('folder-of-images')

# Re-train it for only 1 epoch for this demo. You can increase it.
occupancyModel.retrain(initial_epochs=1)

# Test the re-trained model
predictions = occupancyModel.predict(imgs)

# Save the re-trained model
occupancyModel.save('myCoolNewModelv0.1')
```

Soft-story Building Classifier

The Soft-story Building Classifier is a module built upon the *Generic Image Classifier* module.

The module is shipped with BRAILS, so you don't have to install it standalone if you've installed BRAILS following the *Installation* instruction.

It takes a list of street view images of buildings as the input, and classify the buildings into two categories: soft-story building and other building.

Use the module

A pretrained model is shipped with BRAILS. So you can use it directly without training your own model.

The first time you initialize this model, it will download the model from the internet to your local computer.

The images used in the example can be downloaded from here.

```
# import the module
from brails.modules import SoftstoryClassifier

# initilize a soft-story classifier
ssModel = SoftstoryClassifier()

# define the paths of images in a list
imgs = ['image_examples/Softstory/Others/3110.jpg',
        'image_examples/Softstory/Softstory/901.jpg']

# use the model to predict
predictions = ssModel.predict(imgs)
```

The predictions look like this:

```
Image :  image_examples/Softstory/Others/3110.jpg     Class : others (96.13%)
Image :  image_examples/Softstory/Softstory/901.jpg     Class : softstory (96.31%)
Results written in file softstory_preds.csv
```

Sample images used in this example are:

Fig. 1.4.13: image_examples/Softstory/Others/3110.jpg Non-Soft-story Building

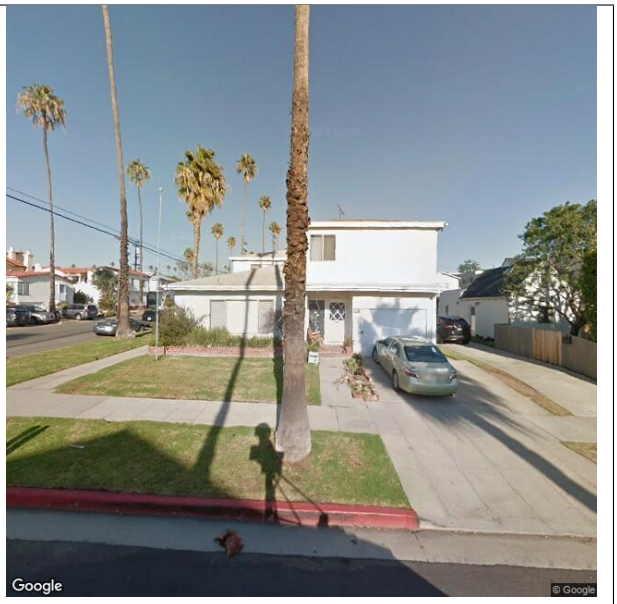


Fig. 1.4.14: image_examples/Softstory/Softstory/901.jpg Soft-story Building

Note: The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Retrain the model

You can retrain the existing model with your own data.

```
# Load images from a folder
ssModel.loadData('folder-of-images')

# Re-train it for only 1 epoch for this demo. You can increase it.
ssModel.retrain(initial_epochs=1)

# Test the re-trained model
predictions = ssModel.predict(imgs)

# Save the re-trained model
ssModel.save('myCoolNewModelv0.1')
```

Year Built Classification

With this module images of houses can be classifier into categories, where each represents a range of years in which the house was built. For classification, the path of a folder holding the images has to be supplied. The result will be a comma separated value file in that folder, listing the filenames and classification result.

The code is optimized to extract suitable features from Google Streetview images to classify houses into decades. However, the code does not make assumptions about the semantic meaning of the provided folders. They have to be consistent between the training, validation and test folders but can be categorized by decades, or short or longer than a decade, and any specified timespans.

The predictions fall in 6 categories:

```
0 : before 1969
1 : 1970-1979
2 : 1980-1989
3 : 1990-1999
4 : 2000-2009
5 : after 2010
```

Use the module

A pretrained model is shipped with BRAILS. So you can use it directly without training your own model.

The first time you initialize this model, it will download the model from the internet to your local computer.

```
# import the module
from brails.modules import YearBuiltClassifier

# initialize a year classifier
model = YearBuiltClassifier()

# define the paths of images in a list
from glob import glob
imgs = glob('image_examples/Year/*/*.jpg')

# use the model to predict
predictions = model.predict(imgs)
```

The module is currently under active development and testing. More details about the training, modification, improvement of this module can be found here.

Note: The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Raised Foundation Classification

What is Raised Foundation Classification

The code in this package enables to see if a building is elevated on piles piers or posts (PPP).

For classification, the path of a folder holding the images has to be supplied. The result will be a comma separated value file in that folder, listing the filenames, classification (1: elevated, 0: not elevated), and the confidence of the prediction.

Use the module

A pretrained model is shipped with BRAILS. So you can use it directly without training your own model.

The first time you initialize this model, it will download the model from the internet to your local computer.

The images used in the example can be downloaded from here.

```
# import the module
from brails.modules import FoundationHeightClassifier

# initialize a roof classifier
model = FoundationHeightClassifier()

# define the paths of images in a list
from glob import glob
imgs = glob('image_examples/Foundation/*/*.jpg')

# use the model to predict
predictions = model.predict(imgs)
```

The predictions look like this:

```
Image :  a.jpg     Class : 1 (52.5%)
Image :  b.jpg     Class : 1 (56.36%)
Image :  c.jpg     Class : 0 (83.4%)
Image :  d.jpg     Class : 0 (63.43%)
Results written in file tmp/FoundationElevation.csv
```

The images used in this example are:

Fig. 1.4.15: image_examples/Foundation/Elevated/a.jpg Elevated

Fig. 1.4.16: image_examples/Foundation/Elevated/b.jpg Elevated

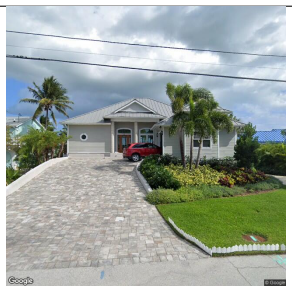


Fig. 1.4.17: image_examples/Foundation/NotElevated/c.jpg Not Elevated

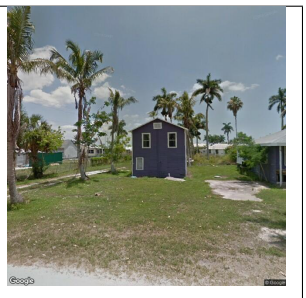


Fig. 1.4.18: image_examples/Foundation/NotElevated/d.jpg Not Elevated

This module is currently under active development and testing. Currently, for the data set used, classification reaches

an F1-score of 72% on a random test set that holds 20% of the data. Further optional code to improve the quality and speed of the classification is available. More details about the training, modification, improvement of this module can be found here.

Note: The classifier takes an image as the input and will always produce a prediction. Since the classifier is trained to classify only a specific category of images, its prediction is meaningful only if the input image belongs to the category the model is trained for.

Number of Floors Detector

The module is bundled with BRAILS, hence its use does not require a separate installation if BRAILS was installed following the *Installation* instructions.

This module enables automated detection of number of floors in a building from image input. It takes the directory for an image or folder of images as input and writes the number of floor detections for each images into a CSV file.

Use the module

```
# Import the module
from brails.modules import NFloorDetector

# Initialize the detector
nfloorDetector = NFloorDetector()

# Define the path of the images:
imDir = "datasets/test/"

# Detect the number of floors in each image inside imDir and write them in a
# CSV file. The prediction can be also assigned to DataFrame variable:
predictions = nfloorDetector.predict(imDir)

# Train a new detector using EfficientDet-D7 for 50 epochs
nfloorDetector.load_train_data(rootDir="datasets/")
nfloorDetector.train(compCoeff=7,numEpochs=50)
```

1.4.4 Workflow

The workflow is designed to facilitate the creation of regional building inventories.

It is implemented in a class, CityBuilder.

```
from brails.CityBuilder import CityBuilder

cityBuilder = CityBuilder(attributes,
                numBldg,
                random,
                bbox,
                place,
                footPrints,
                save,
                fileName,
                workDir,
```

(continues on next page)

(continued from previous page)

```
                GoogleMapAPIKey,
                overwrite,
                reDownloadImgs)
```

attributes (list) A list of building attributes, such as ['*roofshape*', '*occupancy*', '*softstory*', '*elevated*', '*year*', '*numstories*'], which are available in the current version.

numBldg (int) Number of buildings to generate.

random (bool) Randomly select numBldg buildings from the database if random is True.

bbox (list) [north, west, south, east], which are latitudes and longitudes of two corners that define a region of interest.

place (str) The region of interest, e.g., 'Berkeley, California'.

footPrints (str) The footprint provide, choose from 'OSM' or 'Microsoft'. The default value is 'OSM'.

save (bool) Save temporary files. Default value is True.

fileName (str) Name of the generated BIM file. Default value will be generated if not provided by the user.

workDir (str) Work directory where all files will be saved. Default value is './tmp'

GoogleMapAPIKey (str) Google API Key. Must be provided to use the workflow.

overwrite (bool) Overwrite existing tmp files. Default value is False.

reDownloadImgs (bool) Re-download even an image exists locally. Default value is False.

Specify attributes to be collected

Use the key 'attributes' to specify a list of attributes you intend to collect for each building. Available ones in the current version include: ['roofshape', 'occupancy', 'softstory', 'elevated', 'year', 'numstories']. These attributes will be inferred from images using specific *modules*.

- roofshape is the roof class, details can be found in *Roof Shape Classifier*.
- occupancy is the occupancy class, details can be found in *Occupancy Classifier*.
- softstory is the soft-story attribute, details can be found in *Soft-story Building Classifier*.
- elevated is the foundation elevation attribute, details can be found in *Raised Foundation Classification*.
- year is the year built, details can be found in *Year Built Classification*.
- numstories is the number of stories, details can be found in *Number of Floors Detector*.

Limit the number of buildings to be collected

The workflow will download a street view image and a satellite view image for each building. The images are downloaded from Google Maps using your personal Google API Keys. The price of the API calls can be found here. As of February 3, 2021, $7 per 1,000 street view images and $2 per 1,000 satellite images. Each Google account has $200 free monthly usage. Exceeding that limit will result in being charged by Google.

You can use the key 'numBldg' to limit the number of buildings to be generated.

Control the selection randomness

In a region, numBldg of buildings will be generated. You can use the key 'random' to specify if you want to randomly select numBldg buildings from the database. If its value is False, BIM will be generated for the first numBldg buildings found in the footprint database.

Define the region of interest

There are two options to define the region of interest: 'bbox' or 'place'.

If 'bbox' is provided, the workflow will retrieve numBldg buildings within the bounding box defined by 'bbox'.

If 'bbox' is empty and 'place' is provided, the workflow will search the database based on 'place'.

Footprints options

Use the key 'footPrints' to specify the source of building footprints to be used in the workflow. Currently, the workflow supports ' OSM' and 'Microsoft'.

Examples

Check the *Examples*.

1.5 Troubleshooting

1.5.1 Installation

Windows users may experience difficulties because of two dependencies required: GDAL and Fiona.

If you are a Conda user and you are installing BRAILS in the Conda environment, this is unlikely to happen.

If you are not a Conda user, the solution is to download Fiona and GDAL's wheel files (https://www.lfd.uci.edu/~gohlke/pythonlibs/) and manually install them.

Make sure you choose the correct files based on your Python version.

For example, if you are using Python3.8:

1. Go to https://www.lfd.uci.edu/~gohlke/pythonlibs/ and download these files:

   GDAL-3.1.4-cp38-cp38-win_amd64.whl

   Fiona-1.8.18-cp38-cp38-win_amd64.whl

   here cp38 means python3.8.
2. pip install GDAL-3.1.4-cp38-cp38-win_amd64.whl
3. pip install Fiona-1.8.18-cp38-cp38-win_amd64.whl

Then you'll be able to:

pip3 install BRAILS

1.5.2 Internet connection

The trained models and accompanying datasets, when called the first time, need to be downloaded from the internet.

Images also need to be downloaded during the running.

Therefore, please make sure you are connected to the internet.

1.6 Examples

1.6.1 Example 1: Modules

The following is an example showing how to call pretrained models to predict on images.

You can run this example on your computer or in this notebook on Google Colab.

The images used in the example can be downloaded from here: image_examples.zip.

```
# import modules
from brails.modules import RoofClassifier, OccupancyClassifier,␣
↪SoftstoryClassifier

# initilize a roof classifier
roofModel = RoofClassifier()

# initilize an occupancy classifier
occupancyModel = OccupancyClassifier()

# initilize a soft-story classifier
ssModel = SoftstoryClassifier()

# use the roof classifier

imgs = ['image_examples/Roof/gabled/76.png',
        'image_examples/Roof/hipped/54.png',
        'image_examples/Roof/flat/94.png']

predictions = roofModel.predict(imgs)

# use the occupancy classifier

imgs = ['image_examples/Occupancy/RES1/51563.png',
        'image_examples/Occupancy/RES3/65883.png']

predictions = occupancyModel.predict(imgs)

# use the softstory classifier

imgs = ['image_examples/Softstory/Others/3110.jpg',
        'image_examples/Softstory/Softstory/901.jpg']

predictions = ssModel.predict(imgs)
```

1.6.2 Example 2: Workflow

The following is an example showing how to create a building inventory for a city.

You can run this example on your computer or in this notebook on Google Colab.

You need to provide the Google maps API key for downloading street view and satellite images.

Instructions on obtaining the API key can be found here: https://developers.google.com/maps/documentation/embed/get-api-key.

Use should limit the number of buildings (numBldg) because of *this*.

```
from brails.CityBuilder import CityBuilder

cityBuilder = CityBuilder(attributes=['occupancy','roofshape'],
                numBldg=1000,random=False, place='Lake Charles, LA',
                GoogleMapAPIKey='put-your-API-key-here',
                overwrite=True)

BIM = cityBuilder.build()
```

attributes (list) A list of building attributes, such as [‘story’, ‘occupancy’, ‘roofshape’], which are available in the current version.

numBldg (int) Number of buildings to generate.

random (bool) Randomly select numBldg buildings from the database if random is True.

place (str) The region of interest, e.g., Berkeley, California.

GoogleMapAPIKey (str) Google API Key.

overwrite (bool) Overwrite existing tmp files. Default value is False.

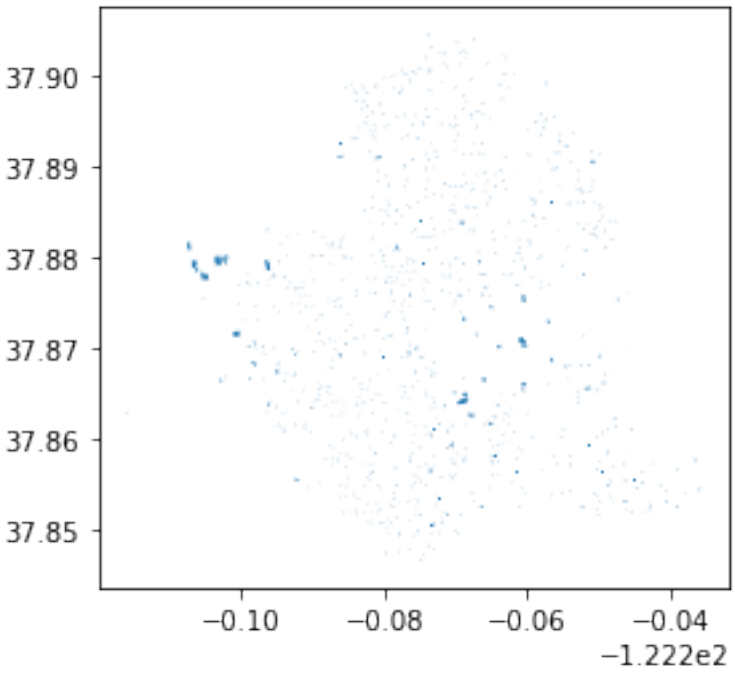


Fig. 1.6.1: Generated Buildings

1.6.3 Example 3: Workflow

The following is an example showing how to create a building inventory for a region defined using a bounding box.

You can run this example on your computer or in this notebook on Google Colab.

You need to provide the Google maps API key for downloading street view and satellite images.

Instructions on obtaining the API key can be found here: https://developers.google.com/maps/documentation/embed/get-api-key.

Use should limit the number of buildings (numBldg) because of *this*.

```
from brails.CityBuilder import CityBuilder

cityBuilder = CityBuilder(attributes=['softstory','occupancy','roofshape'],
                numBldg=100,random=False, bbox=[37.872187, -122.282178,37.
↪870629, -122.279765],
                GoogleMapAPIKey='put-your-API-key-here',
                overwrite=True)

BIM = cityBuilder.build()
```

attributes (list) A list of building attributes, such as [‘story’, ‘occupancy’, ‘roofshape’], which are available in the current version.

numBldg (int) Number of buildings to generate.

random (bool) Randomly select numBldg buildings from the database if random is True.

bbox (list) [north, west, south, east], which defines a region of interest.

GoogleMapAPIKey (str) Google API Key.

overwrite (bool) Overwrite existing tmp files. Default value is False.

1.7 Bugs & Feauture Requests

If you have any bugs, feauture requests, or questions about how to install or use the application please post on the Message board. To avoid duplication try the *Search* feature before creating a *New Topic*. When creating a *New Topic* to report a bug or new feature request it would be helpful if you could place in the *Subject* area, as shown in Fig. 1.7.1, an indication of what the post is about:

1. **BUG**, for bugs, i.e. BUG: Program crashes when starting
2. **FEATURE**, for new feature requests, i.e. FEATURE: Incorporate OpenSeesPy

1.8 Copyright and License

BRAILS is copyright “The Regents of the University of California” and is licensed under the following BSD license:

```
The source code is licensed under a BSD 2-Clause License.

Copyright (c) 2017-2021, The Regents of the University of California

All rights reserved.
```

(continues on next page)

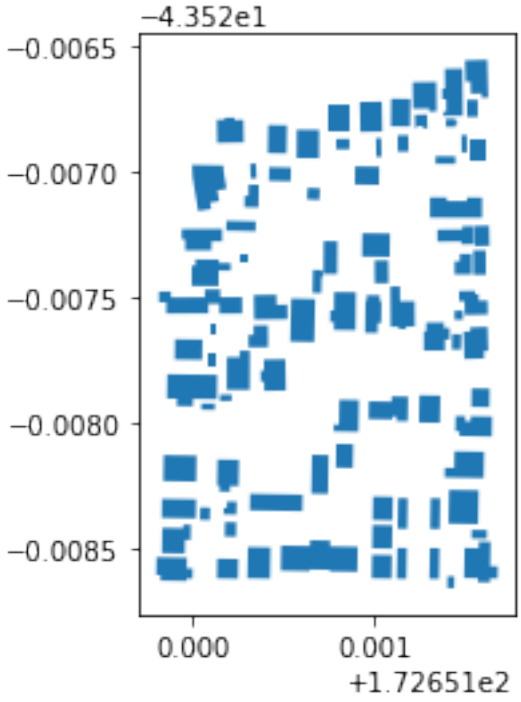


Fig. 1.6.2: Generated Buildings

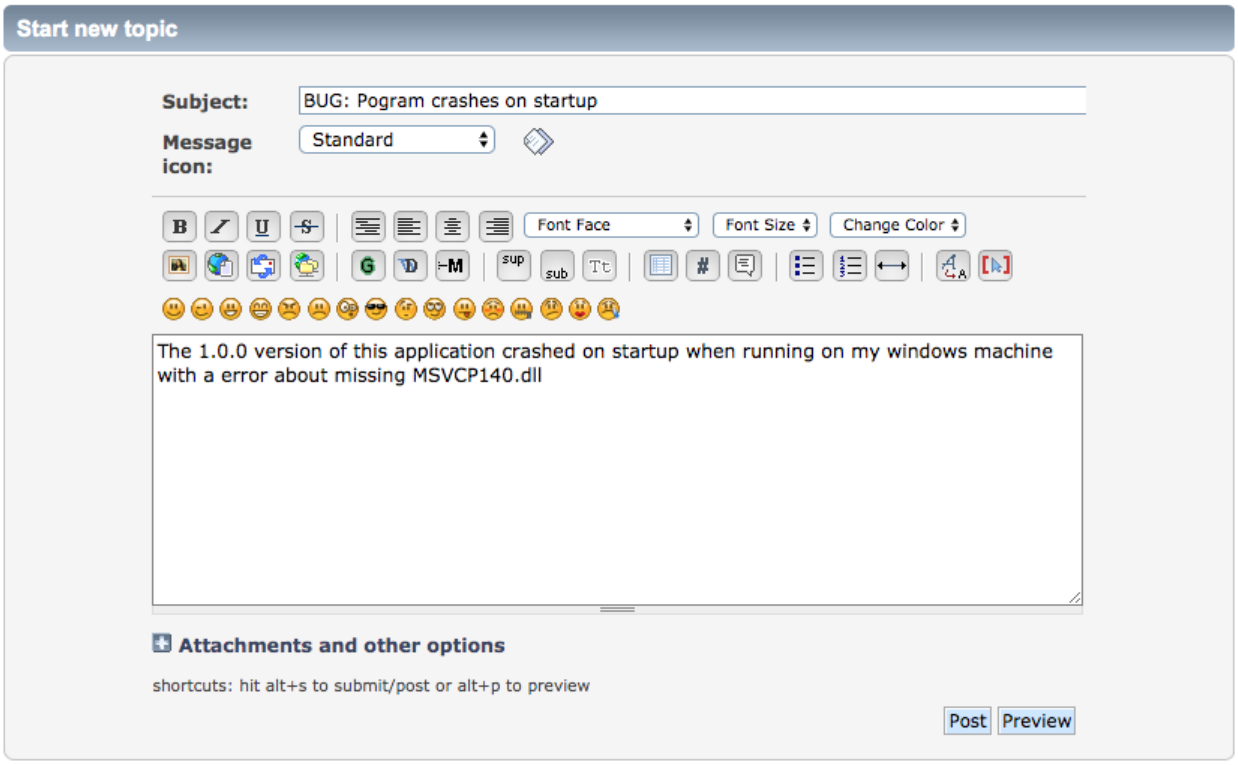


Fig. 1.7.1: Example of Submitting a Bug Report to Message Board

(continued from previous page)

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
  and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

1.9 Theory and Implementation

1.9.1 Framework

This section presents a framework that works as an abstraction providing generic functionalities and can be selectively changed by additional user-written codes with user-provided input data, thus providing a region-specific building information harvesting tool. The framework provides a standard way to create realistic building inventory databases and it is a universal, reusable environment that provides particular functionalities facilitating regional-scale building information modeling.

As shown in Fig. 1.9.1, the framework consists of two steps: data fusion and data enhancement.

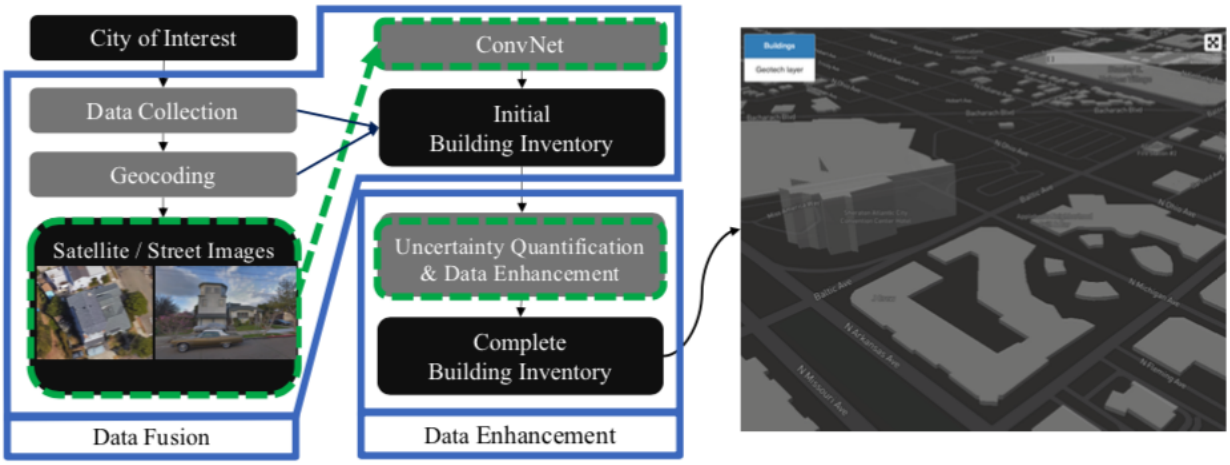


Fig. 1.9.1: Framework

Due to the complexity and the size of source data and the cost to collect it, building information at the regional-scale is usually not able to be inferred from a single resource but multiple resources, such as images, point clouds, property tax

records, crowd sourcing map, etc. And these data usually belong to different owners and stored in different formats. The framework combines these sources using data fusion (or data integration), which is the process of integrating multiple building information data to produce more consistent, accurate, and useful information than that provided by any individual data source. The expectation is that fused building information data is more informative and synthetic than the original data.

The product of data fusion is an initial building inventory, in which a significant amount of building information is missing because of data scarcity. For example, crowd sourcing maps have issues with completeness, especially for rural regions. They are more complete in densely urbanized areas and targeted areas of humanitarian mapping intervention. Similarly, a significant amount of property tax assessment records are missing from administrative databases. The incompleteness issue is common for almost all data sources. In this framework, the missing values will be filled by modellings performed based on the incomplete initial database.

Extracting building information from images

An initial building inventory, containing basic indexing information such as addresses or coordinates of individual buildings, and other basic descriptions such as year built and structure type, has been created in the previous section. Based on the indexing information, satellite or street view images of each building can be retrieved using Google Maps API. Then, the deep learning technique ConvNet is utilized to extract building features (that don't exist in the initial database) from these images. ConvNet is a class of deep neural networks inspired by biological processes in that the connectivity pattern between neurons resembles the organization of the animal visual cortex [HW68] and is most commonly applied to analyzing images. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

ConvNet is a supervised learning algorithm, which means the images need to be labeled for training. Therefore the most important part is to build a labeled dataset. OSM is a platform hosting real world infrastructure information labeled by human. For a typical building, the information might be found in OSM includes: height, number of stories, structure type, exterior material, footprint shape, usage, etc. These information are the valuable source for describing the built environment. However, only a limit number of buildings are labeled in OSM. In this study, the investigators propose to harvest these labels and associate them with images to build a database for deep learning. The ConvNets trained on these database are used to predict building properties when given any images containing unseen and unlabeled buildings. This way, as long as the satellite/street view images of a city can be obtained, a database of building information can be created. The pipeline to extract a specific building property from images is listed as following:

1. Identify a visually comprehensible building property (e.g., exterior construction material) that is intended to be extracted.
2. Retrieve satellite/street view images of individual buildings from Google Map API
3. Label the retrieved images using tags (e.g., exterior construction material type) found in OSM
4. Train a ConvNet on the labeled images to classify between types
5. Apply the trained ConvNet to unlabeled satellite / street view images of buildings in the city of interest

Repeat the above five steps for other building properties, such as number of stories, structural type, etc., as long as they can be visually identified from images. The ConvNet-identified building information is then merged into the initial database resulting in a more detailed inventory.

It should be noted that, due to reasons like heavy occlusions in images or bad viewpoints, predictions from ConvNet are not always with acceptable confidence. To tackle this issue, in *Data enhancement*, a machine learning based approach will be employed to enhance the database.

Data fusion

In order to combine building information obtained form different sources, a data fusion process is designed, as shown in Fig. 1.9.2. There are two starting points of the data flow pipeline: one is the address list and the other one is the building footprints.

The address list is used as the index for querying the supporting data sources (e.g., OSM, tax records, other user provided data, etc.). Once raw data is fetched from these sources, it will be filtered and cleaned to remove duplicated properties and then merged while missing values represented by place holders.

The building footprint is an important supporting data the framework relies on. The method that is commonly used to get building footprints at a large scale is semantic segmentation on high-resolution satellite maps [LHF+19][ZKJS18][BHF+19]. Since it is out of the focus of this study, instead of performing segmentation in the proposed framework on the fly, the framework retrieves footprints from a dataset released by [Mic]. This dataset contains footprints for almost all buildings in the United States extracted from high-resolution satellite maps. For regions outside of the United States, footprints can be inferred from satellite images using semantic segmentation methods in aforementioned references. Each geotagged address has a unique coordinate, therefore can be merged with corresponding building footprints.

Using address as the index, the aforementioned filtered and cleaned basic building information retrieved from multiple data sources can now be merged into the main stream of the data flow pipeline. For buildings with missing information, satellite and street view images of each building are then retrieved and fed into pretrained ConvNets, and predictions on the building features such as number of stories, roof types, etc., can be obtained. Merging the predicted values into the data stream results in the initial building database.

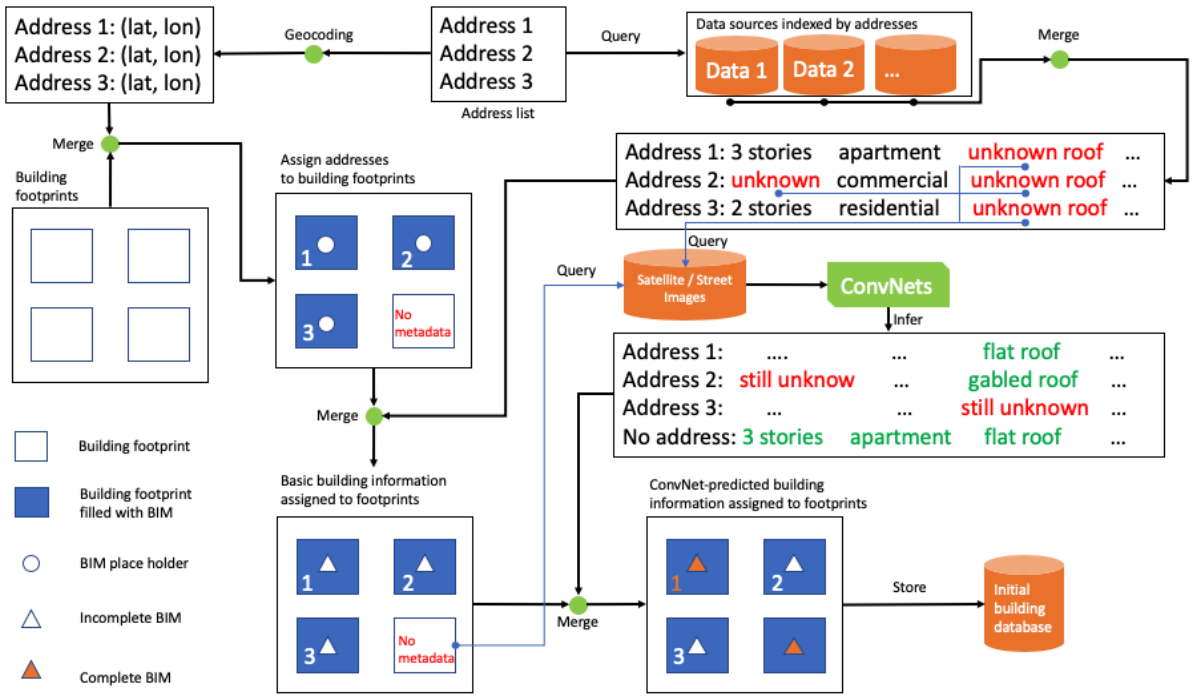


Fig. 1.9.2: Framework

Data enhancement

Note that, after the fusion, the initial building database is still incomplete. Some of the reasons are: firstly no data source is perfect and there are usually a considerable amount of missing items in them; secondly some missing values, for example the year of construction of an individual building, are either visually incomprehensible to a ConvNet, or for some visually comprehensible features, for example the number of stories, if the building is occluded by other objects, usually a tree or a car, in the image, which happens quite often, the feature can not be predicted accurately by ConvNets. These reasons leave gaps in the initial building database. This section describes a machine learning - based data enhancement method to fill the gaps in the data.

Rather than merely a random assortment of objects in space, landscapes, natural resources, the human built environment, and other objects on Earth have orders, which can be described using a spatial patterns - a perceptual structure, placement, or arrangement of objects and the space in between those objects. Patterns may be recognized because of their distance, maybe in a line or by a clustering of points, and other arrangement types.

Such kind of spatial patterns, i.e., the arrangement of individual buildings in space and the geographic relationships among them, exist in the distribution of buildings, too. Buildings, when built, usually have a relationship between each other, i.e., one building is located at a specific location is usually because of another. They can be clustered or dispersed based on their attributes, such as building type, value, construction material, etc., which are usually the manifestation of the demographic characteristics of neighborhoods, such as household income or race. For example, as the city shown in the map Fig. 1.9.3, there are areas denser with buildings than others and clusters of certain types of building are easy to be found in certain regions.

Fig. 1.9.3: Satellite view of buildings in San Francisco

The capability of evaluating spatial patterns is a prerequisite to understanding the complicated spatial processes underlying the distribution of a phenomenon.

In spatial statistics, the semivariogram is a function describing the degree of spatial dependence of a spatial random field or stochastic process. As such, statistics of spatial semivariogram provide a useful indicator of spatial patterns. Semivariogram is essentially a meassure of the degree of dissimilarity between observations as a function of distance. It equals to half the variance of two random variables separated by a vector distance $\boldsymbol{h}$

[Goo97][Van10][WCSJ17][WC18].

$$\gamma(\boldsymbol{h}) = \frac{1}{2} Var[Z(\boldsymbol{\mu}) - Z(\boldsymbol{\mu} + \boldsymbol{h})] \tag{1.9.1}$$

where $Z(\boldsymbol{\mu})$ is the observation at a spatial location μ; $Z(\boldsymbol{\mu} + \boldsymbol{h})$ is the observation at a spatial location $\boldsymbol{\mu} + \boldsymbol{h}$.

It is expected that buildings far away from each other will are more different than buildings that are close to each other. Because based on the first rule of geography that things close together are more similar than things far apart, semivariogram is generally low when two locations are close to each other (i.e. observations at each point are likely to be similar to each other. Typically, semivariogram increases as the distance between the locations grows until at some point the locations are considered independent of each other and semi-variance no longer increases. In the case of buildings, semivariograms will give measures of how much two buildings will vary in attributes (such as height, number of stories, etc.) regarding to the distance between those samples.

Semivariogram function is employed to explore the spatial patterns of different building properties within a selected region. The results show that buildings were indeed built following certain spatial patterns. As a demonstration, the spatial semivariograms of two building properties, number of stories and year of construction, are plotted in Fig. 1.9.4 and Fig. 1.9.5. The horizontal axis represents the distance between a pair of buildings, while the vertical axis represents the dis-similarity of these buildings. The semivariogram figures show that with the increase of the distance between any two buildings, the dis-similarity between them, regrading to number of stories and the year of construction for an example, increased and then fluctuated. Apparently the incremental relationship between the distance and dis-similarity is neither linear nor following any obvious rule. Another note that deserves to be taken here is the curves revealed in Fig. 1.9.4 and Fig. 1.9.5 are city- or region-specific, i.e., the semivariogram curve may reflect the truth of the region being investigated, and may not be exactly correct or directly applicable for describing another region. In other words, the spatial dependence of building features are regional-specific and the semivariogram curves vary regionally.

Since the semivariograms (Fig. 1.9.4 and Fig. 1.9.5) clearly show there is a spatial pattern of the distribution of a certain building property, there must be a function for mapping neighbor information $\boldsymbol{Z}_p$ into Z_n. This function can be constructed implicitly using a neural network.

Imagine a neighborhood consisting of three buildings, Fig. 1.9.6. Pretrained ConvNets can easily extract attributes of building at two ends, such as number of stories, occupancy, structure type, etc. However, for the building in the middle, which is heavily occluded by a tree in this case, no information can be extracted from the image with a satisfying confidence. However, it is possible to predict the features of the building in the middle based on the information of its neighbors, because Fig. 1.9.4 and Fig. 1.9.5 indicates that the attributes of buildings within a community are correlated with each other. The correlations can be learned by neural networks using SURF.

As mentioned in the previous sections, there is a significant portion of building information still missing from the initial database or can not be extracted from images. Using SURF, the missing values can be predicted based on known values of neighboring buildings, hence the gaps in the initial database are filled and the regional building information database is enhanced. Details about how to do this can be found in the documentation of SURF.

1.9.2 Modules

Roof type classifier

Roof type is a crucial information for wind hazard analyses of buildings because it is a key attribute needed for consideration of wind effects on structures.

There are three major roof types, as shown in Table 1.9.1, that are widely used in the world: flat, gabled, hipped.

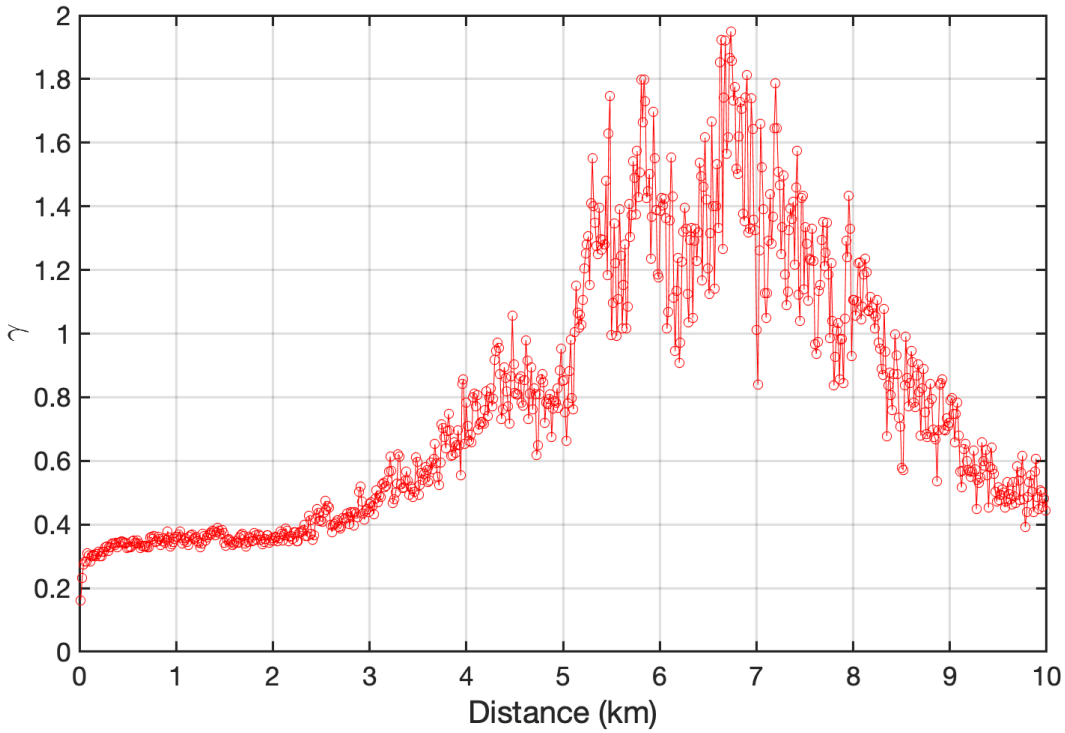


Fig. 1.9.4: Spatial patterns of building information expressed in semivariogram of the number of stories (The horizontal axis represents the distance between a pair of buildings, while the vertical axis represents the dis-similarity of these buildings.) These curves are calculated based on a building dataset covering four coastal cities in the Atlantic County, New Jersey

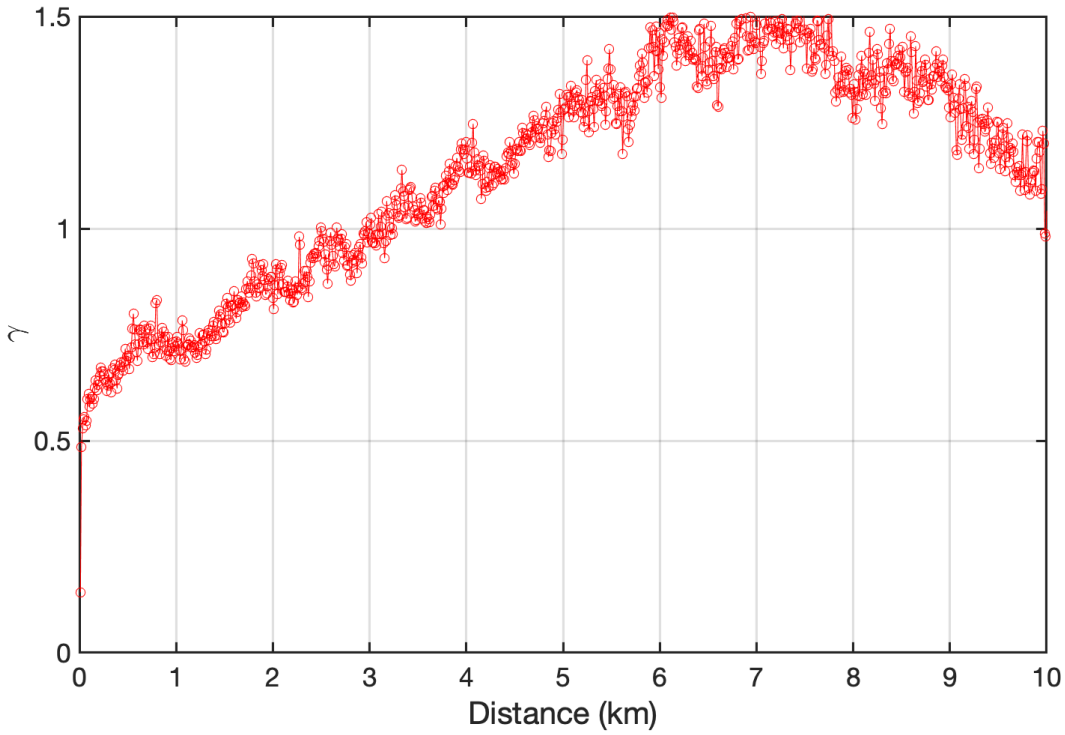


Fig. 1.9.5: Spatial patterns of building information expressed in semivariogram of the year of construction (The horizontal axis represents the distance between a pair of buildings, while the vertical axis represents the dis-similarity of these buildings.) These curves are calculated based on a building dataset covering four coastal cities in the Atlantic County, New Jersey

Fig. 1.9.6: A neighborhood street view

Table 1.9.1: Roof prototypes

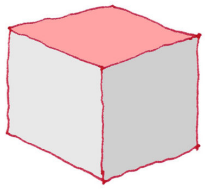
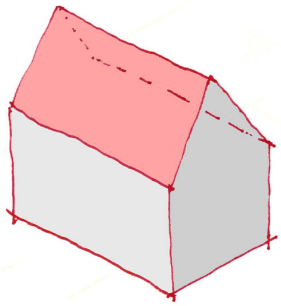
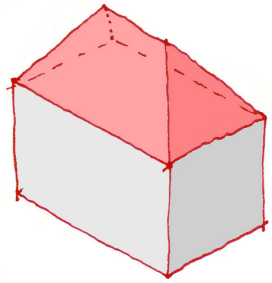
		

Fig. 1.9.7: Flat	Fig. 1.9.8: Gabled	Fig. 1.9.9: Hipped

Correspondingly, a typical satellite image of each roof type is shown in Table 1.9.2.

Table 1.9.2: Example satellite images of different roof types

		
Fig. 1.9.10: Flat	Fig. 1.9.11: Gabled	Fig. 1.9.12: Hipped

Satellite images are a scalable source for inferring roof type information. In the attempt to determine roof type for every building in a region, a ConvNet classifier is trained to take a satellite image of a building and predicts its roof type. A training data set of 6,000 satellite images (2,000 for each roof type: flat, gabled, hipped) is collected. Specifically, ResNet [HZRS16], which is a widely-used ConvNet architecture for image feature recognition, is employed.

The architecture of the model is shown in Fig. 1.9.20. In this module, we used a 50-layer ResNet.

Occupancy classifier

Occupancy class is an important parameter for natural hazard risk analysis of a building. The occupancy model in the current version of BRAILS is trained on a dataset consists of residential buildings that are classified into three categories: single-family buildings (RES1), multi-family buildings (RES3) and commercial buildings (COM). Data of other occupancy classes are being collected and will be added to the new versions of BRAILS.

Examples of different occupancy type is shown in Table 1.9.3.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 1.9.13: ResNet

Table 1.9.3: Examples of different occupancy types

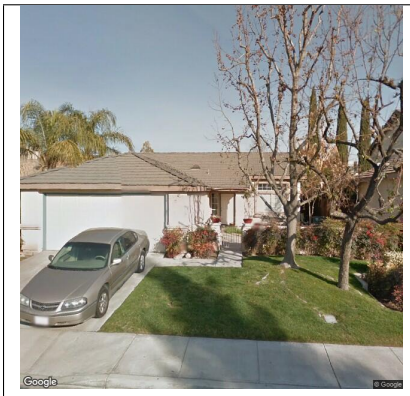


Fig. 1.9.14: Single-Family Building

Fig. 1.9.15: Multi-Family Building

Fig. 1.9.16: Commercial Building

The classifier is trained using a 50-layer ResNet [HZRS16], a widely used ConvNet architecture for images feature recognition.

Its architecture is shown in Fig. 1.9.20.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 1.9.17: ResNet

Soft-story

Structural deficiencies identification is a crucial task in seismic risk evaluation of buildings. In case of multi-story structures, abrupt vertical variations of story stiffness are known to significantly increase the likelihood of collapse during earthquakes. These buildings are called soft story buildings. Identifying these buildings is vital in earthquake loss estimation and mitigation.

One example of soft-story failure is shown in Table 1.9.4.

Table 1.9.4: Soft-story failure

Fig. 1.9.18: Soft-story collapse

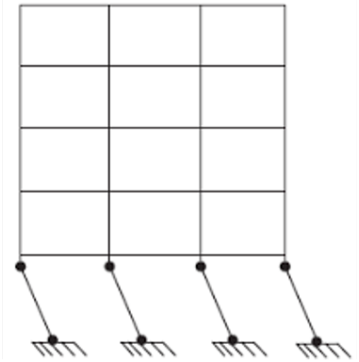


Fig. 1.9.19: Failure mechanism

The classifier is trained using a 50-layer ResNet [HZRS16], a widely used ConvNet architecture for images feature recognition.

Its architecture is shown in Fig. 1.9.20.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 1.9.20: ResNet

Number of Floors Detector

The number of Floors Detector is implemented based on object detection - each floor of a building is detected as a target object.

In general, all modern object detectors can be said to consist of three main components:

1. A backbone network that extracts features from the given image at different scales,
2. A feature network that receives multiple levels of features from the backbone and returns a list of fused features that identify the dominant features of the image,
3. A class and box network that takes the fused features as input to predict the class and location of each object, respectively.

EfficientDet models use EfficientNets pretrained on ImageNet for their backbone network. For the feature network, EfficienDet models use a novel bidirectional feature pyramid network (BiFPN), which takes level 3 through 7 features from the backbone network and repeatedly fuses these features in top-down and bottom-up directions. Both BiFPN layers and class/box layers are repeated multiple times with the number of repetations depending on the compund coefficient of the architecture. Fig. 1.9.21 provides and overview of the described structure. For further details please see the seminal work by Tan, Pang, and Le.

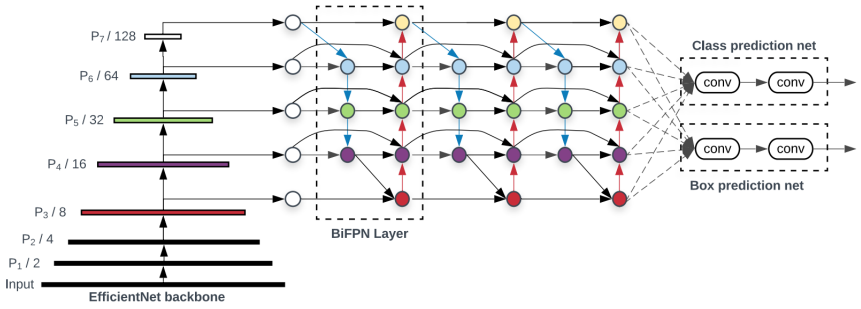


Fig. 1.9.21: A high-level representation of the EfficientDet architecture

Remarkable performance gains can be attained in image classification by jointly scaling up all dimensions of neural network width, depth, and input resolution, as noted in the study by Tan and Le. Inspired by this work, EfficienDet utilizes a new compound scaling method for object detection that jointly increases all dimensions of the backbone network, BiFPN, class/box network, and input image resolution, using a simple compound coefficient, . A total of 8 compounding levels are defined for EffcienDet, i.e., = 0 to 8, with EfficientDet-D0 being the simplest and EfficientDet-D8 being the most complex of the network architectures.

As shown in Fig. 1.9.22, at the time this work was published, EfficientDet object detection algorithms attained the state-of-the-art performance on the COCO dataset. Also suggested in Figure 3 is the more complex the network architecture is, the higher the detection performance will be. From a practical standpoint, however, architecture selection will depend on the availability of computational resources. For example, to train a model on an architecture with a compound coefficient higher than 4, a GPU with a memory of more than 11 GB will almost always be required.

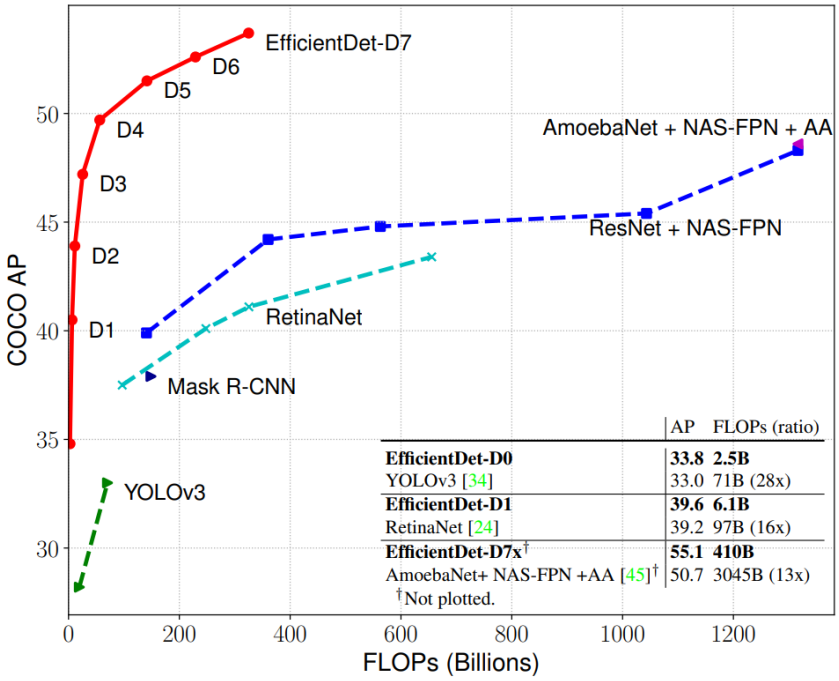


Fig. 1.9.22: A comparison of the performance and accuracy levels of EfficienDet models over other popular object detection architectures on the COCO dataset

1.10 Validation

This section provides the validation test results for the following ML modules:

1.10.1 Roof Classifier

The Roof Classifier's methodology has been presented in *Roof type classifier*, and examples showing how to use it can be found in *Roof Shape Classifier*. This section presents its validation against three datasets.

Dataset 1: Compare with OpenStreetMap Labels

The trained classifier is first tested on a ground truth dataset that can be downloaded from here. We firstly obtained a set of randomly selected buildings in the United States with their roof type labelled on OpenStreetMap. We then downloaded the satellite images from Google Maps for each building. We removed images in which we didn't clearly see there is a roof, examples are shown in Table 1.10.1.

Table 1.10.1: Images removed from the test dataset

The resulting dataset contains roof images in three categories: 32 flat, 40 gabled, 52 hipped. Examples of these satellite images can be found in *Roof type classifier*.

Run the following python script to test on this dataset.

```
import shutil
import os
import pandas as pd
from glob import glob
import wget
import zipfile

# download the testing dataset
wget.download('https://zenodo.org/record/4520781/files/satellite-images-val.zip')
with zipfile.ZipFile('satellite-images-val.zip', 'r') as zip_ref:
    zip_ref.extractall('.')

# get images
flatList = glob('satellite-images-val/flat/*.png')
gabledList = glob('satellite-images-val/gabled/*.png')
hippedList = glob('satellite-images-val/hipped/*.png')

# define the paths of images in a list
imgs=flatList+gabledList+hippedList

# import the module
from brails.modules import RoofClassifier

# initialize a roof classifier
roofModel = RoofClassifier()

# use the model to predict
predictions = roofModel.predict(imgs)

prediction = predictions['prediction'].values.tolist()
label = ['flat']*len(flatList) + ['gabled']*len(gabledList) + ['hipped
↪']*len(hippedList)

# lot results
class_names = ['flat','gabled', 'hipped']
from brails.utils.plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score

# print
print(' Accuracy is   : {}, Random guess is 0.33'.format(accuracy_score(prediction,
↪label)))
cnf_matrix = confusion_matrix(prediction,label)
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix',
↪normalize=True,xlabel='Labels',ylabel='Predictions')
```

The prediction accuracy on this dataset is 90.3%. Precision is 90.3%. Recall is 90.3%. F1 is 90.3%.

The confusion matrix for this validation is shown in Fig. 1.10.1.

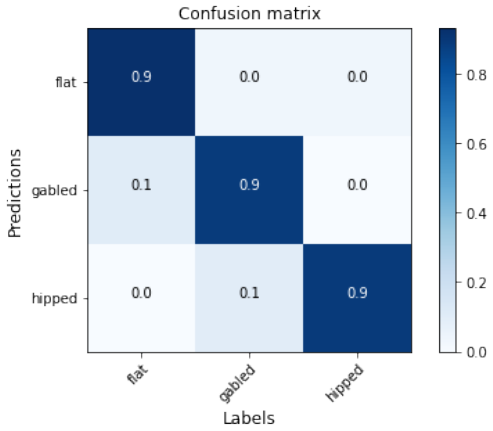


Fig. 1.10.1: Confusion matrix - Roof type classification for OpenStreetMap

Dataset 2: Compare with StEER Hurricane Laura Dataset

The second validation dataset is from StEER. This dataset contains satellite images of building, most are taken before Hurricane Laura.

From StEER, we obtained a list of addresses with their roof types labelled. For each address, we downloaded an satellite image from Google Maps Static API.

Examples of these satellite images can be found in *Roof type classifier*.

The labeling system of StEER is different from the BRAILS roof classification system. The StEER labels include the following classes:

- Gable/Hip Combo
- Hip
- Gable
- Complex
- Flat

The BRAILS roof types include the following classes:

- gabled
- hipped
- flat

To compare these two systems, we selected addresses labeled as 'Flat', 'Gable', 'Hip' from StEER. As a result we got the following numbers of StEER labels:

- hipped, 33
- gabled, 21
- flat , 2

Download the labels, images, scripts for this validation from here.

The following shows the script to run this validation. At the end, the script will plot a confusion matrix and print the accuracy.

```
# ### Load labels and predictions from The Lake Charles Testbed

import pandas as pd

data = pd.read_csv('StEER_Laura.csv')
data.describe()

data = data[(data['RoofShape(StEER)']=='Hip') |
            (data['RoofShape(StEER)']=='Gable') |
            (data['RoofShape(StEER)']=='Flat')]

roofDict = {'Gable':'gabled','Flat':'flat','Hip':'hipped'}
data['RoofShape(StEER)']=data['RoofShape(StEER)'].apply(lambda x: roofDict[x])
data['RoofShape(Testbed)']=data['RoofShape(Testbed)'].apply(lambda x: roofDict[x])


# ### Plot confusion matrix

import sys
sys.path.append(".")
from plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score,f1_score

class_names = list(data['RoofShape(Testbed)'].unique())

predictions = data['RoofShape(Testbed)']
labels = data['RoofShape(StEER)']

cnf_matrix = confusion_matrix(labels,predictions)
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix',
↪normalize=True,xlabel='BRAILS',ylabel='StEER')

for i,cname in enumerate(class_names):
    accuracy = '%.1f'%(cnf_matrix[i][i]/sum(cnf_matrix[i]))
    TP = cnf_matrix[i][i]
    FP = sum(cnf_matrix[:,i])-cnf_matrix[i,i]
    FN = sum(cnf_matrix[i,:])-cnf_matrix[i,i]
    F1 = '%.1f'%(TP/(TP+0.5*(FP+FN)))

    print(f'{cname}: Accuracy = {accuracy}, F1 = {F1}')


# ### Copy images to directories {label}-{prediction} for inspection

import os
import shutil

predDir = 'tmp/images/roof_predictions'
if not os.path.exists(predDir):
    os.makedirs(predDir)

falseNames = []
def copyfiles(bim):
    for ind, row in bim.iterrows():
        label = row['RoofShape(StEER)']
        pred = row['RoofShape(Testbed)']
```

(continues on next page)

(continued from previous page)

```
        lon, lat = '%.6f'%row['Longitude'], '%.6f'%row['Latitude']

        oldfile = f'tmp/images/TopView/TopViewx{lon}x{lat}.png'
        newfile = f'{predDir}/{label}-{pred}/TopViewx{lon}x{lat}.png'

        thisFileDir = f'{predDir}/{label}-{pred}/'
        if not os.path.exists(thisFileDir): os.makedirs(thisFileDir)

        try:
            shutil.copyfile(oldfile, newfile)
        except:
            print(oldfile)

copyfiles(data)
```

In the files you downloaded, there are folders with names like gabled-hipped, which means those are images that are labelled as 'gabled' in StEER dataset, but they are predicted as 'hipped'. You can browse those images to investigate deeper.

The confusion matrix tested on this dataset is shown in Fig. 1.10.2.

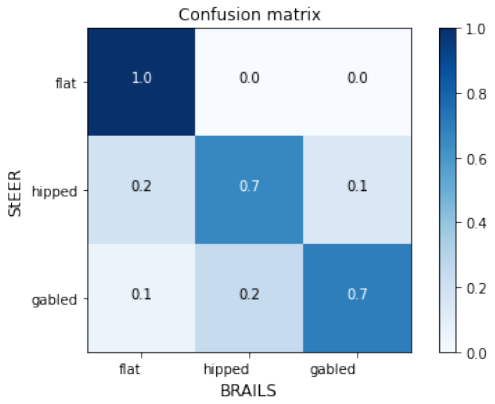


Fig. 1.10.2: Confusion matrix - Roof type classification for Hurricane Laura

The accuracy for three classes are:

- flat: Accuracy = 1.0, F1 = 0.4
- hipped: Accuracy = 0.7, F1 = 0.7
- gabled: Accuracy = 0.7, F1 = 0.8

Dataset 3: Compare with StEER Hurricane Irma Dataset with Post-disaster Images

The third validation dataset is also from StEER. This dataset contains satellite images of building that are taken after Hurricane Irma and labels of roof shapes.

From StEER, we obtained a list of addresses with their roof types labelled. For each address, we downloaded an satellite image from Google Maps Static API. It should be noted that these images were taken after the Hurricane Irma.

These post-disaster images are very different from the training dataset, in which all images are taken before the disasters. Keep in mind, the BRAILS roof model is not designed to recognize the post-disaster images. The aim of this validation is to see how BRAILS performs on these post-disaster images, though the model is not trained for this purpose.

Examples of these satellite images can be found in Table 1.10.2.

Table 1.10.2: Example post-hurricane satellite images

			
Fig. 1.10.3: A severely damaged building with its roof cover totally removed	Fig. 1.10.4: A gabled roof with a temporary fix by partial covering with a blue tarp	Fig. 1.10.5: A building removed from the land, leaving an empty lot	Fig. 1.10.6: A building with a hipped roof that was not damaged

The labeling system of StEER is different from the BRAILS roof classification system. The StEER labels include the following classes:

- Gable
- Hip
- Complex
- Flat
- Hip/Gable
- Hip,Complex
- Gable,Complex
- Other
- Gable,Flat
- Hip/Gable,Complex
- Hip,Flat
- Gambrel
- Hip,Gable
- Hip,Other

- Hip/Gable,Flat

The BRAILS roof types include the following classes:

- gabled
- hipped
- flat

To compare these two systems, we selected addresses labeled as 'Flat', 'Gable', 'Hip' from StEER. As a result we got the following numbers of StEER labels:

- gabled, 459
- hipped, 180
- flat, 72

Download the labels, images, scripts for this validation from here.

The following shows the script to run this validation. It will use BRAILS to download images from Google Map Static API and perform predictions on these images. At the end, the script will plot a confusion matrix and print the accuracy.

```
import pandas as pd
data = pd.read_csv("Irma_validation.csv")
data.describe()


# ### Use BRAILS to download satellite images

# You don't have to download again, it's already included in this example
# But feel free to uncomment and download again. You need a Google API Key, and set␣
↪reDownloadImgs=True
'''
from brails.workflow.Images import getGoogleImagesByAddrOrCoord

addrs =  list(data['Addr'])
getGoogleImagesByAddrOrCoord(Addrs=addrs, GoogleMapAPIKey='Your-Key',
                             imageTypes=['TopView'],imgDir='tmp/images',ncpu=2,
                             fov=60,pitch=0,reDownloadImgs=False)
'''
data['TopViewImg']='tmp/images/TopView/TopViewx'+data['Addr'].str.replace(' ','-')+'.
↪png'


# ### Predict

from brails.modules import RoofClassifier
roofModel = RoofClassifier()
roofPreds = roofModel.predict(list(data['TopViewImg']))

data['RoofShape(BRAILS)']=list(roofPreds['prediction'])
data['prob_RoofShape(BRAILS)']=list(roofPreds['probability'])
roofDict = {'Gable':'gabled','Flat':'flat','Hip':'hipped'}
data['RoofShape(StEER)']=data['roof_shape'].apply(lambda x: roofDict[x])


# ### Plot confusion matrix

import sys
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
sys.path.append(".")

from plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score,f1_score

class_names = list(data['RoofShape(BRAILS)'].unique())

predictions = data['RoofShape(BRAILS)']
labels = data['RoofShape(StEER)']

cnf_matrix = confusion_matrix(labels,predictions,labels=class_names)
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,xlabel='BRAILS',
↪ylabel='StEER')


for i,cname in enumerate(class_names):
    accuracy = '%.1f'%(cnf_matrix[i][i]/sum(cnf_matrix[i]))
    TP = cnf_matrix[i][i]
    FP = sum(cnf_matrix[:,i])-cnf_matrix[i,i]
    FN = sum(cnf_matrix[i,:])-cnf_matrix[i,i]
    F1 = '%.1f'%(TP/(TP+0.5*(FP+FN)))

    print(f'{cname}: Accuracy = {accuracy}, F1 = {F1}')



# ### Copy images to directories {label}-{prediction} for inspection

import os
import shutil

if not os.path.exists('tmp/images/roof_predictions/'):
    os.makedirs('tmp/images/roof_predictions/')

for ind, row in data.iterrows():
    addrstr = row['Addr'].replace(' ','-')
    picname = f'tmp/images/TopView/TopViewx{addrstr}.png'

    label = row['RoofShape(StEER)']
    pred = row['RoofShape(BRAILS)']
    if True:
        thisFileDir = f'tmp/images/roof_predictions/{label}-{pred}/'
        if not os.path.exists(thisFileDir):
            os.makedirs(thisFileDir)
        shutil.copyfile(picname, thisFileDir+picname.split('/')[-1])
```

In the files you downloaded, there are folders with names like gabled-hipped, which means those are images that are labelled as 'gabled' in StEER dataset, but they are predicted as 'hipped'. You can browse those images to investigate deeper.

The confusion matrix tested on this dataset is shown in Fig. 1.10.7.

The accuracy for three classes are:

- gabled: Accuracy = 0.2, F1 = 0.3

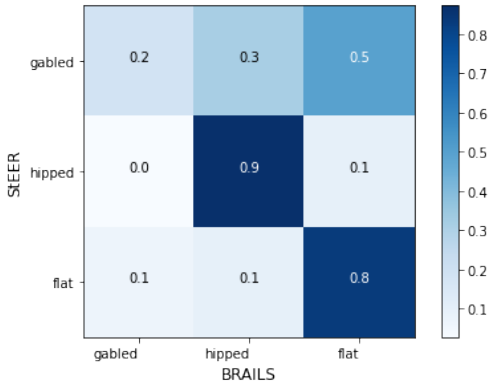
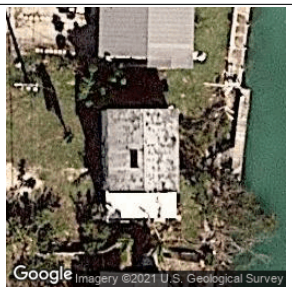


Fig. 1.10.7: Confusion matrix - Roof type classification for Hurricane Irma

- hipped: Accuracy = 0.9, F1 = 0.6
- flat: Accuracy = 0.8, F1 = 0.3

Examples of false predictions are shown in Table 1.10.3.

Table 1.10.3: Examples of false predictions

 Fig. 1.10.8: Label: Gabled, BRAILS Prediction: Flat	 Fig. 1.10.9: Label: Gabled, BRAILS Prediction: Flat	 Fig. 1.10.10: Label: Gabled, BRAILS Prediction: Hipped	 Fig. 1.10.11: Label: Gabled, BRAILS Prediction: Hipped

It shows the accuracy for the gable is not as high as the other classes. A further look into the images we found the following facts:

1. Most roofs that are labelled as 'hipped' are not damaged during the hurricane, examples are in Table 1.10.4.
2. Roofs labelled as 'flat' and 'gabled' are the classes got most damages, examples are in Table 1.10.5 and Table 1.10.6.
3. When roofs are damaged, their satellite images are different from pre-disaster images, examples are shown in Table 1.10.7, This could cause difficulties to the model to recognize features.
4. This validation doesn't remove those images with an empty lot, which negatively influences the accuracy.
5. Bias in dataset is very common. This validation doesn't consider the possible bias in the StEER labels (examples can be found in Table 1.10.8), which also negatively influences the accuracy.

Table 1.10.4: Examples of post-hurricane satellite images: hipped

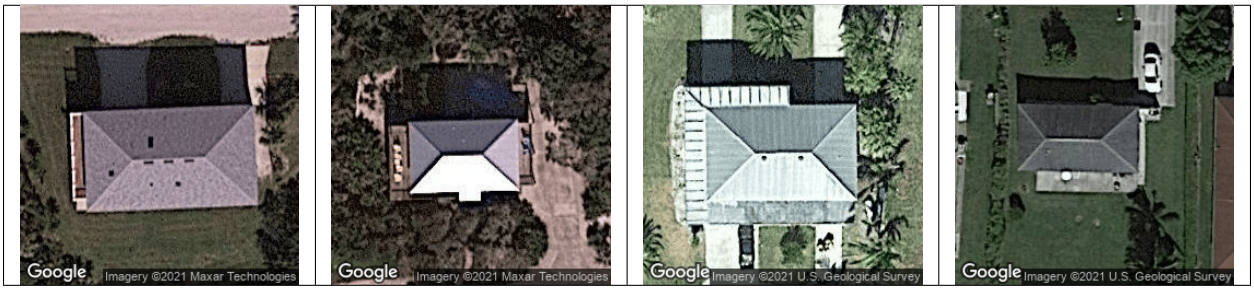


Table 1.10.5: Examples of post-hurricane satellite images: flat

Table 1.10.6: Examples of post-hurricane satellite images: gabled

Table 1.10.7: Examples of post-hurricane satellite images: severely damaged buildings / empty lot

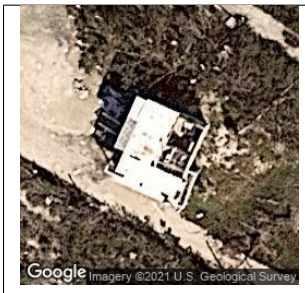
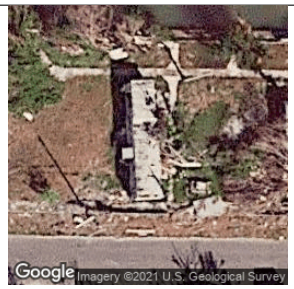
 Fig. 1.10.12: A severely damaged roof	 Fig. 1.10.13: A severely damaged roof	 Fig. 1.10.14: A severely damaged roof	 Fig. 1.10.15: A severely damaged roof
 Fig. 1.10.16: An empty lot	 Fig. 1.10.17: An empty lot	 Fig. 1.10.18: An empty lot	 Fig. 1.10.19: An empty lot

Table 1.10.8: Example of post-hurricane satellite images: Bias in the labels

 Fig. 1.10.20: Label: Flat, BRAILS Prediction: Gabled	 Fig. 1.10.21: Label: Flat, BRAILS Prediction: Gabled	 Fig. 1.10.22: Label: Gabled, BRAILS Prediction: Flat	 Fig. 1.10.23: Label: Gabled, BRAILS Prediction: Flat
 Fig. 1.10.24: Label: Gabled, BRAILS Prediction: Hipped	 Fig. 1.10.25: Label: Gabled, BRAILS Prediction: Hipped	 Fig. 1.10.26: Label: Hipped, BRAILS Prediction: Flat	 Fig. 1.10.27: Label: Hipped, BRAILS Prediction: Gabled

1.10.2 Occupancy Classifier

The Occupancy Classifier's methodology has been presented in *Occupancy classifier*, and examples showing how to use it can be found in *Occupancy Classifier*. This section presents its validation against two datasets.

Dataset 1: Compare with OpenStreetMap Labels

The trained classifier is tested on a ground truth dataset that can be downloaded from here. We firstly obtained a set of randomly selected buildings in the United States with occupancy tags found on OpenStreetMap. We then downloaded the street view images from Google Street View for each building. We removed images in which we didn't clearly see there is a building. The dataset contains 98 single family buildings (RES1), 97 multi-family buildings (RES3) and 98 commercial buildings (COM). Examples of these street view images can be found in *Occupancy Classifier*.

Run the following python script to test on this dataset.

```
# download the testing dataset

import wget
import zipfile
wget.download('https://zenodo.org/record/4553803/files/occupancy_validation_images.zip
↪')
with zipfile.ZipFile('occupancy_validation_images.zip', 'r') as zip_ref:
    zip_ref.extractall('.')

# prepare the image lists

import shutil
import os
import pandas as pd
from glob import glob

class_names = ['RES3', 'COM' ,'RES1']

labels = []
images = []
for clas in class_names:
    imgs = glob(f'occupancy_validation_images/{clas}/*.jpg')
    for img in imgs:
        labels.append(clas)
        images.append(img)

# import the module
from brails.modules import OccupancyClassifier

# initialize the classifier
occupancyModel = OccupancyClassifier()

# use the model to predict
pred = occupancyModel.predict(images)
predictions = pred['prediction'].tolist()

# Plot results
from brails.utils.plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score

print(' Accuracy is   : {}, Random guess is 0.33'.format(accuracy_score(predictions,
↪labels)))
```

(continues on next page)

(continued from previous page)

```
cnf_matrix = confusion_matrix(predictions,labels)
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix',
↪normalize=False,xlabel='Labels',ylabel='Predictions')
```

The confusion matrix tested on this dataset is shown in Fig. 1.10.28.

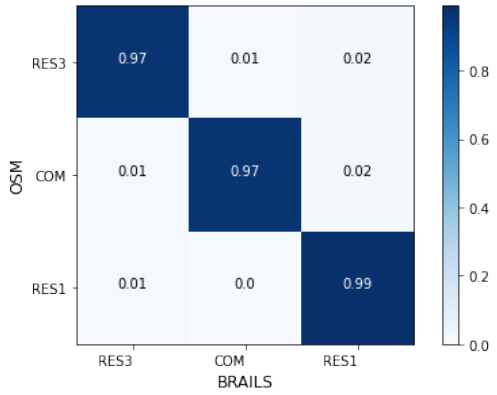


Fig. 1.10.28: Confusion matrix - Occupancy Class classifier

The accuracy for the two classes are:

- RES3: Accuracy = 0.97, F1 = 0.97
- COM: Accuracy = 0.97, F1 = 0.98
- RES1: Accuracy = 0.99, F1 = 0.97

Dataset 2: Compare with NJDEP Dataset

The second validation dataset is from New Jersey Department of Environmental Protection (NJDEP).

NJDEP developed a building inventory for flood hazard and risk analysis as part of its flood control and resilience mission. In this dataset, we can find building footprints with their occupancy types labelled. We randomly selected a subset of those records, for each we downloaded a street view image from Google Maps Static API.

Examples of these satellite images can be found in *Occupancy classifier*.

The NJDEP occupancy data includes the following labels:

- RES1 26574
- RES3A 1714
- COM1 1110
- RES3B 1016
- RES3C 779
- RES3D 566
- COM8 187
- AGR1 113
- RES4 111
- COM4 100

- GOV1 90
- IND2 83
- COM3 74
- REL1 67
- RES3E 52
- EDU1 48
- IND3 37
- GOV2 24
- COM7 16
- RES3F 15
- IND1 13
- EDU2 11
- IND4 11
- IND5 6
- COM2 3
- COM10 3
- COM6 2
- IND6 2
- COM5 1

The BRAILS occupancy system include the following classes:

- RES1
- RES3
- COM

To compare these two systems, we renamed some NJDEP labels:

- RES1 -> RES1
- RES3A -> RES3
- RES3B -> RES3
- RES3C -> RES3
- RES3D -> RES3
- RES3F -> RES3
- RES3E -> RES3
- COM1 -> COM
- COM2 -> COM
- COM3 -> COM
- COM4 -> COM
- COM5 -> COM

- COM6 -> COM
- COM7 -> COM
- COM8 -> COM
- COM10 -> COM

From the relabelled records, we selected the following for validation:

- RES1, 1,000 randomly selected from RES1
- RES3, 1,000 randomly selected from RES3
- COM, 1,000 randomly selected from COM

You can download the labels, images, scripts for this validation from here.

The following shows the script to run this validation. At the end, the script will plot a confusion matrix and print the accuracy.

```
import pandas as pd
data = pd.read_csv("AtlanticCountyBuildingInventory.csv")
data.describe()

def getCls(x):
    if 'RES1' in x:
        return 'RES1'
    elif 'RES3' in x:
        return 'RES3'
    elif 'COM' in x:
        return 'COM'
    else: return 'remove'

data['occupancy']=data['OccupancyClass'].apply(lambda x: getCls(x))

#data=data[data['occupancy']!='remove']
RES1 = data[data['occupancy']=='RES1'].sample(n=1000, random_state = 1993)
RES3 = data[data['occupancy']=='RES3'].sample(n=1000, random_state = 1993)
COM = data[data['occupancy']=='COM'].sample(n=1000, random_state = 1993)
data = pd.concat([RES1,RES3,COM])



# ### Use BRAILS to download street view images

import sys
sys.path.append("/Users/simcenter/Codes/SimCenter/BIM.AI")
from brails.workflow.Images import getGoogleImagesByAddrOrCoord

addrs =  list(data[['Longitude','Latitude']].to_numpy())
getGoogleImagesByAddrOrCoord(Addrs=addrs, GoogleMapAPIKey='Your-Key',
                             imageTypes=['StreetView'],imgDir='tmp/images',ncpu=2,
                             fov=60,pitch=0,reDownloadImgs=False)


data['StreetViewImg']=data.apply(lambda row: f"tmp/images/StreetView/StreetViewx{'%.6f
↪'%row['Longitude']}x{'%.6f'%row['Latitude']}.png", axis=1)

import os
import shutil
```

(continues on next page)

(continued from previous page)

```
# Remove empty images
data = data[data['StreetViewImg'].apply(lambda x: os.path.getsize(x)/1024 > 9)]
# Remove duplicates
data.drop_duplicates(subset=['StreetViewImg'], inplace=True)


# ### Predict

from brails.modules import OccupancyClassifier
occupancyModel = OccupancyClassifier()
occupancyPreds = occupancyModel.predict(list(data['StreetViewImg']))


data['Occupancy(BRAILS)']=list(occupancyPreds['prediction'])
data['prob_Occupancy(BRAILS)']=list(occupancyPreds['probability'])


# ### Plot confusion matrix

import sys
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
sys.path.append(".")

from plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score,f1_score

class_names = list(data['Occupancy(BRAILS)'].unique())

predictions = data['Occupancy(BRAILS)']
labels = data['occupancy']

cnf_matrix = confusion_matrix(labels,predictions,labels=class_names)
plot_confusion_matrix(cnf_matrix, classes=class_names, normalize=True,xlabel='BRAILS',
↪ylabel='NJDEP')


for i,cname in enumerate(class_names):
    accuracy = '%.1f'%(cnf_matrix[i][i]/sum(cnf_matrix[i]))
    TP = cnf_matrix[i][i]
    FP = sum(cnf_matrix[:,i])-cnf_matrix[i,i]
    FN = sum(cnf_matrix[i,:])-cnf_matrix[i,i]
    F1 = '%.1f'%(TP/(TP+0.5*(FP+FN)))

    print(f'{cname}: Accuracy = {accuracy}, F1 = {F1}')


# ### Copy images to directories {label}-{prediction} for inspection

import os
import shutil

predDir = 'tmp/images/occupancy_predictions'
if not os.path.exists(predDir):
    os.makedirs(predDir)
```

(continues on next page)

(continued from previous page)

```
falseNames = []
def copyfiles(bim):
    for ind, row in bim.iterrows():
        label = row['occupancy']
        pred = row['Occupancy(BRAILS)']

        lon, lat = '%.6f'%row['Longitude'], '%.6f'%row['Latitude']

        oldfile = f'tmp/images/StreetView/StreetViewx{lon}x{lat}.png'
        newfile = f'{predDir}/{label}-{pred}/StreetViewx{lon}x{lat}.png'

        thisFileDir = f'{predDir}/{label}-{pred}/'
        if not os.path.exists(thisFileDir): os.makedirs(thisFileDir)

        try:
            shutil.copyfile(oldfile, newfile)
        except:
            print(oldfile)

copyfiles(data)
```

In the files you downloaded, there are folders with names like RES-COM, which means those are images that are labelled as 'RES' in NJDEP dataset, but they are predicted as 'COM'. You can browse through those images to investigate deeper.

The confusion matrix tested on this dataset is shown in Fig. 1.10.29.

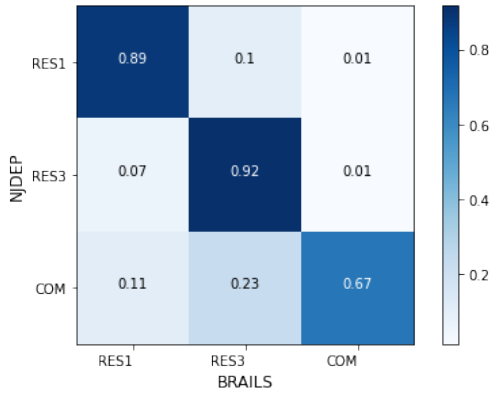


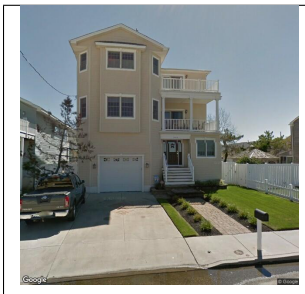
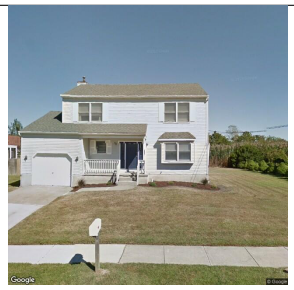
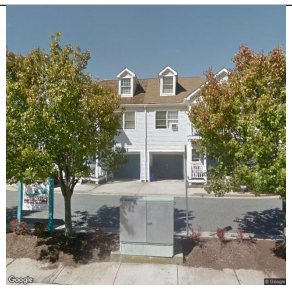
Fig. 1.10.29: Confusion matrix - Occupancy type classification for NJDEP

The accuracy for the two classes are:

- RES1: Accuracy = 0.89, F1 = 0.86
- RES3: Accuracy = 0.92, F1 = 0.83
- COM: Accuracy = 0.67, F1 = 0.79

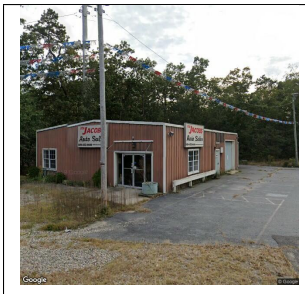
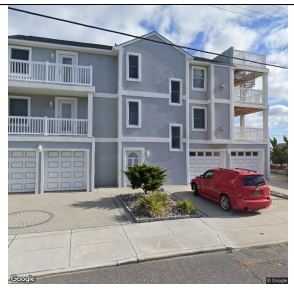
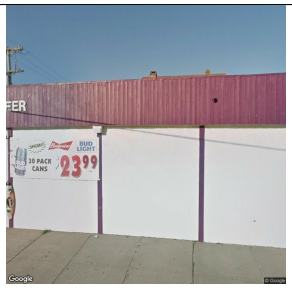
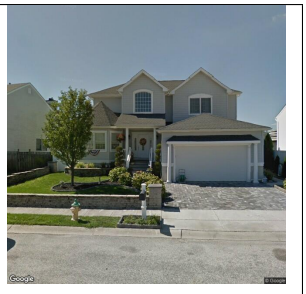
Examples of false predictions are shown in Table 1.10.9.

Table 1.10.9: Example of false predictions

 Fig. 1.10.30: Label: RES1, BRAILS Prediction: RES3	 Fig. 1.10.31: Label: RES1, BRAILS Prediction: RES3	 Fig. 1.10.32: Label: RES3, BRAILS Prediction: COM	 Fig. 1.10.33: Label: RES3, BRAILS Prediction: COM

Note: Bias in dataset is very common. This validation doesn't consider the possible bias in the labels (examples can be found in Table 1.10.10), which also negatively influences the accuracy.

Table 1.10.10: Example of street view images: Bias in the labels

 Fig. 1.10.34: Label: RES1, BRAILS Prediction: COM	 Fig. 1.10.35: Label: RES1, BRAILS Prediction: RES3	 Fig. 1.10.36: Label: RES3, BRAILS Prediction: COM	 Fig. 1.10.37: Label: RES3, BRAILS Prediction: RES1

1.10.3 Soft-story Building Classifier

The Soft-story Building Classifier is validated here.

The trained classifier is tested on a ground truth dataset that can be downloaded here. Accuracy is 83.8%. Precision is 83.8%. Recall is 83.8%. F1 is 83.8%.

Run the following python script to test on this dataset.

```
# download the testing dataset

import wget
import zipfile
wget.download('https://zenodo.org/record/4508433/files/softstory-buildings-val.zip')
with zipfile.ZipFile('softstory-buildings-val.zip', 'r') as zip_ref:
    zip_ref.extractall('.')

# prepare the image lists
```

(continues on next page)

(continued from previous page)

```
import shutil
import os
import pandas as pd
from glob import glob

softstoryList = glob('softstory-buildings-val/softstory/*.png')
othersList = glob('softstory-buildings-val/others/*.png')

# define the paths of images in a list
imgs=softstoryList+othersList

# import the module
from brails.modules import SoftstoryClassifier

# initialize the classifier
model = SoftstoryClassifier()

# use the model to predict
predictions = model.predict(imgs)

prediction = predictions['prediction'].values.tolist()
label = ['softstory']*len(softstoryList) + ['others']*len(othersList)

# Plot results
class_names = ['softstory','others']
from brails.utils.plotUtils import plot_confusion_matrix
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score,accuracy_score

print(' Accuracy is   : {}, Random guess is 0.5'.format(accuracy_score(prediction,
↪label)))
cnf_matrix = confusion_matrix(prediction,label)
plot_confusion_matrix(cnf_matrix, classes=class_names, title='Confusion matrix',
↪normalize=True,xlabel='Labels',ylabel='Predictions')
```

The confusion matrix tested on this dataset is shown in Fig. 1.10.38.

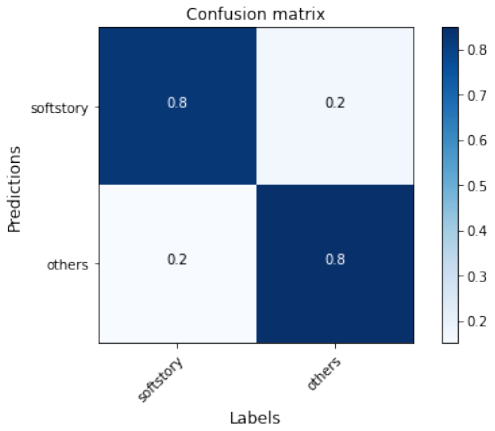


Fig. 1.10.38: Confusion matrix - Soft-story building classifier

1.10.4 Number of Floors Detector

On a randomly selected set of in-the-wild building images from New Jersey's Bergen, Middlesex, and Moris Counties, the model attains an F1-score of 86%. Here, in-the-wild building images are defined as street-level photos that may contain multiple buildings and are captured with random camera properties. `confusion_nFloorWildv2` is the confusion matrix of the model inferences on the aforementioned in-the-wild test set.

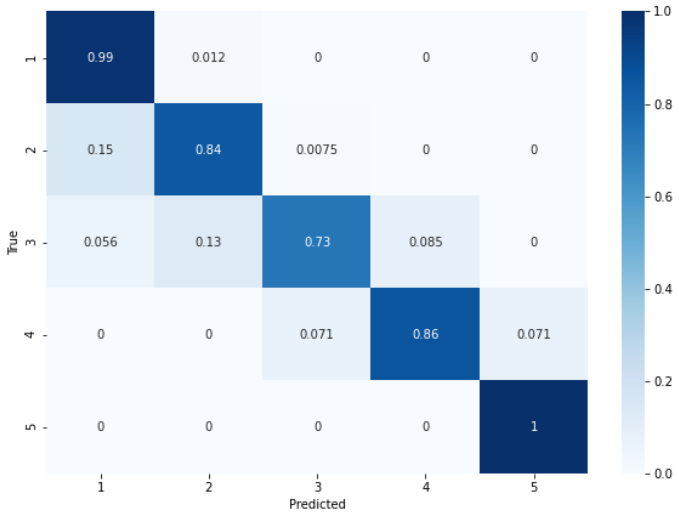


Fig. 1.10.39: Confusion matrix of the pretrained model on the in-the-wild test set

If the test images are constrained such that a single building exists in each image, the building is viewed with minimal obstructions, and the images are captured such that the image plane is nearly parallel to the frontal plane of the building facade, the F1-score of the model is determined as 94.7%. `confusion_nFloorClean` shows the confusion matrix for the pretrained model on a test set generated according to these constraints.

Table **??** shows a sample of images removed from the in-the-wild test set that were found to display weak resemblance of the visual cues necessary for a valid number of floor predictions.

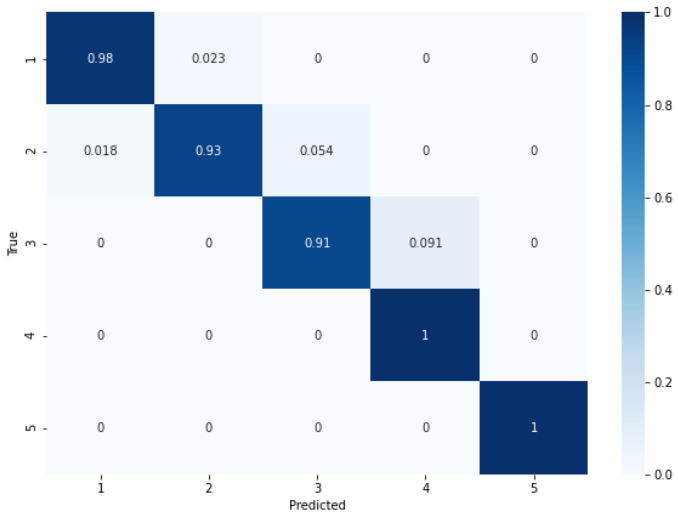


Fig. 1.10.40: Confusion matrix of the pretrained model on the dataset containing lightly distorted/obstructed images of individual buildings

Table 1.10.11: In-the-wild street level imagery removed as a part of dataset cleaning

Fig. 1.10.41: Heavily occluded building facade

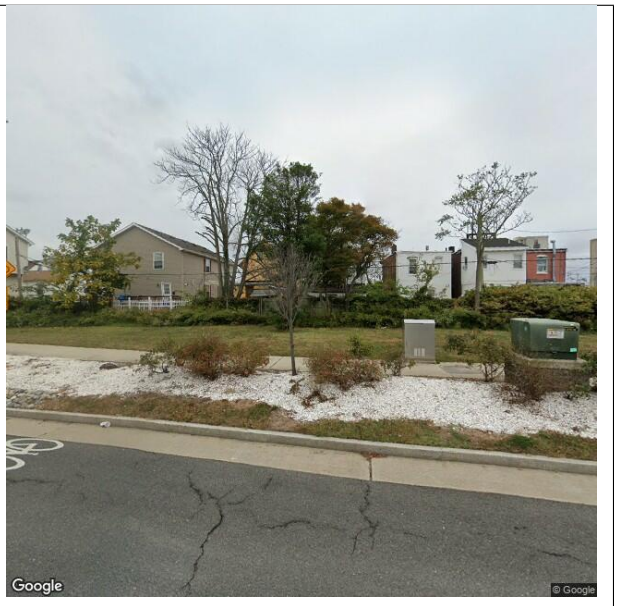
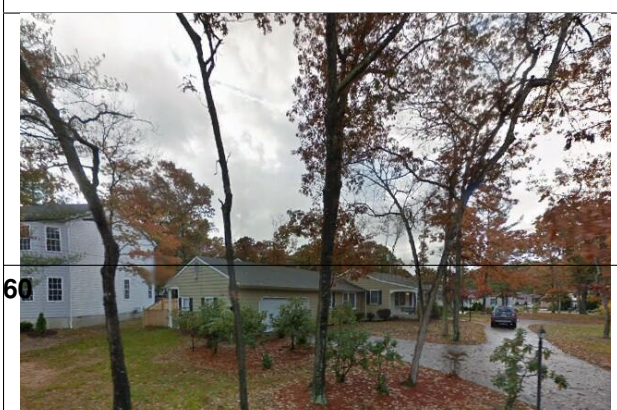


Fig. 1.10.42: Closely spaced buildings: obscure prediction target



Note:

1. Year Built Classifier is currently under active development and testing. More details about the training, modification, improvement, and validation of this module can be found here.
2. Raised Foundation Classifier is currently under active development and testing. More details about the training, modification, improvement, and validation of this module can be found here.

Note: DISCLAIMER: The modules are implemented to demonstrate the potentials of ML methods to help establish building attributes and inventories for regional scale simulation. The modules are tested extensively using the data sets as reported herein for validations. How these modules generalize to new and unseen data from different geographical locations depends on how similar they are to the training data. Generalization of machine learning models remains an active research area. Users should exercise cautions when the modules are used beyond their intended purposes and trained model ability.

1.11 How to Extend

The framework is being developed and maintained by the author.

The developer manual will be provided here if community contributions are needed in the future.

1.12 Coding Style

1.12.1 Python Style

For code written in Python SimCenter programmers follow the widely used Guide PEP 8

CHAPTER

TWO

HOW TO CITE

Charles Wang, Sascha Hornauer, Barbaros Cetiner, Yunhui Guo, Frank McKenna, Qian Yu, Stella X. Yu, Ertugrul Taciroglu, & Kincho H. Law. (2021, March 1). NHERI-SimCenter/BRAILS: Release v2.0.0 (Version v2.0.0). Zenodo. http://doi.org/10.5281/zenodo.4570554

CHAPTER

THREE

CONTACT

NHERI-SimCenter nheri-simcenter@berkeley.edu

CHAPTER

FOUR

REFERENCES

BIBLIOGRAPHY

[BHF+19] Benjamin Bischke, Patrick Helber, Joachim Folz, Damian Borth, and Andreas Dengel. Multi-task learning for segmentation of building footprints with deep neural networks. *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1480–1484, 2019.

[Goo97] Pierre Goovaerts. *Geostatistics for natural resources evaluation*. Oxford University Press on Demand, 1997.

[HW68] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[LHF+19] Weijia Li, Conghui He, Jiarui Fang, Juepeng Zheng, Haohuan Fu, and Le Yu. Semantic segmentation-based building footprint extraction using very high-resolution satellite images and multi-source gis data. *Remote Sensing*, 11(4):403, 2019.

[Mic] Microsoft. US Building Footprints. URL: https://github.com/microsoft/USBuildingFootprints.

[Van10] Erik Vanmarcke. *Random fields: analysis and synthesis*. World Scientific, 2010.

[WC18] C Wang and Q Chen. A hybrid geotechnical and geological data-based framework for multiscale regional liquefaction hazard mapping. *Géotechnique*, 68(7):614–625, 2018.

[WCSJ17] Chaofeng Wang, Qiushi Chen, Mengfen Shen, and C Hsein Juang. On the spatial variability of cpt-based geotechnical parameters for regional liquefaction evaluation. *Soil Dynamics and Earthquake Engineering*, 95:153–166, 2017.

[ZKJS18] Kang Zhao, Jungwon Kang, Jaewook Jung, and Gunho Sohn. Building extraction from satellite images using mask r-cnn with building boundary regularization. *CVPR Workshops*, pages 247–251, 2018.

[BHF+19] Benjamin Bischke, Patrick Helber, Joachim Folz, Damian Borth, and Andreas Dengel. Multi-task learning for segmentation of building footprints with deep neural networks. *2019 IEEE International Conference on Image Processing (ICIP)*, pages 1480–1484, 2019.

[Goo97] Pierre Goovaerts. *Geostatistics for natural resources evaluation*. Oxford University Press on Demand, 1997.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778. 2016.

[HW68] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[LHF+19] Weijia Li, Conghui He, Jiarui Fang, Juepeng Zheng, Haohuan Fu, and Le Yu. Semantic segmentation-based building footprint extraction using very high-resolution satellite images and multi-source gis data. *Remote Sensing*, 11(4):403, 2019.

[Mic] Microsoft. US Building Footprints. URL: https://github.com/microsoft/USBuildingFootprints.

[SVI+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[Van10] Erik Vanmarcke. *Random fields: analysis and synthesis*. World Scientific, 2010.

[WC18] C Wang and Q Chen. A hybrid geotechnical and geological data-based framework for multiscale regional liquefaction hazard mapping. *Géotechnique*, 68(7):614–625, 2018.

[Wan20] Chaofeng Wang. Occupancy test. December 2020. URL: https://doi.org/10.5281/zenodo.4386991, doi:10.5281/zenodo.4386991.

[WCSJ17] Chaofeng Wang, Qiushi Chen, Mengfen Shen, and C Hsein Juang. On the spatial variability of cpt-based geotechnical parameters for regional liquefaction evaluation. *Soil Dynamics and Earthquake Engineering*, 95:153–166, 2017.

[Wan19] Charles Wang. Random satellite images of buildings. October 2019. URL: https://doi.org/10.5281/zenodo.3521067, doi:10.5281/zenodo.3521067.

[WYM+19] Charles Wang, Qian Yu, Frank McKenna, Barbaros Cetiner, Stella X. Yu, Ertugrul Taciroglu, and Kincho H. Law. Nheri-simcenter/brails: v1.0.1. October 2019. URL: https://doi.org/10.5281/zenodo.3483208, doi:10.5281/zenodo.3483208.

[ZKJS18] Kang Zhao, Jungwon Kang, Jaewook Jung, and Gunho Sohn. Building extraction from satellite images using mask r-cnn with building boundary regularization. *CVPR Workshops*, pages 247–251, 2018.