City Wide Energy Footprinting

Peter Wei

September 2018

Listings

1 Server

1.1 ICSL Server

The icsl server can be accessed by running the following:
`ssh icsl@icsl.ee.columbia.edu`
Password: `I*csl`
The Web.py server in Section 1.3 is located in the directory `citywidefootprinting`.

1.2 Static Webpages

Web pages can be hosted by the icsl server. html files are in directory `static/`.
The current page to view the traffic camera model is `icsl.ee.columbia.edu:8001/static/index.html`.

1.3 Web.py Server

```
class MyApplication():
    def run(self, port, middleware):

    def runDynamicPopulation(self):
```

Listing 1: energyServer.py

The software is run on a web.py server. There are two main commands for starting the web server.

```
./python_app restart
./python_app stop
./python_app status
```

The bash script python_app calls the python file run.py, which in turn runs the server in energyServer.py. The class MyApplication() includes the standard web.py run() method for running a webserver.

2 Population Baseline

2.1 Main routine

```
class showDynamicPopulation:
    def init(self):

    def CT2EUI(self):

    def getBlocks2Occupancy(self):
```

Listing 2: dynamicPopulation.py

The run() method instantiates the showDynamicPopulation class, which is the main routine that is run every 30 seconds.

init(self): is the initialization function that is called once, and initializes every data processing class (Section 2.2).

CT2EUI(self): is an inapporpriately named method for synthesizing two dictionaries, namely BBL2CT and BBL2EUI (self.energyDictionary), into a new dictionary CT2EUI. Borough Block Lot is switched with Census Tract in order to allow for plotting of block levels. EUI stands for Energy Usage Intensity. This method will probably be replaced in the future.

getBlocks2Occupancy(self): is a method for creating a dictionary (self.block2Occupancy) which describes the most recent change in population. This method can be called often to update the population counts.

2.2 Data Classes

Currently there are 4 sources of data: MTA stream (subway), historical subway turnstile data, building attribute data, and building energy data.

2.2.1 MTA Stream Data

```
class subwayStream:
    def __init__(self):

    def stationDefinitions(self, stationFile):
```

```python
    def getData(self, lines):
```

Listing 3: subwayStream.py

This data class receives streamed data from the MTA lines. This information mostly consists of location of the subway trains, future stops and times.

The stationDefinitions method parses a csv file containing the stop IDs referred to in the stream (such as 103N) and converts to the parent station (such as 103).

The getData method calls the MTA API for each subway feed (there are 9 feeds). The method returns a dictionary where the keys are the stations, and the value is the number of trains that have passed the station since the last call to getData.

2.3 Energy Data

```python
class loadEnergy:
    def __init__(self):

    def loadLL84(self, LL84File):
```

Listing 4: loadEnergy.py

This class parses energy data from New York City's Local Law 84 dataset. The dataset contains energy consumption information for buildings over 50,000 square feet. The loadLL84 method returns a dictionary self.energyDictionary, which is a key:value pair of the Borough Block Lot:Energy Usage Intensity.

2.4 Subway Turnstile Data

```python
class remoteDictionary:
    def __init__(self):

    def loadRemote2Station(self, remoteKeyFile):

    def loadStation2Coordinates(self, coordinatesFile):

    def loadTurnstile(self, turnstileFile):

    def createLegend(self):
```

Listing 5: Remote2StopID.py

Currently the most complex of the data processing classes. The purpose of this class is to produce two dictionaries: self.timeSeriesDataEntries and self.timeSeriesDataExits. The keys for both dictionaries are the station IDs, and the values are the counts for number of entries and exits, respectively, through the turnstiles of the station.

The turnstile dataset is formatted as a remote, and the current count of entries/exits. The remote is a unique identifier for the specific turnstile or set of turnstiles. Unfortunately, there seems to be no direct translation between the remote ID and the station ID.

The first method, createLegend, is a dictionary hard coded (by me) to translate the abbreviated names of the stations to the standard station ID.

The second method loadRemote2Station creates a dictionary with the remote ID defined in the remote file (TurnstileData/Remote-Booth-Station-3.csv) to the station ID from createLegend.

Finally, loadTurnstile takes the turnstile file (new file is uploaded to MTA website every week), and creates the time series dictionaries.

This class also has as method loadStation2Coordinates which creates a dictionary of station to (latitude, longitude) coordinates. This method is called in the init function, and is used later in the building data class.

2.5 Building Data

```
class buildingData:
    def __init__(self):

    def loadPLUTO(self, PLUTOfile, name):

    def loadCSV(self, PLUTOfile):

    def closestStation(self, stationCoordinates, borough, name):

    def station2Blocks(self):

    def saveObj(self, obj, name):

    def loadObj(self, name):
```

Listing 6: buildingData.py

This data class contains a number of methods for determining closest station to a building.

loadPLUTO and loadCSV use the PLUTO datasets for each borough to create two dictionaries: block2building, which is a key:value pair of the block (Census Tract) and a list of coordinates of the buildings in the block. The second dictionary, BBL2CT, is a key:value pair of the Borough Block Lot to Census Tract.

closestStation takes the stationCoordinates from the Remote2StopID class, and calculates the closest station to each block according to Euclidean distance. The limit for maximum distance to a station is 5 miles; thus, some blocks will not have a closest station if there is no station within that distance. This method is the most computationally intense, and thus pickle is used to save and load the dictionary for quick recovery. The return dictionary is self.nearestStationDictionary.

station2Blocks is a quick method for inverting the self.nearestStationDictionary.

2.6 Map Plotting

```
class plotNYCblocks:
    def __init__(self, EUI, borough):

    def clearPopulation(self):

    dynamicPopulation(self, newPopulation):

    def loadCensusData(self, borough, blockFile):

    def maxBlock(self):

    def minBlock(self):

    def maxBlockEnergy(self):

    def instantiateFigure(self):

    def drawBoroughs(self, boroughFile):

    def drawBuildings(self, buildingFile):

    def drawBlocks(self, blockFile):

    def drawSubwayLines(self, subwayLinesFile):

    def drawSubwayStations(self, subwayStationsFile):

    def plotGraph(self):
```

Listing 7: plotNYCblocks.py

This class is a mess. The base population is first loaded using the loadCensusData method, which takes the population count data from the 2010 US Census at the block level. This layer serves as the base population of the map, and is stored in self.PopulationDictionary.

dynamicPopulation is a method for adding and removing population from the self.PopulationDictionary. For example, if new real-time subway data has arrived, the populations will change; dynamicPopulation must be called in the case. If starting over, clearPopulation is a method for clearing the self.PopulationDictionary.

The remaining methods are for drawing maps. For drawing a map, call the instantiateFigure method first, and the PlotGraph method last. In between, any number of the drawing functions can be called; they will be drawn one after the other.

The first drawing function, usually called first, is the drawBoroughs method. This method draws the outline of the boroughs from the borough shapefiles.

drawBlocks draws blocks from the block shapefile, and colors the different blocks according to self.populationDictionary. I haven't yet determined a robust way to match the colors in drawBlocks and in plotGraph (for the legend), so this will need to be changed later.

drawSubwayLines draws the subway lines except for the staten island line. The drawSubwayStations method draws a dot at the positions of the subway stations.

drawBuildings is a method similar to drawBlocks, but for the building shapefile; this method takes forever due to the number of buildings in NYC. Use as a last resort.

There are a number of example runs to show how to draw different maps. The can be called from dynamicPopulation.py, as in the short methods at the bottom of that class.

3 Car Counting

This directory is for counting traffic.

3.1 TensorFlow

The methodology for training a TensorFlow model can be found here: `https://becominghuman.ai/tensorflow` or in section 4.

3.2 Web Camera Feeds

Code for obtaining stills from the web cameras are found in `getDOTstream.py`.

```
class stream:
    def GET(self):

class testCamera:
    def GET(self):

class getStream:
    def __init__(self, url):

    def getImage(self):
```

Listing 8: getDOTstream.py

This file follows the typical web.py framework. The first class, stream, pulls an image from the web camera feed url, and calls getStream.getImage() for TensorFlow classification.

The testCamera class is just for testing the SSD model from TensorFlow.

3.3 TensorFlow Classifier

Boilerplate code for loading a pretrained model, and for classifying an image by passing it through the model.

```
class CarDetector:
    def __init__(self):

    def getClassification(self, img):
```

Listing 9: TF_SSD.py

This class is called in `getDOTstream.py` to classify images.

Glossary

Borough Block Lot Method commonly used in New York City to identify a specific building. 2–4

Census Tract Number which refers to the tract which a block is grouped with during the 2010 US Census. 2, 4

Energy Usage Intensity Average energy usage per area. 2, 3

4 TensorFlow

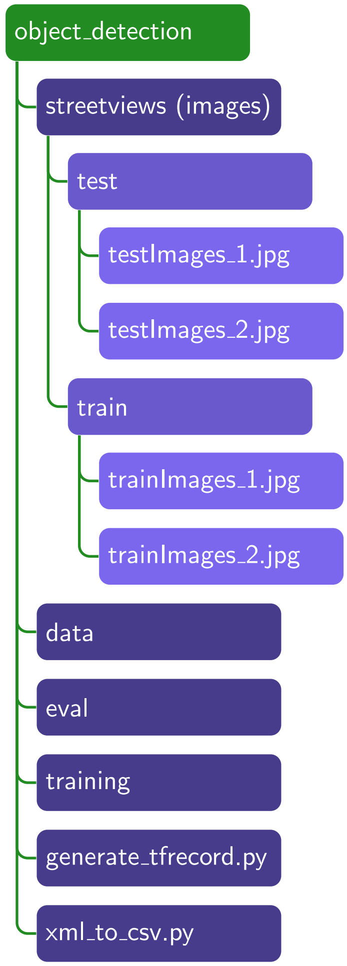
BEFORE DOING ANYTHING

When opening up a new terminal, make sure to do the following:

1. Go to the *models/research* directory
2. Run the following:

```
export PYTHONPATH=$PYTHONPATH:‘pwd‘:‘pwd‘/slim
```

If you don't do this, you will get some sort of "cannot import name input reader pb2" error. And it will be frustrating.



4.1 labelImg

The images can be annotated using the labelImg application. The labelImg application can be compiled by downloading the source:
`https://github.com/tzutalin/labelImg#macos`

```
brew install python3
pip install pipenv
pipenv --three
pipenv shell
pip install py2app
pip install PyQt5 lxml
make qt5py3
rm -rf build dist
python setup.py py2app -A
```

```
mv "dist/labelImg.app" /Applications
```

The application labelImg will generate xml files, which will be saved in the same directory (e.g. train or test) as the image files.

4.2 XML to CSV

The file *xml_to_csv.py* translates the xml files to csv, concatenates the csv files, and saves them in the data directory. The original files are taken from `https://github.com/datitran/raccoon_dataset`, and are modified to convert both train and test images.

This can be run as follows:

```
python xml_to_csv.py
```

4.3 TFRecords

TFRecords are required to run the TensorFlow. They can be generated from the csv files (which are hopefully now stored in the directory `data/`). The TFRecords can be generated using the file `generate_tfrecord.py`. If using more than one class, remember to change the `def class_text_to_int(row_label)` function to reflect multiple classes.

One line in the main function must be modified before running the python command:

```
path = os.path.join(os.getcwd(), 'PATH TO IMAGES') #e.g. streetviews/test
```

Run the following commands:

```
python generate_tfrecord.py --csv_input=data/train_labels.csv
--output_path=data/train.record

python generate_tfrecord.py --csv_input=data/test_labels.csv
--output_path=data/test.record
```

Making sure to change the path to image files before running each. These commands will generate two files: test.record and train.record, in the data folder.

4.4 Training

Make sure to have a file called object-detection.pbtxt, which contains the classes to be trained on:

```
item {
id: 1
name: 'window'
}
```

In the folder `models/research/object_detection/smaples/configs/`, you can find a multitude of model configuration files. The corresponding model files can be found in:

`https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md`

Download the model, unzip, and put in the base directory. Place the config file in the `training/` folder.

A few changes should be made to the config file:

- Change `num_classes` to the number of classes (e.g. 1)
- Change `batch_size` lower if using a CPU. 10 seems reasonable.
- Change `fine_tune_checkpoint` to the `model.ckpt` file in the model folder. E.g. `ssd_mobile_v1_coco/model.ckpt`
- Change training `input_path` to the training records (e.g. `data/train.record`)
- Change evaluation `input_path` to the testing records (e.g. `data/test.record`)
- Might also need to change the `label_map_path` for training and evaluation to the .pbtxt file in the data folder.
- Change `num_steps` to a reasonable number. Running on GPU might make this better.
- Change `eval_config` number of examples to something reasonable, such as 10.

Now we are finally ready to run the training. The new TensorFlow changed the location of `train.py`, but we can still run it (for now). Run the following:

```
python legacy/train.py --logtostderr
  --train_dir=training/
  --pipeline_config_path=training/ssd_mobilenet_v1_coco.config
```

If you see loss, this is a success.

4.5 Evaluation

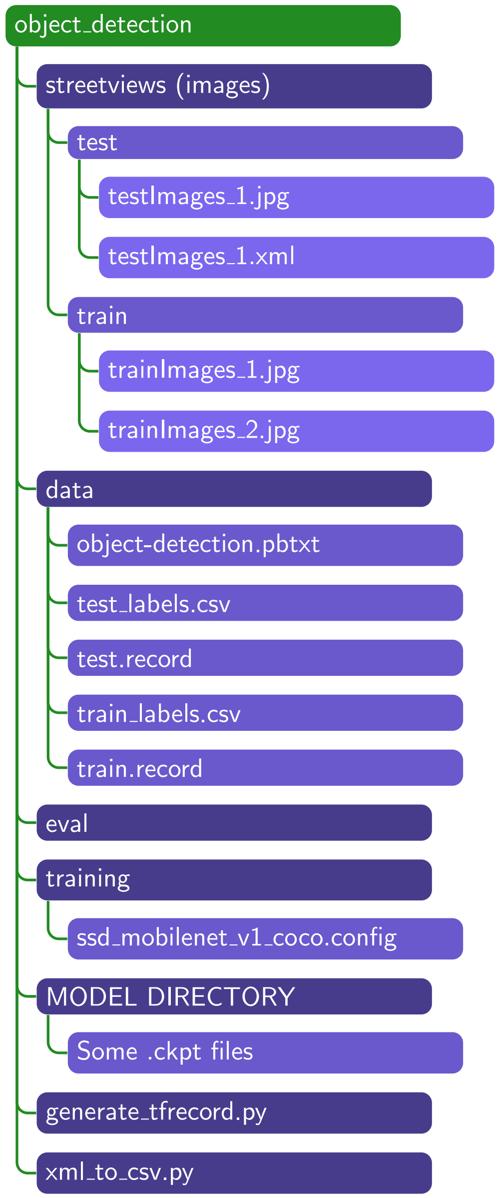
To evaluate the model, run the following:

```
python legacy/eval.py
  --logtostderr
  --pipeline_config_path=training/ssd_mobilenet_v1_coco.config
  --checkpoint_dir=training/
  --eval_dir=eval/
```

The results will be saved in the `eval/` directory (make on if you don't have one). Sometimes this command takes forever, so running ctrl-c to cancel it seems to still save the results.

To visualize the evaluation results, run:

```
tensorboard --logdir=eval/ #for viz eval results
tensorboard --logdir=training/ #for viz training results
```



4.6 Exporting Model

To export the model, use the `export_inference_graph.py` file, and move it to the folder containing the model.config file (in this case, training/). You may need to create a folder called `fine_tuned_model` first.

The new file, `frozen_inference_graph.pb` is the file that can be loaded into a system.

4.7 Traps to Avoid

1. Again, make sure you are in the `models/research/` folder before running the export PYTHONPATH command.

2. When saving the model as pb, it will not work when transferring across platforms if the **tensorflow versions** are not the same.