**SpaceNet2 - Marathon Match - Solution Description**

**Overview**

Congrats on winning this marathon match. As part of your final submission and in order to receive payment for this marathon match, please complete the following document.

1. **Introduction**

Tell us a bit about yourself, and why you have decided to participate in the contest.

* Handle: wleite
* Placement you achieved in the MM: 2nd

1. **Solution Development**

How did you solve the problem? What approaches did you try and what choices did you make, and why? Also, what alternative approaches did you consider?

* My solution is roughly the same from the first challenge.
* What I did in a different way was to test one change at time, as we had much more time, instead of relying on intuition or on a very small testing subset, as I did in the first contest.

1. **Final Approach**

Please provide a bulleted description of your final approach. What ideas/decisions/features have been found to be the most important for your solution performance:

* After adapting the solution from the first round to work with the different organization and formats of input files, I revisited the main points of my solution. Few things were removed, as they only add complexity without any visible gain. Others where adjusted, based on the results of tests with the new data sets. The following paragraphs describe the final solution.
* **Cities:** Each city is treated in a completely independent way, i.e. training and testing cycles were executed for each city, as buildings characteristics look somewhat different, and there are enough data for each city.
* **Image Formats:** My final solution uses the high-resolution pan image, the 3-band RGB pan sharpen and the low-resolution multiband. I tried to replace the multiband images by their pan sharpen version, but it produced worst results. Although visually the pan sharpen version looks much better, I am not sure it adds more information, compared to the original low-resolution version, as the high-resolution pan image is already used.
* **Image Preprocessing:** A simple blur is applied to the gray and each RGB channels, using a standard 3x3 neighborhood of the pixel with the weight. After blurring RGB values, each pixel is converted to HSL channels, which will be used later. Finally, a simple edge detection is executed for each channel of the multiband images and for the gray channel. The edge detection combines horizontal and vertical values of edges computed using a Sobel filter.
* **Pixel Classification Training:** This is the first step of my solution, implemented by the class BuildingDetectorTrainer. It uses as input 60% of the training images (for a given city). It builds two random forests of binary classification trees: one that classifies each pixel as **belonging or not to the border**, and the other as **inside of a building or not**. Pixels are treated individually, i.e. border and inside classification is based on individual pixels (in the first contest a 2x2 area was used here).

The resulting random forests were saved into two files: rfBorder.dat and rfBuilding.dat. My final model has 60 trees for each of these forests.

The features used were the same for the two classification forests. It uses the average, variance and skewness for small neighborhood squares around the evaluated pixel (more focused than used in the first contest). A total of 68 of values were used in the final submission, less than I used in the first contest. H (hue), S (saturation), G (gray) and E (edge) channels were used for the high-resolution versions of the images. For the lower-resolution multiband versions, the raw pixel value and edge value were used, for all 16 channels

To keep the number of sampled pixels in a reasonable limit, not every pixel is used for training. Pixels inside buildings are subsampled randomly in a rate of 1/8. For pixels outside the buildings, the subsample ratio is 1/16. All border pixels are used (1/1). For images with no buildings at all, the subsample ratio is 1/64.

* **Polygon Evaluation Training:** This is the second main step of the solution, implemented by the class PolygonMatcherTrainer. It used as input 40% of the training images (a different set of images from the previous step). For local tests it used a little less, reserving about 10% of the images for evaluating the solution.

Initially, it used the classifiers previously built to assign for each pixel a probability value of being a “border” and of being “inside a building”.

The following step is finding polygon candidates from the pixels, based on their classifications (border / inside). It then combines border and inside information in a single value, subtracting (possibly weighted) “border” values from “inside” values. The goal is to keep each building region separated from its neighbor buildings, so a simple flood fill can detect a single building. Borders act as separators to avoid merging different buildings close to each other.

A simple flood fill (expanding groups of 4-connected pixels) is executed, using as an input parameter the threshold level to decide where to stop flooding (higher values break the result into many smaller groups, while lower values join pixels into a smaller number of larger groups). Finally, a “border” is added to found groups because the building borders were not filled by the previous step. This process is repeated for many different threshold values.

A convex hull procedure is used to generate a polygon from a group of pixels. At this point, no verification (intersection, size, shape, position) is made, as these are only “candidates”.

Now the actually training part starts. Each candidate polygon is compared to ground truth buildings, calculating the best IOU value. A random forest of binary classification trees is built to predict, for a given polygon candidate, if its IOU is above 0.5. Final submission used 120 trees. In the first contest I used regression trees that tried to predict the IOU value of a given polygon candidate. This time, tests showed that using a binary classification produced similar final scores, but much more stable in regard of the chosen threshold used later to accept or reject polygons.

Features used are very similar to what I had in the first match using general polygon properties (including area, lengths of the smallest rectangle that contains the polygon, proportion between sides of this rectangle, proportion between the area of the bounding rectangle and the actual area, predicted “border” and “inside” values).

The resulting random forest is saved as a file named rfPolyMatch.dat.

* **Testing (Finding polygons):** For local tests, I used 10% of training data to evaluate my solution. This step is very straightforward, as it reuses methods described before. It is implemented by the BuildingDetectorTester class.

Using the same process for finding polygon candidates, many polygon candidates are found and then evaluated using the trained random forest.

At this point, each polygon candidate has a predicted value of having a IOU above 0.5. All candidates are then sorted, from higher to low probability values. Then each polygon is checked to verify if the intersection area with any previously accepted polygon is higher than 10% of either polygon area and, in such case, it is discarded.

Only polygons with a predicted probability of having a “high IOU” (above 0.5) higher than certain threshold are accepted. This threshold number came from local tests, trying to balance precision and recall to achieve the maximal F-Score. This is the only parameter that was tuned differently for each city, although the effect on final score was minor.

The actual values used in my final submission were:

* + - * Vegas: 0.40;
      * Paris: 0.38;
      * Shanghai: 0.31;
      * Khartoum: 0.29.

1. **Open Source Resources, Frameworks and Libraries**

Please specify the name of the open source resource along with a URL to where it’s housed and it’s license type:

* I did not use any external resource or library.

1. **Potential Algorithm Improvements**

Please specify any potential improvements that can be made to the algorithm:

* I believe that the search for polygon candidates is the part that could provide the most visible gain. I tried many different things during this second contest, but didn’t manage to find a better approach than the one used in the first round. It clearly misses a lot of buildings (specially smaller ones), and that prevent them to even be considered during the polygon evaluation phase.
* Another point that could be improved is the usage of multichannel pan sharpen images. Visually they as much better than the original multichannel images, but when I tried to use them instead, I got worse results. I had a feeling that the real information contained in the pan sharpen images is about the same contained in the original images, mixed with the high-resolution pan images, which were already used.

1. **Algorithm Limitations**

Please specify any potential limitations with the algorithm:

* It assumes that polygons are convex, so for regions with a lot of irregular building, this can be a serious limitation. I successfully implemented “concave-hull” algorithm this time. It produced much better matches for many cases, increasing the IOU value for “easy” cases, but decreased the final score, so I decided to go back to the original convex hull approach.

1. **Deployment Guide**

Please provide the exact steps required to build and deploy the code:

1. In the case of this contest, a Dockerized version of the solution was required, which should run out of box.
2. My solution only depends of JRE 8. All the rest (model files, compiled classes etc.) can be just copied.
3. **Final Verification**

Please provide instructions that explain how to train the algorithm and have it execute against sample data:

Training and testing scripts were provided with the solution. They just call “java” with the expected parameters:

**java** –Xmx120G -cp bin:lib/imageio-ext-geocore-1.1.16.jar:lib/imageio-ext-streams-1.1.16.jar:lib/imageio-ext-tiff-1.1.16.jar:lib/imageio-ext-utilities-1.1.16.jar:lib/jai\_codec-1.1.3.jar:lib/jai\_core-1.1.3.jar:lib/jai\_imageio-1.1.jar **SpacenetMain** **train <dataFolder1> [<dataFolder2>...]**

**java** –Xmx40G -cp bin:lib/imageio-ext-geocore-1.1.16.jar:lib/imageio-ext-streams-1.1.16.jar:lib/imageio-ext-tiff-1.1.16.jar:lib/imageio-ext-utilities-1.1.16.jar:lib/jai\_codec-1.1.3.jar:lib/jai\_core-1.1.3.jar:lib/jai\_imageio-1.1.jar **SpacenetMain** **test <dataFolder1> [<dataFolder2>...] <output csv file>**