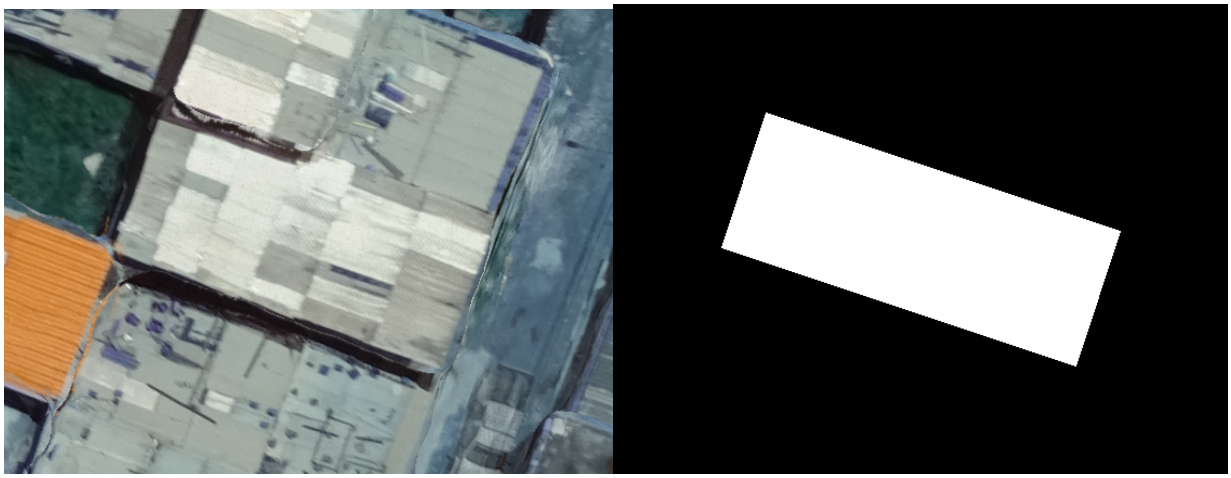
1. Who are you (mini-bio) and what do you do professionally?

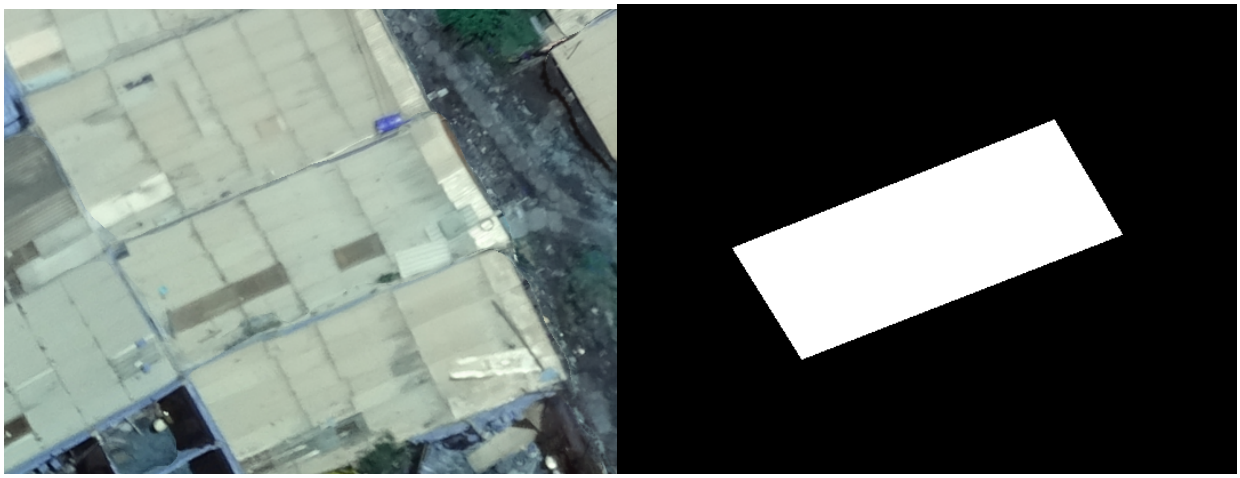
I'm Ph.D in microelectronics field. Currently I'm working in Institute of Designing Problems in Microelectronics (part of Russian Academy of Sciences) as Leading Researcher. I often take part in machine learning competitions. I have extensive experience with GBM, Neural Nets and Deep Learning as well as with development of CAD programs in different programming languages (C/C++, Python, TCL/TK, PHP etc).

2. High level summary of your approach: what did you do and why?

The solution is divided into the following stages:

Stage 1 [image extraction]: For every house on each map its RGB image is extracted with 100 additional pixels from each side as well as mask based on POLYGON description in GEOJSON format. See examples below:





Generated files:

../modified_data/train_img/*.png, ../modified_data/test_img/*.png, ../modified_data/train.csv, ../modified_data/test.csv,

File with code: r01_extract_image_data.py

Stage 2 [metadata extraction and found neighbors data]: on the next stage we extract the following data:

1) Meta information for each house in train and test dataset: **alias, region, epsg, center_x, center_y, area, tiff_id**. Where **center_x** and **center_y** represent location of house center on the map, **area** – is the area of house roof (based on provided POLYGON).

2) Several statistics on the material of roofs of neighboring houses for a given house:

- detailed statistics on the nearest 10 houses (roof material, coordinates, distance, area of the roof)

- distribution of roof material for the nearest 10 and 100 neighbors

- the number of neighbors and the distribution of roof material from neighbors within a radius of 1000 conventional units

- the number of neighbors and the distribution of roof material from neighbors within a radius of 10,000 conventional units

Features from these files used later in the 2nd level models in addition to neural net predictions. It's needed because neural nets only see the roof image and related polygon. But neural net doesn't know the location of house on the overall map. Meta features as well as some neighbor statistics helps to make predictions more precise.

This is possible for the following reasons:

1) In certain regions, various types of materials are more popular

2) Obviously, material from neighboring houses has a higher chance of being similar.

3) Annotators who made markings on one map are mistaken in a similar way or choose some material more often (for example, roof material "other").

All obtained data are stored in CSV files for later fast access.

Generated files: train_neigbours.csv, test_neigbours.csv, neighbours_clss_distribution_10_train.csv, neighbours_clss_distribution_10_test.csv, neighbours_clss_distribution_100_train.csv, neighbours_clss_distribution_100_test.csv, neighbours_clss_distribution_radius_1000_train.csv, neighbours_clss_distribution_radius_1000_test.csv, neighbours_clss_distribution_radius_10000_train.csv, neighbours_clss_distribution_radius_10000_test.csv. All files located in «*../features/*» folder.

File with code: r02_find_neighbours.py

Stage 3 [train different convolutional neural nets]: This is the longest and most time-consuming stage of calculations. At this stage, 7 different models of neural networks are trained, which differ in following: the type of neural network, the number of folds, different set of augmentations, some training procedure differences, different input shape, etc.

General training principle for all neural networks:

1) A 4-channel image is provided to the input of the neural network, where the first 3 channels are an RGB image, and the 4 channel is a mask obtained from POLYGON. In order to use ImageNet Pre-train weights, the convolution weights of the first layer of the neural network are recalculated in a special way:

   w_new[:, :, j, :] = ref_ch * w[:, :, j%ref_ch, :] / upd_ch

2) The output of the neural network is a softmax layer with 5 neurons, and loss function: categorical_crossentropy.

3) Training is on all images, validation only on verified images.

4) Typical augmentations: HorizontalFlip, RandomRotate90 - the probabilities are chosen so that each of the 8 possible turns and reflections occur uniform.

   Other augmentations:

   RGBShift(p=0.5, r_shift_limit=(-20, 20), g_shift_limit=(-20, 20), b_shift_limit=(-20, 20)),

```
ShiftScaleRotate(shift_limit=0.1, scale_limit=0.1, rotate_limit=45, p=0.5, border_mode=cv2.BORDER_CONSTANT),
OneOf([
    MedianBlur(p=1.0, blur_limit=7),
   Blur(p=1.0, blur_limit=7),
   GaussianBlur(p=1.0, blur_limit=7),
], p=0.2),
OneOf([
     IAAAdditiveGaussianNoise(p=1.0),
     GaussNoise(p=1.0),
], p=0.2),
OneOf([
     ElasticTransform(p=1.0, alpha=1.0, sigma=30, alpha_affine=20),
     GridDistortion(p=1.0, num_steps=5, distort_limit=0.3),
     OpticalDistortion(p=1.0, distort_limit=0.5, shift_limit=0.5)
], p=0.2)
```

5) Some special augmentation was used: borders from 0 to 100 pixels from each side were randomly cut off (I remind that at the preprocessing stage we cut out images with a margin of 100 pixels on each side). On validation, exactly 50 pixels were cut off from each side.

6) Most neural networks have an additional Dense (FullyConnected) layer after GlobalAveragePooling with following Dropout with a probability value of 0.5 or even more to reduce overfit.

List of neural networks and training specialty for each of them:

1) Densenet121 - input size 224x224 pixels

2) Inception Resnet v2 - input size 299 pixels, two network were trained with different options

3) EfficientNetB4 - input size 380 pixels

4) DenseNet169 - input size 224 pixels, added argumentation with a random small shift and rotation of the mask (emulates human markup errors), 6 folds

5) ResNet34 - a small network, input size 224 pixels, a minimum set of augmentations

6) SeResnext50 - input size 224 pixels

7) ResNet50 - input size 224 pixels, 10 Kfold

After training, for each fold, the best models are found by the value of loss on validation. On all images of the training set and test set, an inference is launched and the probabilities for each type of roof are obtained. Each image is passed through the model several times, using all 8 configurations of 90-degree rotations and reflections, as well as cutting off various border values from the edges of the image.

The resulting files are used further in second-level models. Predictions on test data have the same format as submit and can be tested on LeaderBoard to check if everything is fine.

Generated files:

d121_kfold_valid_TTA_32_5.csv, d121_kfold_test_TTA_32_5.csv,

irv2_kfold_valid_TTA_32_5.csv, irv2_kfold_test_TTA_32_5.csv,

irv2_kfold_v2_valid_TTA_32_5.csv, irv2_kfold_v2_test_TTA_32_5.csv,

effnetb4_kfold_valid_TTA_32_5.csv, effnetb4_kfold_test_TTA_32_5.csv,

densenet169_kfold_valid_TTA_32_6.csv, densenet169_kfold_test_TTA_32_6.csv,

resnet34_kfold_valid_TTA_32_5.csv, resnet34_kfold_test_TTA_32_5.csv

seresnext50_kfold_valid_TTA_32_5.csv, seresnext50_kfold_test_TTA_32_5.csv,

resnet50_kfold_valid_TTA_32_10.csv, resnet50_kfold_test_TTA_32_10.csv

Files with code:

cnn_v1_densenet121/r16_classification_d121_train_kfold_224.py
cnn_v1_densenet121/r26_classification_d121_valid_kfold_224.py
cnn_v2_irv2/r16_classification_irv2_train_kfold_299.py
cnn_v2_irv2/r17_classification_irv2_train_kfold_299_v2.py
cnn_v2_irv2/r26_classification_irv2_valid_kfold_299.py
cnn_v2_irv2/r26_classification_irv2_valid_kfold_299_v2.py
cnn_v3_efficientnet_b4/r17_classification_efficientnet_train_kfold_380.py
cnn_v3_efficientnet_b4/r26_classification_efficientnet_valid.py
cnn_v4_densenet169/r17_classification_densenet169_train_kfold_224.py
cnn_v4_densenet169/r26_classification_densenet169_valid.py
cnn_v5_resnet34/r17_classification_train_kfold_224.py
cnn_v5_resnet34/r26_classification_valid.py
cnn_v6_seresnext50/r17_classification_train_kfold_224.py
cnn_v6_seresnext50/r26_classification_valid.py
cnn_v7_resnet50/r17_classification_train_kfold_224.py
cnn_v7_resnet50/r26_classification_valid.py

Stage 4 [2nd level models]:

Predictions from all neural networks, as well as metadata and data for neighbors obtained in the previous steps are further used as input for second-level models. Since the dimension of the task is small, I used three different GBM modules - LightGBM, XGBoost and CatBoost. Each of them starts several times with random parameters. And the predictions are then averaged. The result of each model is a submit file, in the same format as the Leader Board.

Generated files: catboost_ensemble.csv, xgboost_ensemble.csv, lightgbm_ensemble.csv

Files with code: gbm_classifiers/r15_run_catboost.py, gbm_classifiers/ r16_run_xgboost.py, gbm_classifiers/ r17_run_lightgbm.py

Stage 5 [Final ensemble of 2nd level models]:

Predictions from each GBM classifier then averaged with the same weight. And the final prediction is generated.

Generated files: subm/submission.csv

Files with code: r20_ensemble_avg.py

3. Copy and paste the 3 most impactful parts of your code and explain what each does and how it helped your model.

Code sample 1

r01_extract_image_data.py:

```
def process_single_row(train_df, index, type):
    global total_multi_train, total_multi_test

    delta = 100
    sample = train_df.iloc[index]

    # print('Go for: {}'.format(sample.id))
    if type == 'train':
        out_png = TRAIN_PATH + '{}.png'.format(sample.id)
    else:
        out_png = TEST_PATH + '{}.png'.format(sample.id)

    polygon = sample.geometry
    if 'MULTI' in str(polygon):
        if type == 'train':
            total_multi_train += 1
        else:
            total_multi_test += 1
        print('Some multi here', sample.id)
    else:
        if os.path.isfile(out_png):
            return

    tuples = polygon_to_tuples(polygon)
    proj = Proj(init='epsg:{}'.format(sample.epsg))
    tiff_path = sample.tiff_path
    coordinates = transform_coordinates([tuples], proj)[0]

    tiff = rasterio.open(tiff_path)
    # print('Tiff size:', tiff.width, tiff.height)
    pixels = [tiff.index(*coord) for coord in coordinates]
    xmin, xmax, ymin, ymax = 1000000000, -10000000000, 1000000000,
-10000000000
    for w, h in pixels:
        if w > xmax:
            xmax = w
        if w < xmin:
            xmin = w
        if h > ymax:
            ymax = h
        if h < ymin:
            ymin = h
    if xmin < delta:
        print('Too small xmin')
    if ymin < delta:
        print('Too small ymin')
    if xmax > tiff.width - delta:
        print('Too big xmax')
    if ymax > tiff.height - delta:
        print('Too big ymax')

    window = ((int(xmin), int(xmax)), (int(ymin), int(ymax)))
    xmin_ex = xmin - delta
    xmax_ex = xmax + delta
```

```
    ymin_ex = ymin - delta
    ymax_ex = ymax + delta
    window_expanded = ((int(xmin_ex), int(xmax_ex)), (int(ymin_ex),
int(ymax_ex)))

    # print(pixels)
    # print(window)
    b = tiff.read(1, window=window_expanded)
    g = tiff.read(2, window=window_expanded)
    r = tiff.read(3, window=window_expanded)
    alpha = tiff.read(4, window=window_expanded)
    # print(r.shape, r.min(), r.max(), r.mean())
    # print(g.shape, g.min(), g.max(), g.mean())
    # print(b.shape, b.min(), b.max(), b.mean())
    # print(alpha.shape, alpha.min(), alpha.max(), alpha.mean())
    if alpha.max() > 255:
        print('Big alpha!')
        exit()

    mask = np.zeros(b.shape, dtype=np.int32)
    poly = np.array(pixels)
    poly[:, 0] -= xmin_ex
    poly[:, 1] -= ymin_ex
    if 1:
        r1, c1 = poly[:, 0].copy(), poly[:, 1].copy()
        poly[:, 0], poly[:, 1] = c1, r1
    # print(poly)

    mask = cv2.fillConvexPoly(mask, poly, 255)
    img = np.stack((b, g, r), axis=2)
    # show_image(img)
    # show_image(mask)
    if type == 'train':
        cv2.imwrite(out_png, img)
        cv2.imwrite(TRAIN_PATH + '{}_mask.png'.format(sample.id),
mask.astype(np.uint8))
        cv2.imwrite(TRAIN_PATH_ALPHA + '{}.png'.format(sample.id),
alpha.astype(np.uint8))
    else:
        cv2.imwrite(out_png, img)
        cv2.imwrite(TEST_PATH + '{}_mask.png'.format(sample.id),
mask.astype(np.uint8))
        cv2.imwrite(TEST_PATH_ALPHA + '{}.png'.format(sample.id),
alpha.astype(np.uint8))
```

In this code, the image and mask are extracted for one house on the map. Initially, I had difficulty getting the right coordinates.

Code sample 2

r16_classification_d121_train_kfold_224.py:

```
def model_DenseNet121_multich(size, upd_ch):
    from keras.models import Model, load_model
    from keras.layers import Dense, Input
    from keras.applications import DenseNet121

    ref_ch = 3
    required_layer_name = 'conv1/conv'

    weights_cache_path = CACHE_PATH + 'model_DenseNet121_{}
ch_imagenet_{}.h5'.format(upd_ch, size)
```

```
    if not os.path.isfile(weights_cache_path):
        model_ref = DenseNet121(include_top=False,
                                weights='imagenet',
                                input_shape=(size, size, ref_ch),
                                pooling='avg',)
        model_upd = DenseNet121(include_top=False,
                                weights=None,
                                input_shape=(size, size, upd_ch),
                                pooling='avg', )

        for i, layer in enumerate(model_ref.layers):
            print('Update weights layer [{}]: {}'.format(i, layer.name))
            if layer.name == required_layer_name:
                print('Recalc weights!')
                config = layer.get_config()
                use_bias = config['use_bias']
                if use_bias:
                    w, b = layer.get_weights()
                else:
                    w = layer.get_weights()[0]
                print('Use bias?: {}'.format(use_bias))
                print('Shape ref: {}'.format(w.shape))
                shape_upd = (w.shape[0], w.shape[1], upd_ch, w.shape[3])
                print('Shape upd: {}'.format(shape_upd))

                w_new = np.zeros(shape_upd, dtype=np.float32)
                for j in range(upd_ch):
                    w_new[:, :, j, :] = ref_ch * w[:, :, j%ref_ch, :] /
upd_ch
                if use_bias:
                    model_upd.layers[i].set_weights((w_new, b))
                else:
                    model_upd.layers[i].set_weights((w_new,))
                continue
            else:
                model_upd.layers[i].set_weights(layer.get_weights())
        model_upd.save(weights_cache_path)
    else:
        model_upd = load_model(weights_cache_path)

    x = model_upd.layers[-1].output
    x = Dense(NUM_CLASSES, activation='softmax', name='prediction')(x)
    model = Model(inputs=model_upd.inputs, outputs=x)
    # print(model.summary())
    return model
```

This function is used to convert a 3-channel input from a neural network to a 4-channel one with replacing weights at the first convolutional layer. This is important for training the network not from scratch, but with ImageNet weights, while using a four-channel input. The function is used with minor changes for all neural networks.

Code sample 3

r26_classification_d121_valid_kfold_224.py:

```
def get_TTA_image_classification(img_orig):
    img_batch = []
    # delta_list = [100, 75, 50, 25]
```

```
    delta_list = [99, 95, 90, 85, 80, 75, 70, 65, 60, 55, 50, 45, 40, 35, 30,
25, 20, 15, 10, 5, 1]
    for d in delta_list:
        img = img_orig[d:-d, d:-d, :].copy()
        if img.shape[0] < 10 or img.shape[1] < 10:
            print('Image shape is small: {} {}'.format(d, img.shape))
            continue
        if (img.shape[0] != SHAPE_SIZE[0]) or (img.shape[1] !=
SHAPE_SIZE[1]):
            img = cv2.resize(img, (SHAPE_SIZE[1], SHAPE_SIZE[0]),
interpolation=cv2.INTER_LINEAR)
        for i in range(8):
            im = get_mirror_image_by_index(img, i)
            img_batch.append(im)

    img_batch = np.array(img_batch, dtype=np.float32)
    img_batch = preproc_input_classifcation(img_batch)
    return img_batch
```

This is the code for Test Time Augmentation. In this function, all versions of its rotations and reflections, as well as different options for cutting borders, are prepared for single image. All this allows us to get more precise predictions of the neural network after averaging. This function is also used for all neural networks at the inference stage.

4. What are some other things you tried that didn't necessarily make it into the final workflow (quick overview)?

Almost everything I did was eventually included in the final version, except the following:

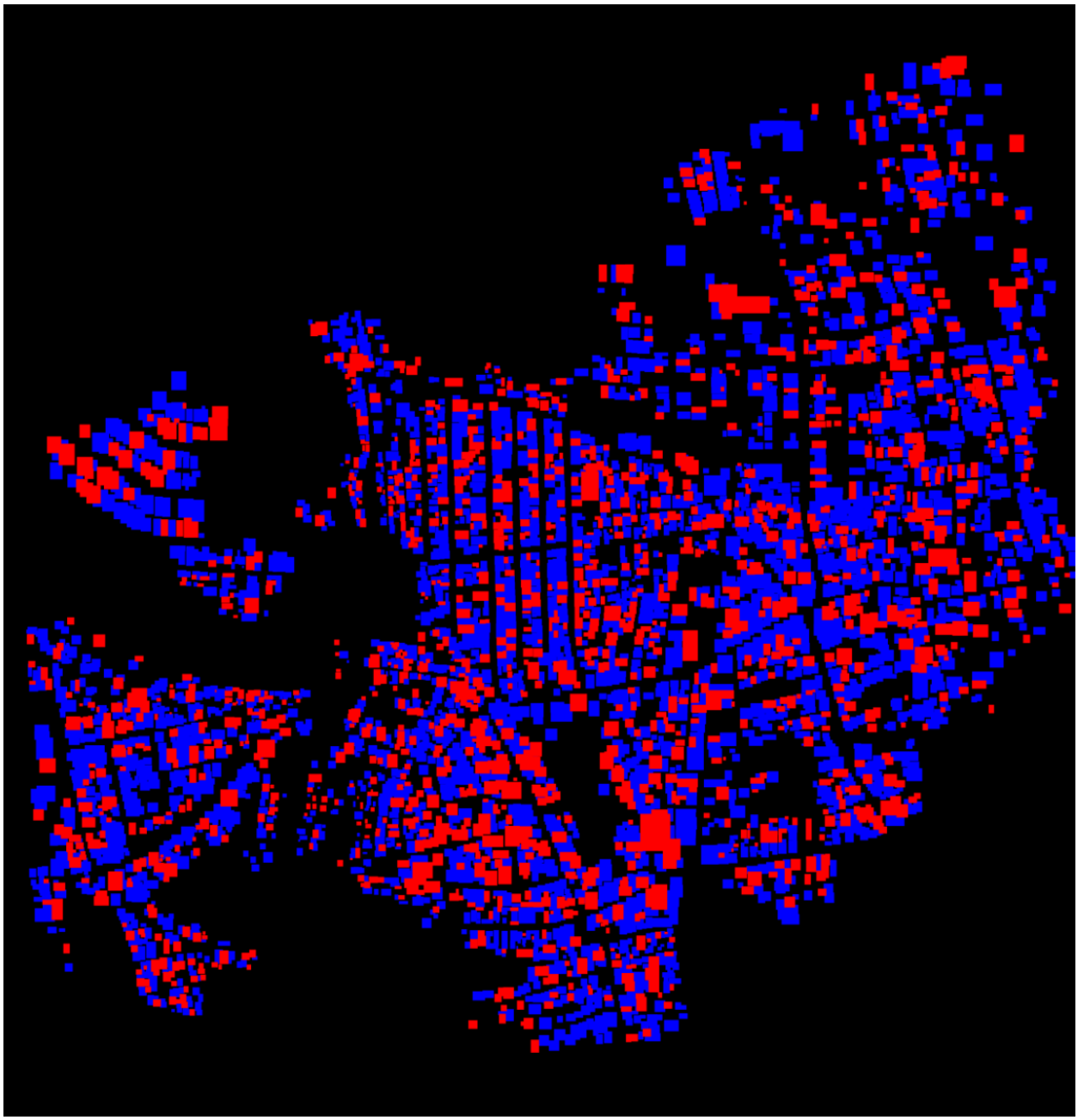
1) Pseudolabels did not work. I tried to predict test images with best model and then use them for training. Validation was improved a lot, but LB became worse.

2) A second-level model based on neural networks. I used default classification NN in Keras for the same data as GBM classifiers. It gives result slightly worse than any other GBM classifier. And it didn't increase score for overall ensemble on validation as well as on LB.

5. Did you use any tools for data preparation or exploratory data analysis that aren't listed in your code submission?

All analysis and data preparation were done in Python. I didn't use any outside tools and didn't make any additional annotation in training dataset.

I checked visually that train and test are mixed uniform across all the pictures in order to understand how to do validation correctly. See image below, which shows projection of all houses on map (red – train data, blue – test data).



6. **How did you evaluate performance of the model other than the provided metric, if at all?**

Local validation for me quite well coincided with the values on LB, locally it was even a little worse. The only trick I used was validation only on Verified labels.

7. **Anything we should watch out for or be aware of in using your model (e.g. code quirks, memory requirements, numerical stability issues, etc.)?**

The code for training neural nets requires powerful GPUs like NVIDIA GTX 1080 Ti. And it's computationally expensive. All other parts of code should be run fast and with lower requirements. I'd recommend using CPU with 6-8 cores. Also SSD hard drive is critical, since during neural net training code read many data from disc and it can be bottleneck in case of default HDD. Minimum RAM memory requirement is 32 GB. It's possible to use several GPUs for training different neural networks.

Code is split in independent parts, has logical structure and print useful debug information. So it should be easy to run on your side.

8. **Do you have any useful charts, graphs, or visualizations from the process?**

I didn't prepare anything graphical this time.

9. **If you were to continue working on this problem for the next year, what methods or techniques might you try in order to build on your work so far? Are there other fields or features you felt would have been very helpful to have?**

It seems to me that in this task it is already difficult to move somewhere further in terms of a significant improvement in the result. To seriously improve the result, you need to improve the markup of the data, which turned out to be quite noisy. It seems to me that in the case of obtaining high-quality markup, a neural network (even a single one) will be able to solve this problem with almost 100% accuracy.