A simple and interpretable baseline for predictive maintenance

Ivan Klenovskiy Follow
Oct 20, 2019 · 9 min read ★

Futuristic picture since we are doing ML

A while ago I have encountered a predictive maintenance problem on my data science job. Highly unbalanced classes (since the number of breakages is typically less than 1%), huge

number of non-interpretable sensors — ok, but what is next? Since I lacked any experience with such tasks, I started googling and reading everything about ”data science for PM” I could possibly find.

To my disappointment, most of the articles I have discovered did not provide me with valuable insights — they were either some complex Arxiv papers, which tested new theoretical methods on unavailable data, or some high-level blogging posts, short on reusable ideas.

After a couple of months of hands-on experience solving a predictive maintenance (set to PM further) problem with machine learning, weary with toil, I decided to share the baseline approach I have developed so far. It may be fruitful to people who, as me a couple of months ago, do not know what to do and want to have some sort of a beacon nearby.

In the post, I will explain the business side of the problem and list typical requirements to the algorithm from the client, which may significantly restrict the set of data science tools available. Next, I will explain my approach in details and underline its pros and cons. Finally, I will share some practical hacks and bits and pieces of code which can be helpful in implementing it.

Business problem

You can skip this part if you are interested only in technical things.

Every ML algorithm installed in a business process should somehow increase profits. In PM we usually have a large-scale machine, consuming expensive resources to produce some sort of final output through a complex multi-stage process. With time those machines break down unexpectedly, causing an increase in idle time and a loss of valuable resources, since a malfunctioning machine typically wastes them.

This is where an ML algorithm can enter the scene — it can predict malfunctions (I use stops, breakages, stoppages interchangeably throughout the text) in advance, using data from the sensors installed in the machine. However, even if we are able to predict those stops in advance, without knowing the cause, financial value of our prediction would be modest.

First, if the only information we report to engineers is that there is a high chance of breakage in the next 20 minutes, they cannot do much with it. Probably, they can stop the machine and do some sort of deep maintenance but this is costly. Moreover, with large-scale machines, they will not necessarily discover the cause for malfunction in such a short period of time, therefore, the value of our prediction is nullified. Second, our algorithm will have a certain amount of false positives. With the presence of false positives, stopping the machine to check for obscure malfunction will be rarely approved from the client side.

Hence, our algorithm, besides predicting the probability of stoppage, should somehow hint on the possible causes. It

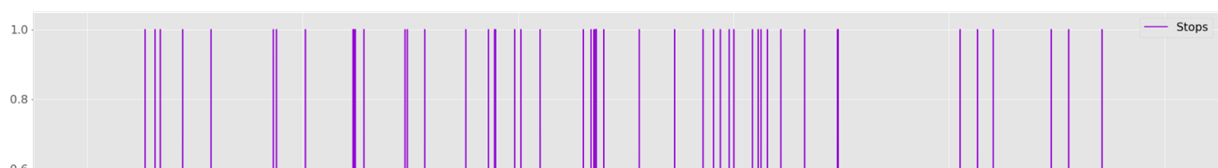
should provide certain guidance for the technicians searching for a malfunction during preventive maintenance. In this case, our ML algorithm truly solves client's pain.

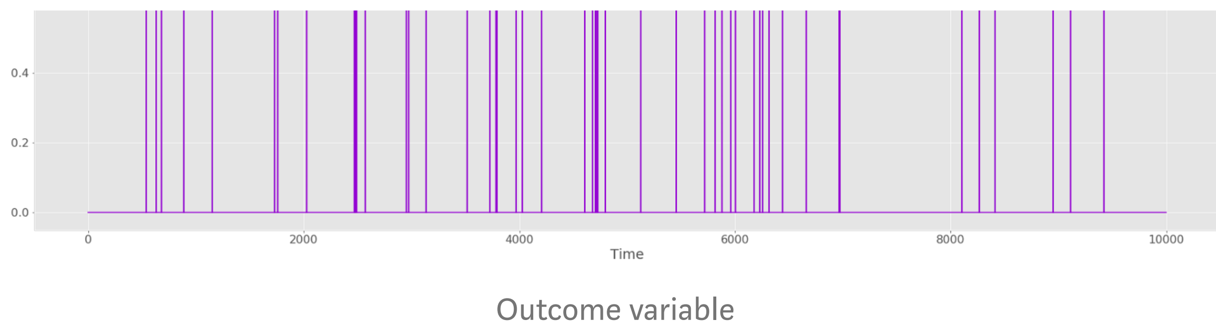
Happy manufacturing worker

Data Preparation

Now let's take a closer look at a typical predictive maintenance data. Here is how the outcome variable will look like once you have finished data preparation. $\mathbf{Y = 1}$ if the stop is soon happening and $\mathbf{Y = 0}$ if there is no stop in the next $\mathbf{K}$ minutes.

You can see that classes are unbalanced and machine learning models should give a higher weight for rare class in order to distinguish anomalies causing stops with stronger attention.





Outcome variable

But to obtain this time series you will have to do some grunt work. Usually, they will provide you with two datasets — one with information about stops (their nature, causes, and timestamp) and a second with all the data from the sensors.

We need to merge those two files to create our data set. I advise to drop observations (code sample of data processing is available below) when machine was broken and observations near the stops, as this information might be noisy. For instance, one could drop all observations for which time to next stop is less than 5 minutes.

Next, let's label all observations with time to next stop longer than 20 minutes as zero, while label all observations with time to next stop less than 20 minutes as one (you can vary this cut-off based on your data). This operation slightly improves your class balance.

Usually there are different types of stops occurring in different parts of the machine. I suggest to filter the data and choose the most common one and then generalize the solution to other stops. Different types of stops have distinct causes, therefore, information about each stop will be contained in distinct sets of

sensors. Hence, one generic model is unlikely to perform well.

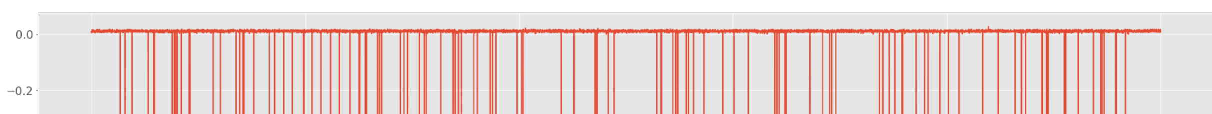
Some code which might help you with the operations described above:

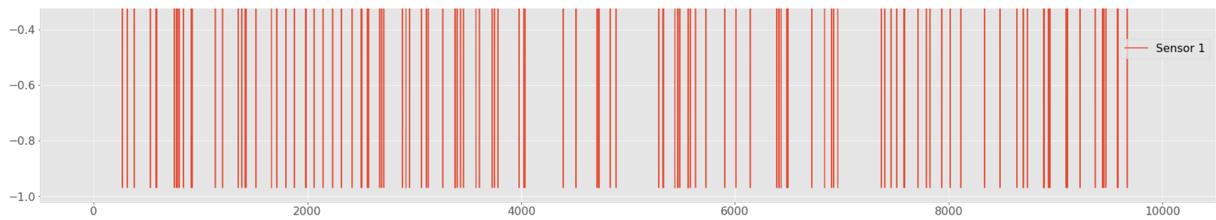
```
1  def y_prep(tmp, interval = [10,5], x = 'Time', type =
2      # forward fill current stops
3      tmp[['Cur_Stop', 'Cur_Stop_end']]  =tmp[['Stop_t',
4      # back fill future stops
5      tmp[['Nxt_Stop', 'Nxt_Stop_start']] = tmp[['Stop_t
6      # time since last and next stop
7      tmp['delta_cur'] = ((tmp['Cur_Stop_end'] - tmp[x])
8      tmp['delta_nxt'] = ((tmp['Nxt_Stop_start'] -tmp[x]
9      tmp['Y'] =  (tmp[x]<=interval[0]).astype(int)*(tmp
10     tmp = tmp[(tmp['Y'] == 0)&(tmp[x] >interval[1])|
11                 ((tmp['Y'] == 1)&((tmp['Nxt_Stop'] == ty
12 return tmp
```

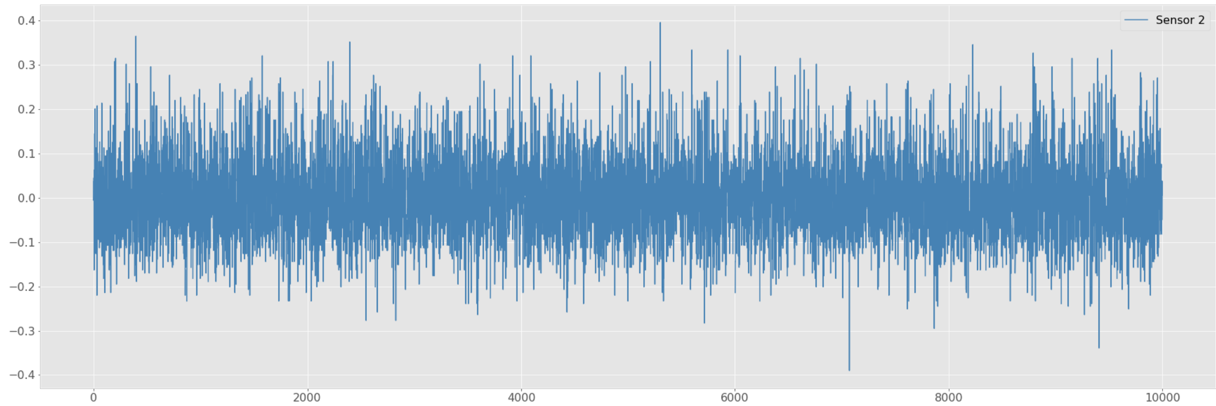
creating_Y.py hosted with ❤ by GitHub view raw

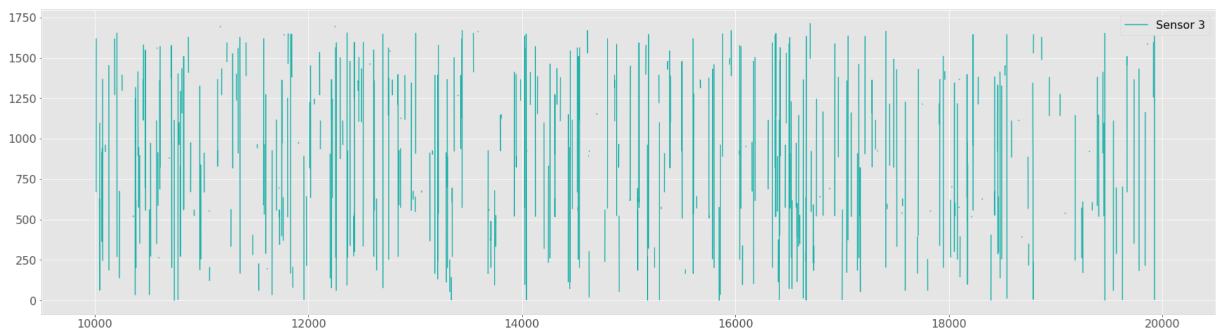
https://gist.github.com/ikleni/fa9421f23852459e166f14a5f4edb584

Second, we should deal with the explanatory variables (**Xs**). After we merge them with an outcome variable, we should try to reduce dimensionality in order to draw meaningful conclusions. Without any prior knowledge of the industry specifics, we can't select which ones correspond to the important predictors based on our intuition. Just take a look at some plots of Xs:









Examples of sensors

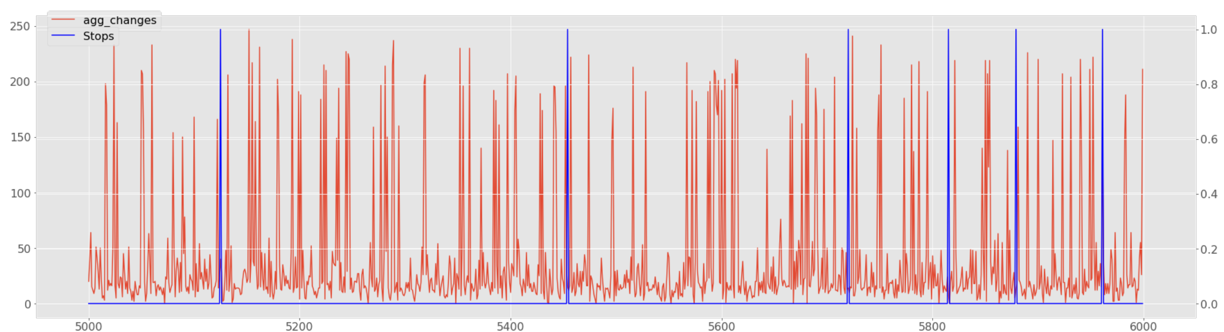
However, we have a lot of techniques at our hand to reduce dimensionality:

- Drop sensors with extra-low variance
- Group and aggregate (via PCA or other similar techniques) sensors according to their location/role in the machine (this information can be obtained from their tags)
- Aggregate changes of variables (how many sensors have just changed their regime)

Selecting variables based on raw correlation with outcome variable would be a bad idea

In my task, aggregations of changes of variables was helpful since it both contained information that machine is unstable and reduced the number of variables.

Here is one such aggregated variable:



Aggregated variable and Y

And some code which might help with dimensionality reduction:

```
1  def del_const_feat(X, thres = 5, dtype = int):
2      constant_cols = []
3      if dtype == int:
4          for i in X.dtypes[(X.dtypes == int)].index:
5              if  X.shape[0]- X[i].value_counts().sort_v
6                  constant_cols.append(i)
7      if dtype == float:
8          for i in X.dtypes[(X.dtypes == float)].index:
9              if  X[i].std() == 0:
10                 constant_cols.append(i)
11             if  X.shape[0]- X[i].value_counts().sort_v
12                 constant_cols.append(i)
```

```
13
14        X = X.drop(columns=constant_cols)
15        return X
```

dropping_cnst.py hosted with ❤ by GitHub view raw

https://gist.github.com/ikleni/81fd229802d62af099a92b037af9e500

```
1   def agg_dev(X, feats, cat = True):
2       X = X.set_index('Id')
3
4       # for lists of discrete features
5       if cat == True:
6
7           # count how many features changed in value
8           alpha = np.zeros(len(tmp_mini3)) # placeholder
9           for i in feats:
10              # get lagged values
11              tmp_mini = X[i].shift(1 ).reset_index()
12              tmp_mini2 = X[i].reset_index()
13              tmp_mini3 = pd.merge(tmp_mini, tmp_mini2,
14                                   on = 'Id', how = 'inn
15
16              # add 1 for each feature which differs
17              # from its lagged version
18              alpha +=(tmp_mini3[tmp_mini3.columns[1]] !
19                           tmp_mini3[tmp_mini3.columns[2]]).
20          tmp_mini['change_sum' + str(len(feats))] = alp
21          X = X.reset_index()
22          X= pd.merge(X, tmp_mini2[['Id', 'change_sum' +
23                         how = 'left', on = 'Id')
24          name = 'change_sum' + str(len(feats))
25  return X, name
```

agg_feat.py hosted with ❤ by GitHub view raw

agg_feat.py hosted with ❤ by GitHub view raw

https://gist.github.com/ikleni/f5cab0760f6121fed42ebc44e78bafe1

We can restrict our sensor space further.

Let’s iteratively pick each sensor and try to predict stops based on its quantiles. If the current time series quantile of the sensor is somewhat indicative of the stop, then the sensor contains information about abnormal behavior of the machine. I suggest selecting about 5–15 (this range was applicable to my particular problem) sensors whose quantile model has the strongest predictive power.

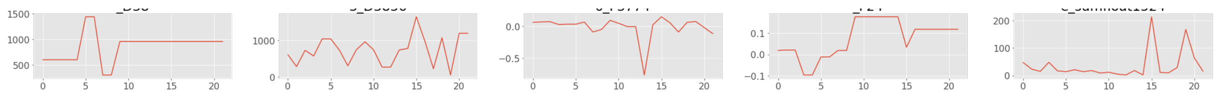
```
1   def class_score(dt, feature, q):
2       # checks how well does a particular feature
3       # split the data based on quantiles
4
5       q0, q1 = np.quantile(dt[feature], q[0] ), np.quant
6       dt['pred'] = ((dt[feature]< q0) |(dt[feature]> q1)
7       if dt[(dt['pred']==1)].shape[0] > 2:
8           score = dt[(dt['Response']==1)&(dt['pred']==1)
9           dt[(dt['pred']==1)].shape[0]
10      else:
11          score = 0
12      return (score,dt[(dt['pred']==1)].shape[0])
13
14  def quantile_search(dt, n_best = 20):
15      # selects best features in terms of
16      # quantile score
17      features =  list(dt.dtypes[(dt.dtypes == float)].i
18      features = [i for i in features if i not in ['Resp
19      results = {}
20      quant_list_bot = [0.1, 0.05, 0.01,0.001, 0.0001]
```

```
20        quant_list_bot = [0.1, 0.05, 0.01,0.001, 0.0001]
21        quant_list_top = [1-i for i in quant_list_bot]
22
23        quant_paired = list(itertools.product(quant_list_b
24        for i in features:
25            tmp_s = list(map(lambda x : class_score(dt, i,
26            tmp_ss = [i[0] for i in tmp_s]
27            tmp_sn = [i[1] for i in tmp_s]
28            opt_q = quant_paired[tmp_ss.index(max(tmp_ss))
29            opt_s = max(tmp_ss)
30            opt_n = tmp_sn[tmp_ss.index(max(tmp_ss))]
31            results[i] = [opt_q, (opt_s,opt_n)]
```

quantile_search.py hosted with ❤ by GitHub view raw

The loss of predictive power from this step depends on the data and nature of the stops as sometimes complex interactions of a certain order might be crucial for prediction. However, our main goal is a simple baseline interpretability and interpretability for the client.

Another useful thing to do is to check the behavior of selected sensors right before the stop and during stable working period. If you notice anomalies in your time series plots exclusively before the stops, you have a good chance of solving the problem.



Sensors just prior to the stop (look for simultaneous jumps in certain variables)

The code which generates such plots:

```
1   def visual(X, iot = ['sensor1'], batch =60 , look_b =
2     # obtain indexes
3     stops = X[(X['Response']== 1)].Time.values
4     non_stops = X[(X['delta_nxt'] >100)&(X['Y']== 1)].Ti
5     idx = np.random.choice(stops, size=batch)
6     for i in idx:
7       temp = X[(X.Time == i)|((i - X.Time ) < look_b )&(
8       plt.figure(figsize=(30,10))
9       for n,j in enumerate(iot):
10        ax = plt.subplot(4,5, n+1) # here x*y should equ
11        ax.set_title(j[5:])
12        ax.plot(temp[j].values)
13       plt.tight_layout()
14       plt.show()
15
```

plots1.py hosted with ❤ by GitHub view raw

https://gist.github.com/ikleni/95d93442f1042d1c7e3bf392d4322a30

Modeling

Thoughts

Given well-behaved **(X,Y)** (I assume that you have done other standard data preprocessing, such as dealing with missing

values) data we can start the modeling stage.

Prior to testing some models, we can hypothesize about where the signal is contained. Machines are designed in order to operate normally. Also, they are prescribed with a stable operating regime indicating an interval of values for each sensor. If for some reason (engineers did not change a certain parameter smoothly) machine deviates from the prescribed operating regime it is likely to break. With time, machine becomes less stable and the operating regime should account for that shift. Rarely does this happen — this might be our bread and butter for this task.

I expect the stops to be well predicted by deviations for each sensor from some new range of values. Also, I expect that interactions between several sensors will predict stoppages. For instance, if one sensors deviates from its normal interval while another has value w, no stoppage will occur. However, if that same sensor deviates while another has value z, the stoppage is unavoidable.

I do not expect (this assumption was good for my task) that interactions between a large number of sensors will be helpful for this task. Especially, if we quantify the aggregate number of sensors which are currently out of their standard range of values. This overall machine stability should be one of the key features in the model.

Approach

Having all this in mind, I chose to predict stops with a majority voting ensemble of small models, based on 2–3 sensors and aggregate features. This approach allows to identify which sensors signal the stop. This information can be later directed to engineers, who, based on their knowledge, can identify the cause and fix the problem.

First, I select a set of pairs (triples) of sensors from our restricted set using basic LGBM model with specifically tuned parameters. Selection is done via iterative search over pairs, according to the AUC score on k-fold time series cross-validation. Besides sensor data, those small models include aggregate features and some intuitive sensor-based features, such as:

- Mean
- Trend
- Z-score
- Std

Code for variable generation:

```
1  def feature_creation(temp, i):
2      names = []
3      temp[i+'_'+'mean' + '_' + '60'] = temp[i].shift(1
4      names.append(i+'_'+'mean' + '_' + '60')
5
6      temp[i+'_'+'std' + '_' + '60'] = temp[i].shift(1 )
7      names.append(i+'_'+'std' + '_' + '60')
```

```
8
9        temp[i+'_'+'std' + '_' + '10'] = temp[i].shift(1 )
10
11       temp[i+'_'+'var_diff'] =temp[i+'_'+'std' + '_' + '
12       names.append(i+'_'+'var_diff')
13
14       temp[i + '_'+ 'z_score'] = (temp[i] - temp[i+'_'+'
15           temp[i+'_'+'std' + '_' + '60']+1)
16       names.append(i + '_'+ 'z_score')
17
18       temp[i+'_'+'trendL'] = temp[i].diff().rolling(60 ,
19       names.append(i+'_'+'trendL')
20
21       temp[i+'_'+'trend'+'_'+'short'] = temp[i].diff().r
22       names.append(i+'_'+'trend'+'_'+'short')
23
24       temp = temp.dropna()
25       return names
```

feature_creation.py hosted with ❤ by GitHub view raw

After determining the set of small models, their predictions are combined by basic ensembling. 0.5 quantile is found and if it is higher than a predefined boundary, I predict Y = 1 and the stop should soon occur. **Alarm!**

The above approach is slightly simplified and more can be done to improve its precision, such as:

- Other ensembling techinques (RNN and other …)
- Other first level learners

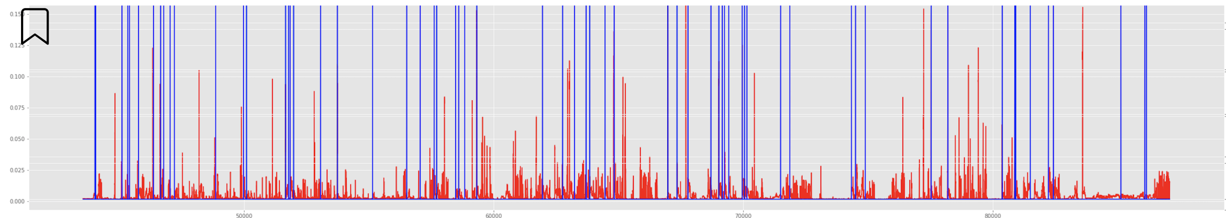
- Search over leftover sensors and add them to some of the small models
- Build a single model for each production regime (summer/winter substantially differ in the distributions of sensor values, this can be checked via Kolmogorov-Smirnov test)

Towards Data Science
Sharing concepts, ideas, and codes.
Follow

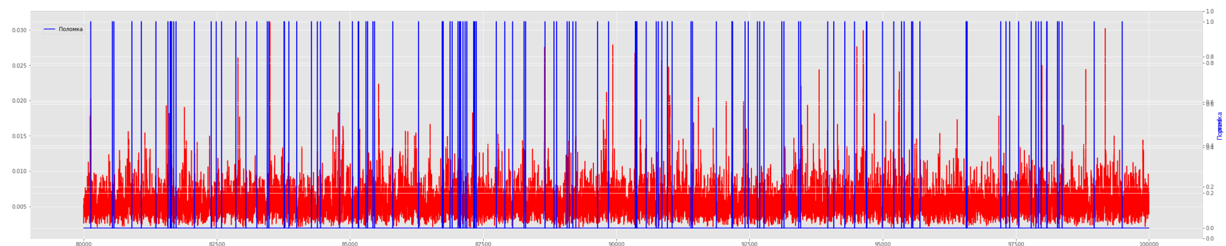
Results

Images, in my view, here are more revealing than words or metrics, if you see that algorithm rarely predicts stops but when in does something indeed happens with the machine — we are fine:

22



Good performance of the baseline



Bad performance

On my data the described framework was able to predict stops with a precision of about 40% and a recall of around 40%, which is a good result for a baseline in predictive maintenance.

Moreover, it provides engineers with references to particular sensors, as they know which sensors signal a higher likelihood of stopping. Thus, they can do a narrowly focused preventive maintenance.

I will end with a short summary of the pros and cons of this approach:

Pros

- Simplicity and interpretability (with such a small number of sensors per model, you can emphasize what part of the machine drives the threat of stop)
- Low constraints on GPU usage and memory

Cons

- Suboptimal in terms of accuracy (well trained NN with all the features might outperform this baseline)
- Computation time constraints (all those iterative searches require time)
- Human time constraints (this approach requires human guidance)

I will add links to the omitted code later. Also, I give credit to Kaggle Bosch competition, which I used to create all those images…