Model documentation and write-up

You can respond to these questions either in an e-mail or as an attached file (any common document format is acceptable such as plain text, PDF, DOCX, etc.) **Please number your responses.**

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

   Most of my career was in software development : from writing simple tools to building large-scale IT systems. Currently I split my working time between two jobs : teaching (I teach children robotics at Moscow school 654) and fintech (building machine learning models to predict future market prices).

   If you are on a team, please complete this block for each member of the team.

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

   First, I did a lot of preprocessing (history-based aggregate functions calculation) of original data to produce features for each point in train and test sets. Then, these features are used to train gradient boosting (LightGBM) model.

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

   The following (slightly simplified version) function is used in data preprocessing to create aggregate features from history. It is the main part of the whole pipeline.

```
def get_aggregates(df, TestTimestamp, period, target_col, cols, func_list, offset_name, col_values, group_cache, noval_name = ''):


  if (tuple(cols), target_col, col_values) in group_cache:
    subset = group_cache[(tuple(cols), target_col, col_values)]
  else:
    if tuple(cols) in get_aggregates.gb_cache:
      gb = get_aggregates.gb_cache[tuple(cols)]
    else:
      if len(cols):
        gb = df.groupby(cols)['Value','Temperature']
        get_aggregates.gb_cache[tuple(cols)] = gb

    if len(cols):
      if col_values in gb.groups.keys():
        subset = gb.get_group(col_values)[target_col] #.set_index('Timestamp')  # Slice with the current values in the corresponding columns
      else:
        subset = df.iloc[0:0] # empty slice
    else:
      subset = df # No slicing, using all data
    group_cache[(tuple(cols), target_col, col_values)] = subset

  first_idx = np.searchsorted(subset.index, TestTimestamp - period[0])
```

```
  last_idx = np.searchsorted(subset.index, TestTimestamp)
  subset = subset.values[first_idx : last_idx]


  postfix = (''.join(cols))+period[1]

  res = OrderedDict()
  for func in func_list:
    if target_col != 'Value':
      agg_name = func[1]+target_col+'By'+postfix+offset_name+noval_name
    else:
      agg_name = func[1]+'By'+postfix+offset_name+noval_name
    if len(subset) == 0:
      res[agg_name] = np.nan   # zero-len subset, unable to calculate
    else:
      res[agg_name] = func[0](subset)  # calculating

  return res
get_aggregates.gb_cache = {}   # static variable with groupby caches
```

In function process_forecastid, random lag used for each sample in the training set. This allows (to some extent) to simulate test situation when we don't have access to the most recent data, and the last data available is lagged by [1..forecast_period_len] steps. Using these lags turned out to be very important to prevent overfitting to the training set

```
for idx, row in train_subset.iterrows():
  aggs = OrderedDict()
  lag = np.random.choice(forecast_len)   # random lag for each train sample
  aggs['lag'] = lag
  aggs['dist_from_start'] = dist_from_start
  dist_from_start += 1
  lag_time = lag*time_step
  for c, p, target_col, offs, func_list in filters:
    if len(c):
      vals = row[list(c)]
      vals = tuple(vals) if len(vals) > 1 else tuple(vals)[0]
    else:
      vals = None
    col_aggs = get_aggregates(train_to_agg, row.Timestamp-offs[0]-
lag_time, p, target_col, c, func_list, offs[1], vals, group_cache)
    aggs.update(col_aggs)
    if row.Validation == 1:           # if we need to calculate noval aggregates as wel
l
      noval_aggs = get_aggregates(train_to_agg, noval_Timestamp-
offs[0], p, target_col, c, func_list, offs[1], vals, group_cache, '_noval')
      noval_aggs['lag_noval'] = lag_noval
      lag_noval += 1
      aggs.update(noval_aggs)

  train_res.append((row.original_index, aggs))
```

Split function in lgb.py :
In this function, features in the validation set are replaced by '_noval' versions, which are lagged not randomly, but in a sequential order, as in the test set. Again, this allows reproducing test situation when data are available with the same lags.

```
for c in val_df.columns:
  if '_noval' in c:
    val_df[c[:-6]] = val_df[c]   # replacing values in validation set to exclude information not available in validation period
val_df = get_ratio_features(val_df)
```

blend method in lgb.py:

During training, different models with slightly different hyperparameters are generated, and blending (averaging on linear or log scale) them allows to improve overall prediction quality

```
def blend(old, new, rem_idx, use_log):
  updated_list = old.copy()
  if rem_idx != -1:
    updated_list = updated_list[:rem_idx] + updated_list[rem_idx+1:]
  updated_list.append(new)
  blended = np.stack(updated_list)
  if use_log:
    blended = np.log1p(blended)
  blended = blended.mean(axis = 0)
  if use_log:
    blended = np.expm1(blended)
  return blended, updated_list
```

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

1D CNNs. When raw consumption history is used as input to the CNN layers, and pre-calculated features are added a s inputs to the following dense layers - it produces some promising results, but I didn't have enough time to fully explore this option.

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

I just used a very simple Python script to plot historical data.

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

   It depends upon what is important in the real life. The current metric penalizes over predictions more than under predictions (because division by average consumption is used). If this property is not desirable, I would suggest using something like RMSLE (root mean squared log error) or any other metric involving logarithms.

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.)?

   At least 64Gb of RAM is recommended. Sometimes there is a spike in memory usage in the very beginning of training a model with LightGBM 2.0.10. If training crashes at this moment, the following may help : adding swap partition or using 'histogram_pool_size' parameter with LightGBM. The later may affect model prediction quality, however. The current version of training script limits histogram_pool_size when less than 48Gb of RAM is available.

   Preprocessing takes 4..5 hours on 16 vCores machine, model training takes 12..16 hours. Changing parameter NUM_ATTEMPTS in lgb.py allows to train faster, but the prediction quality will be generally lower.

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

   I think I do not, as I used only very simple charts to visualize original data.

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?

   First, I would try further exploiting the stochastic nature of this pipeline. Preprocessing results are different after each run, however currently only one version is used. It is possible to at least average models trained on different preprocessed data, but it may be even better to merge preprocessing data with different random lags and use it as an 'augmented' train set.

   Second, I would explore global aggregation. Currently, all aggregates are calculated with respect to specific SiteId, but there may me some patterns in global usage across all sites (or some clusters) worth exploring

   Then, I would try to use neural networks (both convolution and recurrent). In fact, I already tried 1d CNNs, and the preliminary results are promising. These models are (at least in theory) capable to capture some more complex patterns based on raw time series of recent consumption, this may be particularly useful when predicting less periodic time series with no clear weekly/hourly dynamics.