|  |  |  |
| --- | --- | --- |
| **Spring 2024** | **Report #3– 04/21/2024** | **Dingyi Nie** |

**——————————————————————————————————————**

**I. Task Achieved Last Week**

**——————————————————————————————————————**

* Examined the convergence of the XGBoost imuputation strategy.
* Preliminarily tested GL classifier (via multi-layer Saab module + logistic regression)’s performance on the same dataset.

**——————————————————————————————————————**

**II. Feedback and Interaction**

**——————————————————————————————————————**

* **Prof. Kuo’s Feedback**
  + Padding and alignment: make sure the end is aligned
  + Decision based on windowed intervals (window len = avg seq len)
  + In certain layer, combine all channels together

**——————————————————————————————————————**

**III. Report**

**——————————————————————————————————————**

As suggested by Prof. Kuo last week, my first job for this week is to test running multiple iterations of our XGBoost imputer and see if it converges.

Since XGBoost only works on batched inputs with a fixed feature length, we take windowed “patches” (neighboring observations) of a missing observation, flatten into a 1D vector, and replace the missing observation as NaN (XGBoost has the ability to handle NaNs in features). Here we have 2 different strategies for our first imputing iteration:

* We fit an XGBoost regressor directly with training features containing **only observed sensor values**, i.e. all un-observed positions remain NaN, and we rely on XGBoost’s capability to handle these;
* We impute the features with **linear / spline** imputer first, then we fit an XGBoost based on these imputed features, i.e. the only NaN in input feature is the center position to be regressed to.

Then, for the next iterations, we simply use the re-imputed features from the last run.

I tried running 5 iterations with both strategies (per iteration takes about 4~5 minutes). In second strategy, linear imputer is used. I found out that for the 2nd~5th iterations, the initially missing values hardly change (average MSE is below **1e-2** per position), meaning that once missing values are imputed, they can hardly affect how neighboring missing values will be imputed in the following iterations. So I ended using the results from the 1st iteration only.

Next part of my work would include testing the classification performance of GL methods on these imputed datasets. My initial plan was to use PixelHop or IPHop (GL frameworks that incorporate ensemble learning). However the code I collected seems to still be incomplete, so I ended up using a multi-layer Saab feature extraction model (mainly referring to [this](https://github.com/yunchengwang/python-feature-test/tree/main)) to do dimensionality reduction and a logistic regression for classification. Detailed dataflow can be described as follows:

1. P19 has initially ~40 sensors, only 7 of them have a missing rate lower than 70%. We use those sensors’ observations as original input features. We divide the dataset into a train subset (~36000 samples) and a test subset (~4000 samples) randomly. The max sequence length in P19 (number of rows of observations) was 336, so the input feature is shaped (BATCH\_SIZE, C=1, 336, 7). **This conversion is actually non-trivial**, but here I’m simply padding those short features with 0’s to have the same length of 336.
2. 4 cwSaab layers in sequence:
   1. Kernel size = (4,1), stride = (4,1): this reduces the last 3 dimensions into (4, 84, 7)
   2. Kernel size = (4,1), stride = (4,1): this reduces the last 3 dimensions into (16, 21, 7)
   3. Kernel size = (4,2), stride = (4,2): this reduces the last 3 dimensions into (128, 5, 3)
   4. Kernel size = (4,2), stride = (4,2): this reduces the last 3 dimensions into (1024, 1, 1)
3. Fit the Saab module with train set. By far we get a reduced feature of length 1024 for each sample. I then fit a simple logistic regressor for binary classification.
4. Since P19 is highly imbalanced (~94% of labels are negative), I use AUROC as evaluation metric. Finally, we evaluate the model with test set.

Here are the results (rows in grey are baseline competitors, reported in [Raindrop](https://arxiv.org/pdf/2110.05357.pdf)):

|  |  |
| --- | --- |
| Method | Binary classification AUROC |
| Linear imputation | 81.9% |
| XGBoost imputation (strat. 1) | **84.1%** |
| XGBoost imputation (strat. 2) | 84.0% |
| Transformer | 83.2% |
| Transformer-mean | 84.1% |
| GRU-D | 83.9% |
| SeFT | 78.7% |
| mTAND | 80.4% |
| Raindrop | **87.0%** |

As shown above, our XGBoost imputer indeed out-performs linear imputation! Among deep learning competitors, although the newest Raindrop (GNN-guided, 2023) has a higher 87.0% AUROC on P19, our completely GL-based model with imputation strategy 1 still out-performs most of them.

**——————————————————————————————————————**

**IV. Next Steps**

——————————————————————————————————————

* Find a better strategy for handling irregularly shaped time series as image-like input arrays, as targeted by most existing GL frameworks.
* Get the code done for experiment with IPHop.
* Maybe find a way to integrate demographic (static) info about each patient (time series episode) into classification pipeline.

**——————————————————————————————————————**

**V. Milestone**

——————————————————————————————————————

* As stated in I.
* Codes are available here: <https://github.com/d9sus4/GL-TS>