Document Type	Demonstration
Title	Multi-Class Classifier
Version	1.0
Publication Date	3 March 2021
Author	Spyros Martzoukos
Abstract	Multi-Class classifier. Build, evaluate and deploy.
Keywords	Feedforward Neural Networks XGBoost Optuna stratified sampling bootstrap aggregating

Contents

1 Input data & pre-processing

The raw input matrix `input_data.csv` contains 4294 rows and 97 columns. The latter consist of 96 feature columns and one target column. The target column, termed `status`, contains categorical values and there are five such distinct values (classes). Our goal is to build a classifier that predicts the `status` value of an unseen row (in the format of the raw input matrix).

The pre-processing steps that we consider are as follows:

- If a row contains NaN value for the `status` column, then it is removed from the matrix. One such row is thus removed.
- Each of the five categorical values of the `status` column is mapped to a distinct integer, i.e., a *label*.
- In total, the number of NaN entries is significant (> 4,000). If a feature column has more than 50% of its entries as missing values (NaN), then it gets dropped from the matrix. The feature `kubelet_pod_worker_duration_bucket` is thus dropped.
- Regarding the remaining NaN entries, we exploited the fact that the data consist of time series: A missing value of column `x` is replaced by linear interpolation of its time-wise nearest values from column `x`.
- No action is taken for the `timestamp` column (it gets dropped during the construction of the models).

Information regarding the `status` column can be found in Table 1. The severe class imbalance is taken into account during our modelling process.

Label	Status	Frequency	Probability
0	NORMAL	2757	0.6422
1	CPU_BURN_KANBAN_API_GATEWAY	80	0.0186
2	NETWORK_DELAY_KANBAN_API_GATEWAY	646	0.1505
3	NETWORK_DELAY_KANBAN_COMMAND_SERVICE	500	0.1165
4	POD_KILL_KANBAN_API_GATEWAY	310	0.0722

Table 1: Target column information

Finally, we note that although the data are time series, the `timestamp` is only piecewise monotonic. This implies that the raw input matrix was formed by stacking batches of data from different time periods. An approach that builds on the strict monotonicity of the `timestamp` data would require a row-reordering of the raw input matrix. However, this is not a concern for our approach as we treat each row independently from any other row.

2 Methodology & algorithms

With the processed data matrix in hand (henceforth the *input matrix*), our approach for building the classifier is based on a standard train-tune-test method.

First, a *test set* is selected from the input matrix and then removed from the input matrix. The test set is a sub-matrix of the input matrix and, in particular, it consists of 20% of the total rows of the input matrix. The test set's rows are sampled randomly (without replacement) from the input matrix, but we ensure that the formed set preserves the distribution of the target values in the input matrix. In other words, status value `NORMAL` also has $\sim$64% probability of appearing in the test set, and similarly for all other classes. This is the so-called *stratified sampling* approach. After its formation, the test set is isolated from the subsequent process and is only used during evaluation (see Section 3).

The remaining rows from the input matrix (henceforth the *training matrix*) are used to form the *train* and *tune* sets. These sets are used to build the classifier. Given that the training matrix has few rows, we follow the *bootstrap aggregating* (or *bagging*) approach. This means that we firstly build k classifiers $C_1,\ ..., C_k$ from the training matrix and the final classifier is given by an ensemble of $C_1,\ ..., C_k$.

More precisely, a classifier C_i is built independently from any other classifier C_j, $i \neq j,\ i, j = 1,\ ..., k$; the process for *fold* i is as follows:

1. From the training matrix, we sample (without replacement) 80% of its rows and form the train set of fold i. As with the test set, sampling is performed in a stratified way. The remaining rows are used to form the tune set of fold i. Note that, for each fold, the same, original training matrix is used to form the train and tune sets. Consequently, any two train sets of different folds may have overlapping entries, which is desirable and part of the bagging process.
2. Using the train & tune sets of fold i and a classification algorithm, say `algo`, a classifier $C_i^{\texttt{algo}}$ is built. Optuna is used to identify optimal hyper-parameters for the classifier by maximizing the *balanced accuracy* metric.

In our experiments, the number of folds, k, is set to five. The final classifier with respect to algorithm `algo` is given by majority voting of the per fold classifiers. More precisely, if x denotes a feature vector, then the output of the per fold classifiers is given by

$$y_i^{\texttt{algo}} = C_i^{\texttt{algo}}(x),\ \ i = 1,\ ..., k. \tag{1}$$

The final classification output is thus given by

$$y^{\texttt{algo}} \ = \ C^{\texttt{algo}}(x) \ \equiv \ \text{MajorityVoting}(y_1^{\texttt{algo}},\ ..., y_k^{\texttt{algo}}), \tag{2}$$

where MajorityVoting() is a function that maps a multi-set of integers to the most frequent integer in the multi-set. If multiple algorithms are employed, say $\texttt{algo}_1,\ ..., \texttt{algo}_m$, then one can go further and compute

$$y \ = \ C(x) \ \equiv \ \text{MajorityVoting}(y^{\texttt{algo}_1},\ ..., y^{\texttt{algo}_m}). \tag{3}$$

Note that equations (2) and (3) are meaningful only when $k > 2$ and $m > 2$, respectively. We experiment with two algorithms, namely Feedforward Neural Networks (FNNs) through PyTorch and XGBoost. Tables 2 and 3 summarize information about the hyper-parameters that are considered for FNNs and XGBoost, respectively.

The classifier building process can be summarized as follows: Through equation (1) we construct the per fold classifiers $C_i^{\texttt{FNN}}$ and $C_i^{\texttt{XGB}}$, respectively, for all $i = 1, ..., k$. For each fold and for each algorithm, optimal hyper-parameters are identified with Optuna. Finally, through equation (2) we construct the per algorithm classifiers, namely $C^{\texttt{FNN}}$ and $C^{\texttt{XGB}}$. The best classifier is identified through evaluation.

Feedforward Neural Networks

Hyper-parameters to be optimized		
Name	Search space	Value type
Number of hidden layers	{2, 3}	int
Units per hidden layer	[5, 515]	int
Optimizer for backpropagation	{Adam, SGD}	categorical
Dropout probability	[0.2, 0.5]	float
Learning rate	[1e-5, 1e-1]	float
Epochs	[10, 100]	int
Batch size	{32, 64, 128}	int

Fixed values for hyper-parameters	
Name	Value
Activation function	ReLU
Units in Input layer	94
Units in Output layer	5
Scale of features	Standard scale $(x - \mu)/\sigma$
Error metric	Negative log-likelihood loss

Table 2: Hyper-parameters for FFNs. 200 Optuna trials for identifying optimal values.

XGBoost

Hyper-parameters to be optimized		
Name	Search space	Value type
λ	$[\exp(-9), \exp(2)]$	float
α	$[\exp(-9), \exp(2)]$	float
γ	$[\exp(-16), \exp(2)]$	float
Subsample ratio	[0.5, 1.0]	float
Max tree depth	[5, 9]	int
Learning rate	$[\exp(-7), 1.0]$	float
Grow policy	{depthwise, lossguide}	categorical
colsample_bytree	[0.5, 1.0]	float
colsample_bylevel	[0.5, 1.0]	float
min_child_weight	$[\exp(-16), \exp(5)]$	float

Fixed values for hyper-parameters	
Name	Value
Number of decision trees	10
Scale of features	True scale
Error metric	Multi-class cross-entropy loss

Table 3: Hyper-parameters for XGBoost. 500 Optuna trials for identifying optimal values.

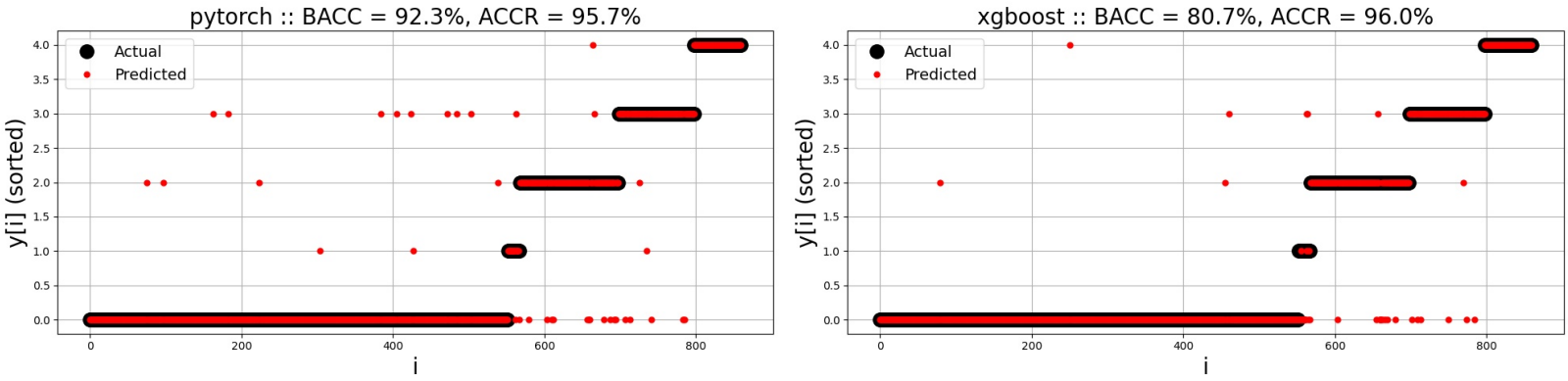
3 Evaluation & results

The test set is used to evaluate the two classifiers, namely $C^{\texttt{FNN}}$ and $C^{\texttt{XGB}}$. Each row in the test set consists of the features values, or feature vector x, and its corresponding, true target value $y^{\texttt{true}}$. Upon input x, the classifier $C^{\texttt{algo}}$ produces an estimate $y^{\texttt{algo}}$. Our aim is to compare $y^{\texttt{true}}$ with $y^{\texttt{algo}}$ for all possible inputs x, i.e., for all rows in the test set, thus producing a score for algorithm `algo`. This process is performed for both algorithms `FNN` and `XGB`. The best classifier is identified by comparing their scores.

The error metric that we use to evaluate a classifier K is *balanced accuracy*, or BACC(K) for short. For completeness, we also report the accuracy score, or ACCR(K) for short.

Our best results are achieved when we consider only a subset of the five per fold classifiers. More precicely, if the classifier of fold i achieves BACC($C_i^{\texttt{algo}}$) $< \theta^{\texttt{algo}}$, for a predefined threshold $\theta^{\texttt{algo}}$, then it gets discarded. Thus, equation (2) is executed using only the per fold classifiers that pass the threshold test.

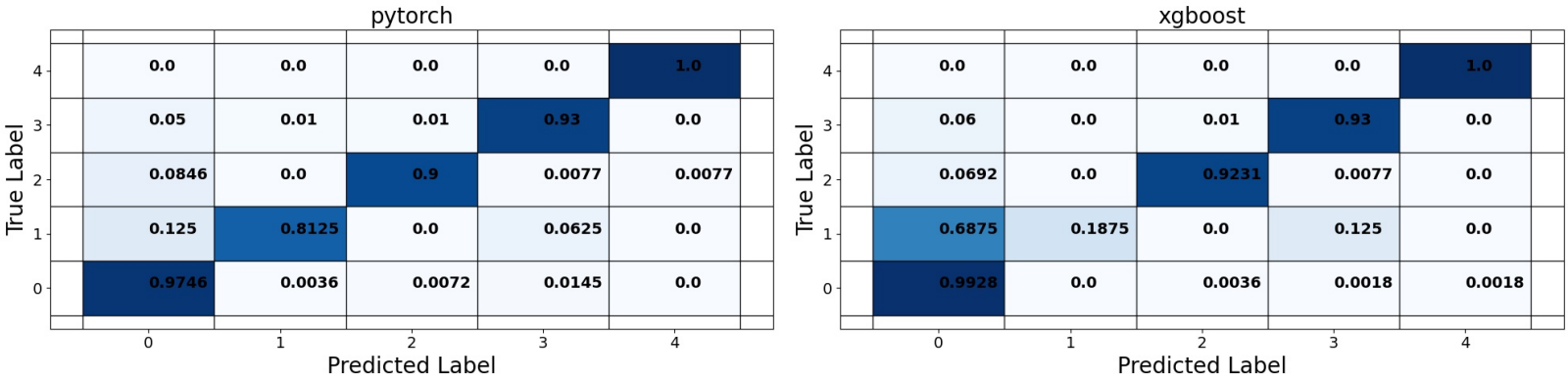
Our results are summarized in Figures 1 & 2.



(a) Feedforward Neural Network

(b) XGBoost

Figure 1: Results for bagging classifiers.



(a) Feedforward Neural Network

(b) XGBoost

Figure 2: Confusion matrices for bagging classifiers.

It terms of ACCR score, both classifiers perform fairly well with comparable results.
In terms of BACC score, $C^{\texttt{FNN}}$ performs better than $C^{\texttt{XGB}}$, mainly because it is better at classifying class/label 1, or `CPU_BURN_KANBAN_API_GATEWAY`. On the other hand, $C^{\texttt{XGB}}$ requires x20 less training + hyper-optimization time than $C^{\texttt{FNN}}$.

Finally, we mention that the whole train-tune-test process was repeated using a different test set. We observed similar results to the ones in Figures 1 & 2, which implies that our approach is fairly robust.

4 Code

The repository `cloudaeye_classifier` consists of three packages: Package `build` for building the classifier, package `evaluate` for evaluating the classifier, and package `deploy` for an example of how the classifier is used in practice.

The execution order is as follows:

1. Script for building the classifier: `build\build.py`
   It constructs the following:
   - Directories: `optuna_dump`, `StandardScaler`
   - Files: `dropped_features.csv`, `test_set.csv`, `label2status.pkl`
2. Script for evaluating the classifier: `evaluate\evaluate.py`
   It constructs the following:
   - Directories: `predictions`, `aggregated_predictions`
   - Files: six .jpg figures
3. Script for deploying the classifier: `deploy\deploy.py`
   Output, i.e., predicted labels with their corresponding timestamps, is printed to the terminal.

Model construction is by default set to 'demo version' for quick execution ($\sim$3 mins). For the fully operational version, the flag `IS_DEMO_VERSION` should be set to `False` in the `config.py` file in each of the packages. In this case, the number of Optuna trials is set to 200 and 500 for FNN and XGBoost, respectively. The execution time is $\sim$3 hrs.

5 Next steps

The focus of next steps is around class/label 1, or status `CPU_BURN_KANBAN_API_GATEWAY`. How do the data behave before this status appears? What happens if we remove rows with `NORMAL` status (or at least a big portion) from the data and build classifiers on the remaining data? Finally, our current approach contains almost no knowledge of the features' nature (other than them being time series). A more thorough approach could benefit from such an investigation.