pynolca: Reference Manual

Xingke Chen

November 26, 2019

Abstract

pynolca is a computationally efficient classification package in Python with noise-resilient functionality. The main advantage of **pynolca** is that it is not only able to deal with conventional classification and binary semi-supervised classification tasks in an online learning fashion efficiently but also suitable for noisy scenarios with the help of noise-resilient loss function. Besides, the kernel module of this package allows users to construct complex, non-linear models, which can promote the classification accuracy significantly. To ease the burden of users, the package provides preprocessing module to tackle some typical problems independent of classifications.

1 Model and Algorithm

pynolca aims to tackle multiclass classification problems (including binary classification). Given a c-ary classification problem, where training data $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n) \in \mathcal{X} \times \mathcal{Y}$ with $\mathsf{card}(\mathcal{Y}) = c$, **pynolca** tries to solve the following optimization model:

$$\min_{f_1,\ldots,f_c \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell^s(\mathbf{f}(\mathbf{x}_i), y_i) + \frac{\lambda}{2} \sum_{k=1}^{c} \|f_k\|_{\mathcal{H}}^2, \tag{1}$$

where the components of the model are described one by one:

- The discriminate function $\mathbf{f}$ is a c-dimensional vector $\mathbf{f} = (f_1, f_2, \ldots, f_c)$ such that the larger f_k gives more confidence to classify the instance as k-th class. That is, $\mathbf{f}(\mathbf{x}) = \arg\max_{k\in\{1,2,\ldots,c\}} f_k(\mathbf{x})$.
- The loss function ℓ^s can be
  - Multiclass ramp loss function

    $$\ell^s_{MR}(\mathbf{f}(\mathbf{x}), y)) = \min\Big\{1 - s, \max\{1 - \big(f_y(\mathbf{x}) - f_{\hat{y}}(\mathbf{x})\big), 0\}\Big\}, \tag{2}$$

    which is defined according to the one-versus-all strategy proposed by [1], where $\hat{y} = \arg\max_{k\in\mathcal{Y}, k\neq y} f_k(\mathbf{x})$. Parameter $s \leq 1$ controls the behavior of the ramp loss.
  - Binary semi-supervised loss

    $$\ell^s_{BSS}(\mathbf{f}(\mathbf{x}), y) = \begin{cases} \ell^s_{MR}(\mathbf{f}(\mathbf{x}), y), & y \neq -1 \\ \max\left\{1 - |f_1(\mathbf{x}) - f_2(\mathbf{x})|, 0\right\}, & y = -1, \end{cases} \tag{3}$$

    where $\mathbf{f} = (f_1, f_2)$ and $y = -1$ means the instance is unlabeled.

- The reproducing kernel Hilbert space $\mathcal{H}$ in which each discriminant function f_k lies is specified by some kernel $\mathcal{K}$ ([2]).
- $\lambda > 0$ is the regularization parameter.

pynolca solves this problem under the online learning framework called NORMA algorithm ([4]). This algorithm gives rise to the discriminant function

$$\mathbf{f}^{(i)} = \begin{bmatrix} f_1^{(i)} \\ f_2^{(i)} \\ \vdots \\ f_c^{(i)} \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_{11}^{(i)} & \alpha_{12}^{(i)} & \cdots & \alpha_{1i}^{(i)} \\ \alpha_{21}^{(i)} & \alpha_{22}^{(i)} & \cdots & \alpha_{2i}^{(i)} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{c1}^{(i)} & \alpha_{c2}^{(i)} & \cdots & \alpha_{ci}^{(i)} \end{bmatrix}}_{\mathbf{A}^{(i)}} \underbrace{\begin{bmatrix} \mathcal{K}(\mathbf{x}_1, \cdot) \\ \mathcal{K}(\mathbf{x}_2, \cdot) \\ \vdots \\ \mathcal{K}(\mathbf{x}_i, \cdot) \end{bmatrix}}_{\mathbf{k}^{(i)}} \tag{4}$$

at i-th step and

$$\alpha_{kj}^{(i)} = \begin{cases} \eta \dfrac{\partial \ell(\mathbf{f}(\mathbf{x}_i), y_i)}{\partial f_k(\mathbf{x}_i)}\Big|_{\mathbf{f}=\mathbf{f}^{(i-1)}}, k = y_i, j = i \\ -\eta \dfrac{\partial \ell(\mathbf{f}(\mathbf{x}_i), y_i)}{\partial f_k(\mathbf{x}_i)}\Big|_{\mathbf{f}=\mathbf{f}^{(i-1)}}, k = \hat{y}_i, j = i \\ 0, k \notin \{y_i, \hat{y}_i\}, j = i \\ (1 - \eta\lambda)\alpha_{kj}^{(i-1)}, \forall k \in \mathcal{Y}, j < i, \end{cases} \tag{5}$$

whence $\eta \in \left[0, \frac{1}{\lambda}\right)$ is the learning rate. For the unity, we need to set $y_i = 1$, $\hat{y}_i = 2$ if $0 < f_1(\mathbf{x}) - f_2(\mathbf{x}) < 1$ and $y_i = 2$, $\hat{y}_i = 1$ if $-1 < f_1(\mathbf{x}) - f_2(\mathbf{x}) < 0$ in binary semi-supervised classification. We summarize the procedure of the Noise-resilient Online Large-scale Classification Algorithm in Algorithm 1.

Algorithm 1 Noise-resilient Online Large-scale Classification Algorithm

Input: $\mathcal{S} = \{(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n)\}$, kernel $\mathcal{K}$, loss function parameter s, regularization coefficient $\lambda > 0$, learning rate $\eta \in [0, \frac{1}{\lambda})$.
Output: Predictions $\{\hat{y}_i\}_{i=1}^n$.
1: Initialize weights $\mathbf{A}^{(0)} = 0$, $\mathbf{k}^{(0)} = 0$.
2: **for** $i = 1$ to n **do**
3: Receive the data $(\mathbf{x}_i, y_i)$
4: Predict $\hat{y}_i = \arg\max_k \mathbf{A}^{(i-1)}\mathbf{k}^{(i-1)}(\mathbf{x}_i)$
5: Compute loss $\ell^s(\mathbf{A}^{(i-1)}\mathbf{k}^{(i-1)}(\mathbf{x}_i), y_i)$.
6: Update $\mathbf{A}^{(i)} = (\alpha_{kj}^{(i)})$ according to Equation (5).
7: Append $k(\mathbf{x}_i, \cdot)$ to $\mathbf{k}^{(i)}$.
8: **end for**

2 Software Architecture

pynolca is organized as five modules, each of which provides specific services for the classification tasks respectively:

1. `preprocessing.py` - a module for implementing simple data preprocessing. The methods in this module are `shuffle` for randomizing datasets, `encoder`, `decoder` for converting original labels into valid labels in **pynolca**.
2. `kernel.py` - a module for data transformation with kernel method. Classes `Linear_Kernel`, `Polynomial_Kernel` and `RBF_Kernel` corresponding to the kernels in Table 1 are incorporated in this module.
3. `loss.py` - a module for calculating the loss while training the classifier, containing the class `Ramp`.
4. `nolca.py` - core module, of which class `NOLCA` is in charge of model training, prediction and visualization.
5. `utils.py` - a bottom module consists of a bunch of internal tool functions that are utilized to provide service for other modules.

3 Software Implementation

This sequel describes the details of the classes implementation in **pynolca**. We explain these classes separately according to the modules they belong to.

preprocessing.py This module allows users to preprocess the original dataset, there are three functions in this module that are used frequently in practice.

Generally, the order of training data may influence the classification performance significantly, especially in online learning settings. For example, usually the data are sorted by the labels. Hence it is possible that most of the training data belong to some specific classes, which leads to an unbalanced classification problem. To tackle this problem, the function `shuffle()` is used to introduce randomness and disorder the original dataset so as to avoid unbalanced labels in the training stage. The function

```
def shuffle(X, y):
```

is able to randomize the dataset `(X, y)`, where `X` is the matrix of predictor variables, and `y`, the labels for `X`. `shuffle(X,y)` returns the randomly reordered dataset $(\texttt{X}^\star, \texttt{y}^\star)$.

One can invoke `shuffle()` in the following scheme:

```
>>> from pynolca import preprocessing
>>> from sklearn import datasets
>>> data = datasets.load_iris()
>>> X = data.data
>>> y = data.target
>>> X_star, y_star = preprocessing.shuffle(X, y)
```

Kernel name	Kernel expression
Linear kernel	$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \langle \mathbf{x}, \mathbf{x}' \rangle$
Polynomial kernel	$\mathcal{K}(\mathbf{x}, \mathbf{x}') = (a\langle \mathbf{x}, \mathbf{x}' \rangle + b)^p$
Radial basis function kernel	$\mathcal{K}(\mathbf{x}, \mathbf{x}') = \exp(-d\|\mathbf{x} - \mathbf{x}'\|^2)$

Table 1: Kernels in **pynolca**.

Moreover, original labels of realistic datasets may not be the data type that the classifier is able to deal with (valid labels take values in $\mathcal{Y} = \{0, 1, \ldots, c-1\}$ and -1 for unlabelled in **pynolca**). It is necessary to convert the original labels (generally strings) into the valid ones. Hence, we implement the following `encoder()` and `decoder()` functions to accomplish this convert. The definitions of the pair of functions are as follows.

```
def encoder(seq, unlabelled_mark = " "):
```

Function `encoder` converts string type or other type labels to valid labels in this package, where `seq` is the original label vector and `unlabelled_mark`, default the blank space, is the label representing "unlabelled", which should be converted to -1. `encoder` returns a tuple consists of the transformed labels `encoded_seq` and the mapping `encoder_dict` : `seq` $\mapsto$ `encoded_seq`.

```
def decoder(encoded_seq, encoder_dict):
```

This function is the inverse mapping of `encoder(seq)`, so it converts the encoded labels to original labels. The argument `encoded_seq` is the predictions thrown by classifier. It returns the original labels with respect to `encoded_seq` according to `encoder_dict`. One can convert labels and recover labels in such a way:

```
>>> from pynolca import preprocessing
>>> from sklearn import datasets
>>> data = datasets.load_iris()
>>> y = data.target
### Convert the original labels to valid labels
>>> Y, D = preprocessing.encoder(y)
### Convert the transformed labels to original labels
### according to the convert dictionary D
>>> y_recover = preprocessing.decoder(Y, D)
```

kernel.py There are three kernel classes `Linear_Kernel`, `Polynomial_Kernel` and `RBF_Kernel` in `kernel.py` (see Table 1). The interfaces are listed as follows:

```
class Linear_Kernel(Kernel):
    def __init__(self):
class Polynomial_Kernel(Kernel):
    def __init__(self, scale_factor = 1, intercept = 1, degree = 2):
class RBF_Kernel(Kernel):
    def __init__(self, d = 1):
```

In class `Polynomial_Kernel`, `degree` parameter controls the expressiveness of features and `scale_factor` and `intercept` are parameters trading off the influence of higher-order versus lower-order terms in the polynomial. And in `RBF_Kernel`, `d` controls the smoothness of the kernel. In general RBF kernels can induce extremely rich expressiveness. Here are some examples of creating instances of `Kernel` object.

```
>>> from pynolca import kernel
>>> test_kernel_1 = kernel.Linear_Kernel()
>>> test_kernel_2 = kernel.Polynomial_Kernel(scale_factor = 1,
...           intercept = 2, degree = 4)
>>> test_kernel_3 = kernel.RBF_Kernel(d = 0.2)
```

loss.py `Ramp`, whose interface is given by

```
class Ramp(Loss):
    def __init__(self, parameter = -np.inf, policy = "static"):
```

has arguments `parameter`, the loss parameter s, and `policy`, the strategy to choose the ramp loss parameter. `policy` attains `"static"` or `"adaptive"`, standing for fixing the parameter as `parameter` and choosing the parameter dynamically according to the strategy proposed by [3], respectively.

The example of constructing instances of objects in `loss.py` is as follows.

```
>>> from pynolca import loss
>>> test_loss_1 = loss.Ramp(policy = "adaptive", parameter = None)
>>> test_loss_2 = loss.Ramp(policy = "static", parameter = -0.1)
```

nolca.py `NOLCA` is the core object in **pynolca**. The user's interface provided by `NOLCA` is

```
class NOLCA(object):
    def __init__(self, kernel = RBF_Kernel(), loss = Ramp(),
                 num_support_vectors = 0, support_vectors = [],
                 sample_weight = None):
    def training(self, X, y, learning_rate = 0.1,
                 reg_coefficient = 0.0, unlabelled = False):
    def predicting(self, x):
    def plot_accuracy_curve(self, x_label = "Number of samples",
                            y_label = "Accuracy"):
    def plot_confusion_matrix(self, x_label = "True label",
                              y_label = "Prediction"):
```

The model can be trained via the method `training()`, moreover, it is able to make prediction by calling function `predicting()`. After training, one can get feedback of the model performance by invoking `plot_accuracy_curve()` for obtaining the overall accuracy curve and `plot_confusion_matrix()` for the confusion matrix plot.

For the compactness of the statements, we put all of these functions and their arguments in Table 2 but the argument `args`, the abbreviation of arguments `num_support_vectors`, `support_vectors`

Interfaces	Description	Arguments	Argument Explanation
`__init__()`	Initialize the necessary parameters of the classifier.	`kernel`	`Kernel` object, the kernel type to be utilized in classification. It can take value the object `Linear_Kernel`, `Polynomial_Kernel` or `RBF_Kernel`, the default value is `RBF_Kernel`.
		`loss`	`Loss` object, the loss type to be utilized in classification, which takes value `Ramp`.
		`args`	Parameters for retraining the classifier.
`training()`	Train the classifier based on training dataset `(X,y)`.	`X`	Array-like (generally 2D **NumPy** array), the $n \times p$ matrix of covariates.
		`y`	Array-like, $n \times 1$ true label vector.
		`reg_coefficient`	Non-negative scalar, default 0.0. The regularization coefficient λ.
		`learning_rate`	Positive scalar, default 0.1. The step size $\eta \in [0, \frac{1}{\lambda})$ in online gradient descent algorithm.
		`unlabelled`	Boolean, default `False`. Indicate whether there are unlabelled observations in dataset, if some instances are unlabelled, `unlabelled` = `True` and `unlabelled` = `False` otherwise.
`predicting()`	Predict the labels of new observations.	`x`	Array-like, the new observation $p \times 1$ vector.
`plot_accuracy_curve()`	Plot the overall accuracy curve.	`x_label`	String, the name of x-axis.
		`y_label`	String, the name of y-axis.
`plot_confusion_matrix()`	Plot the confusion matrix.	`x_label`	String, the name of x-axis.
		`y_label`	String, the name of y-axis.

Table 2: The description of the class `NOLCA` in module `nolca.py`

and `sample_weight` in function `__init__()`, which is used to recover the previously obtained classifier. There is no need to change the default values of `args` when users train their classifier at the first time.

There are many parameters in the classifier object `NOLCA`, which might be useful especially when users want to inspect them and save the models. Therefore, we provide a bunch of extractors for users to obtain these parameters. All of the extractors are listed in Table 3.

4 Installation

pynolca requires some external packages in order to ensure normal operation, of which all are available open source packages. The dependencies consist of

- Python 2.7.x. Python is an interpreted, object-oriented programming language. Relying on a vast of third-party libraries, it is widely used in scientific and numeric computing.
- **NumPy** version 1.8.2 or newer. is the core toolkit for scientific computing in Python, which provides a large number of interfaces for users to manipulate the vector and matrix objects.
- **Matplotlib** version 2.0.0 or newer. The 2D visualization tool in Python.
- **scikit-learn** version 0.18.1 or newer (optional). The tool for data mining and machine learning in Python. The usage of the package is only to run the demo scripts.

The installation steps are as follows: after downloading the package, unzip the file and execute

```
$ python setup.py install --user
```

in the root directory of **pynolca** for a local installation. Or type

```
$ python setup.py install
```

for a global installation that is accessible to all users.

References

[1] Koby Crammer and Yoram Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2(Dec):265–292, 2001.

[2] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[3] Ling Jian, Fuhao Gao, Peng Ren, Yunquan Song, and Shihua Luo. A noise-resilient online learning algorithm for scene classification. *Remote Sensing*, 10(11):1836, 2018.

[4] Jyrki Kivinen, Alexander J Smola, and Robert C Williamson. Online learning with kernels. *IEEE Transactions on Signal Processing*, 52(8):2165–2176, 2004.

Extractor	Corresponding internal variable	Description
`get_weight()`	`_sample_weight`	Obtain the weight matrix **A** of current classifier.
`get_num_support_vectors()`	`_num_support_vectors`	Obtain the number of support vectors.
`get_support_vectors()`	`_support_vectors`	Obtain the collection of all the support vectors.
`get_num_error()`	`_num_error`	Obtain total number of wrong predictions.
`get_accuracy()`	`_accuracy`	Obtain the collection of accuracies of each step.
`get_confusion_matrix()`	`_confusion_matrix`	Obtain the confusion matrix after training.
`get_prediction()`	`_pred_collection`	Obtain the collection of total predictions so far.
`get_num_observations()`	`_num_obversations`	Obtain total number of observations so far.
`get_num_classes()`	`_num_classes`	Obtain the number of classes.

Table 3: The extractors in module `nolca.py`