The guide to Pycle (**Py**thon **C**ompresive **Le**arning toolbox)

Vincent Schellekens

December 10, 2019

Abstract

This is the guide to Pycle, a toolbox for Compressive Learning. It is structured as follows: first we shortly explain the theoretical methods this toolbox implements. Then, we explain how the toolbox is structured, and the main steps that a user should follow to use it. The detailed documentation of all the functionalities in the toolbox is then provided, followed by some practical examples to get started easily.

1 What is Compressive Learning?

$$\underset{\text{Dataset}}{X} \longrightarrow \underset{\text{Sketch}}{\boldsymbol{z}_X} \longrightarrow \underset{\text{Parameters}}{\Theta}$$

Figure 1: Compressive learning .

$$\boldsymbol{z}_X := \frac{1}{n} \sum_{i=1}^{n} \Phi(\boldsymbol{x}_i) \tag{1}$$

See [1] for a complete introduction to compressive learning.

2 An overview of Pycle

2.1 Requirements

The Pycle package builds on a set of standard Python libraries, that are required to run it:

- `numpy`
- `scipy`
- `matplotlib`

2.2 Typical workflow

A typical use of Pycle follows the following steps:

1. Design a sketch operator, then sketch the dataset using the `sketching.py` module.
2. Extract a model from the sketch by a compressive learning method contained in the `compressive_learning.py` module.

$$X \longrightarrow \boldsymbol{z}_X \longrightarrow \Theta$$

Figure 2: Flowchart of a typical compressive learning execution with Pycle.

3 A tutorial tour of `pycle`

We now explain the core features of `pycle`. I'll go over each of the submodules one by one and introduce, in a "tutorial style", the most important tools they provide. Our focus being on understanding, this section is neither concise nor exhaustive (however, section 4 is a methodical list of all the toolbox features).

3.1 Sketching

To use the `sketching` submodule, you first need to import it (to spare me some typing, I personally like tu use `sk` as shorthand). Usual sketching as defined in (1) can then be done by calling `sk.computeSketch` as follows.

```
import pycle.sketching as sk

X = ...   # load a numpy array of dimension (n,d)
Phi = ... # sketch feature map, see later

z = sk.computeSketch(X,Phi)
```

As you might have guessed, `sk.computeSketch(dataset,featureMap)` requires two arguments: the dataset X, and the feature map Φ. Let's start with the simplest one: the dataset X should be given as a 2d numpy array of dimensions (n, d).

Note: Matrix representation conventions. We follow the same convention as most mainstream Python ML modules: the dataset, that we mathematically describe as $X = (\boldsymbol{x}_i \in \mathbb{R}^d)_{i=1}^n \in \mathbb{R}^{d\times n}$, is represented by a numpy array of shape (n, d). In particular, `X[0]` references $\boldsymbol{x_1}$: note the awkward inversion of dimension order. For matrices that don't represent datasets (e.g., Ω in the examples below), we stick to the mathematical convention instead, i.e., a matrix of the type $\mathbb{R}^{a\times b}$ is represented by a numpy array of shape (a, b).

The feature map argument can be specified in one of the two following ways. Either—and this is the method I recommend—you give an instance of a `FeatureMap` object (explained below), or you directly provide a callable Python function (e.g., to compute the second-order moments for the sketch, write `Phi = lambda x: x**2`). Note that the second method is proposed for research purposes, in the case you want to construct a custom feature map that cannot be instantiated with the methods provided within the `FeatureMap` class.

Note: FeatureMap objects. Why do we use `FeatureMap` objects instead of (Python) functions to represent... well, (mathematical) functions? Because we often require additional metadata/methods about Φ (e.g., target dimension m, jacobian $\nabla\Phi$,...). All these parameters and methods are conveniently packaged inside `FeatureMap` objects.

Voilà, you know the basics of how the `sketching` submodule is used! Well, OK, all we saw was a function that computes an average, but hey, that's like, the essence of sketching, it's not my fault. Luckily for you, `sketching` has much more to offer. First and foremost, I'll demonstrate the whole zoo of feature maps that are readily available—I promise you, you'll construct your `FeatureMap` object Φ in no more than two lines of code. After this comes more advanced methods of the toolbox that you may not need right away: I'll show how to automatically select the *scale hyper-parameter* in the aforementioned feature maps (which in practice can be hard to guess *a priori*); I'll then finally explain `pycle`s functions for sketching with Differential Privacy guarantees.

3.1.1 Painless instantiation of standard feature maps: the `SimpleFeatureMap` class

All feature maps used in CL up to now are of the following form, which we call "Simple Feature Map",

$$\Phi(\boldsymbol{x}) = f(\Omega^T\boldsymbol{x} + \boldsymbol{\xi}), \quad \text{where} \quad \Omega = [\boldsymbol{\omega}_1, \cdots, \boldsymbol{\omega}_m] \in \mathbb{R}^{d\times m},\ \boldsymbol{\xi} = [\xi_1, \cdots, \xi_m]^T \in \mathbb{R}^m, \tag{2}$$

and f is a point-wise nonlinearity (i.e., $\Phi_j(\boldsymbol{x}) = f(\boldsymbol{\omega}_j^T\boldsymbol{x} + \xi_j)$ for all j). In general, you can instantiate such a nonlinearity in pycle in the following way:

```
import pycle.sketching as sk

f = ...      # nonlinearity (Python function, tuple or string)
Omega = ...  # (d,m) numpy array
xi = ...     # (m,) numpy array
```

```
Phi = sk.SimpleFeatureMap(f, Omega, xi)
```

Moreover, the "usual" arguments for this simple feature map can be easily called, as we explain below.

- nonlinearity
- projection
- dithering

3.1.2 Designing the sampling pattern in the feature maps: the `estimateSigma` function

3.1.3 Guaranteeing the Differential Privacy of sketching: the `computeSketch_DP` function

3.2 Learning

3.3 Utilities

4 Documentation

4.1 Sketching methods

4.2 Learning tools

4.3 Utilities

5 Conclusion: going further

My primary goals for `pycle` are that it should be:

- **intuitive to use:** practitioners with no background knowledge in compressive learning and little experience in Python should be able to use it to implement compressive learning in their own projects;
- **flexible to new features:** researchers with interest in compressive learning (that want to try out new methods/techniques in CL) should be able to easily extend this code to suit their own needs, without having to re-write things from scratch (and eventually, suggesting to add some features to the toolbox);
- **efficient to run:** the main motivation of compressive learning is based on the fact that it can be much more memory- and time-efficient than traditional learning methods, so the performances of the toolbox should fulfill that promise (note: this goal is still a challenge for me, this item is rather wishful thinking).

With this in mind, if you have any suggestions to improve the toolbox, please don't hesitate to contact me!

References

[1] R. Gribonval, G. Blanchard, N. Keriven, and Y. Traonmilin, "Compressive statistical learning with random feature moments," *arXiv preprint arXiv:1706.07180*, 2017.