An introductory guide to `pycle`
(the **Py**thon **c**ompresive **le**arning toolbox)

Vincent Schellekens

December 18, 2019

Abstract

This is a beginner-friendly guide to `pycle`: a Python toolbox to perform compressive/sketched learning. It first shortly explains what "compressive learning" means, and how this toolbox is organized. A detailed tutorial then teaches you how you should use (or, who knows, extend?) its main features. If you'd rather dive in right away, I also wrote some practical examples available as jupyter notebooks on `pycle`s main github page (`https://github.com/schellekensv/pycle`).

1 Introduction

1.1 What is compressive learning?

In usual machine learning, we fit some parameters θ (e.g., a parametric curve in regression, centroids in k-means clustering, weights in a neural network...) to a given set of training data X. The actual algorithms for such tasks usually necessitate to access this training set multiple times (one complete pass on the dataset is sometimes called an "epoch"). This can be cumbersome when the dataset is extremely large, and/or distributed across different machines. Compressive learning (CL, also called sketched learning) seeks to circumvent this issue by compressing the whole dataset into one lightweight "sketch" vector, that requires only one pass on the dataset and can be computed in parallel. Learning is then done using only this sketch instead of the inconveniently large dataset (that can be discarded). This allows to significantly reduce the computational resources that are needed to handle massive collections of data.

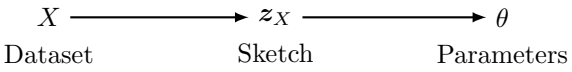

Figure 1: The compressive learning workflow.

More precisely, CL comprises two steps (Figure 1):

1. **Sketching:** The dataset $X = \{\boldsymbol{x}_i \,|\, i = 1, ..., n\}$ (where we assume $\boldsymbol{x}_i \in \mathbb{R}^d$) is compressed as a sketch vector that we note $\boldsymbol{z}_X$, defined as the average over the dataset of some features $\Phi(\boldsymbol{x}_i)$ (the function $\Phi : \mathbb{R}^d \mapsto \mathbb{C}^m$ or $\mathbb{R}^m$ computes m features, possibly complex):

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

   Since this is a simple averaging, sketching can be done on different chunks of X independently, which is quite handy in distributed or streaming applications.

2. **Learning:** The target model parameters θ are then obtained by some algorithm Δ that operates *only* on this sketch,

$$\theta = \Delta(\boldsymbol{z}_X). \tag{2}$$

   Typically, this involves solving some optimization problem $\min_\theta f(\theta; \boldsymbol{z}_X)$.

In the following, these steps are explained intuitively, the formal details being introduced only when needed; for a more solid/exhaustive overview of compressive learning, see [1].

1.2 Requirements

The `pycle` package is built on standard Python scientific computing libraries: `numpy`, `scipy` and `matplotlib`; if you don't already have them installed, follow the instructions at `https://www.scipy.org/install.html`.

1.3 Toolbox organization

The `pycle`toolbox is a collection of several submodules:

1. The `sketching.py` module instantiates feature maps then computes the sketch of datasets with it.
2. The `compressive_learning.py` module contains the actual "learning" methods, extracting the desired parameters from the sketch.
3. The `utils.py` module contains miscellaneous auxiliary functions, amongst others for generating synthetic datasets and evaluate the obtained solutions (as well quantitatively with well-defined metrics as qualitatively with visualization tools).
4. Finally, the `third_party.py` module serves to group code chunks used by `pycle`that are written by other developers but not published as independent packages.

2 A tutorial tour of `pycle`

Let's explore the core submodules of `pycle`! Our focus here is understanding, so this section is a high-level tutorial rather than an exhaustive enumeration of what's inside the toolbox.

2.1 Sketching dataset with the `sketching.py` submodule

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) is simply done by calling `sk.computeSketch`. In practice it typically looks like this (we will fill in the dots later):

```
import pycle.sketching as sk # import the sketching submodule

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

z = sk.computeSketch(X,Phi) # z is a numpy array containing the sketch
```

As you might have guessed, `sk.computeSketch` requires two arguments: the dataset and the feature map. Let's start with the simplest one: the dataset X should be given as a 2-d numpy array of dimensions (n, d), i.e., number of examples times their dimension[1].

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[2].

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.

[1] **Matrix representation conventions:** We follow the same convention as most mainstream Python machine learning modules: the dataset, mathematically described 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).

[2] **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.

2.1.1 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",

$$\Phi(\boldsymbol{x}) = f(\Omega^T \boldsymbol{x} + \boldsymbol{\xi}), \quad \text{where} \quad \Omega = [\boldsymbol{\omega}_1, \cdots, \boldsymbol{\omega}_m] \in \mathbb{R}^{d \times m}, \ \boldsymbol{\xi} = [\xi_1, \cdots, \xi_m]^T \in \mathbb{R}^m, \tag{3}$$

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). One way to interpret this map is to associate it with a one-layer neural network (where we don't necessarily learn the "weights" Ω). 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:** you can simply pass in any Python function $\mathbb{R} \mapsto \mathbb{C}$ or $\mathbb{R}$. However, computing the gradient[3] $\nabla\Phi$ requires $\frac{\mathrm{d}f(t)}{\mathrm{d}t}$; you can also pass in a tuple of functions $(f, \frac{\mathrm{d}f(t)}{\mathrm{d}t})$ so that you can use gradient-based CL methods later. Lastly, you can use one of the predefined nonlinearities by passing a string matching one of the implemented nonlinearities. Those are : ...
- **Projection:**
- dithering

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

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

2.2 Learning from the sketch with the `compressive_learning.py` submodule

2.3 Utilities

3 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!

[3] Noting the Jacobian matrix $\nabla\Phi(\boldsymbol{x}) = [\nabla\Phi_1(\boldsymbol{x}), \cdots, \nabla\Phi_m(\boldsymbol{x})] \in \mathbb{R}^{d \times m}$ and $f'(t) = \frac{\mathrm{d}f(t)}{\mathrm{d}t}$ applied component-wise, we have $\nabla\Phi(\boldsymbol{x}) = \mathrm{diag}\left(f'(\Omega^T \boldsymbol{x} + \boldsymbol{\xi})\right) \cdot \Omega$.

References

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