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

Vincent Schellekens

May 26, 2020

Abstract

This guide presents `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 how to use its main features. To dive in right away, some complementary practical examples are available as jupyter notebooks on the main github page of `pycle`: `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.

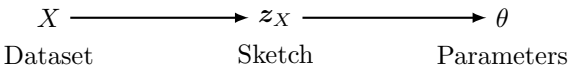

Figure 1: The compressive learning workflow.

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 datasets with the `sketching.py` submodule

To use the `sketching` submodule, you first need to import it; I often use `sk` as shorthand. Sketching, as defined in (1), is done by simply calling `sk.computeSketch`, as follows (we'll 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 shown in the code snippet, `sk.computeSketch` requires two arguments: the dataset X (given as a numpy array of dimensions (n, d)) and the feature map Φ, which must be an instance of a `sk.FeatureMap` object[1]. Actually, all feature maps used up to now in CL are of the following "Simple Feature Map" form:

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

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)[2]. You can easily instantiate such a nonlinearity in `pycle` using the a `SimpleFeatureMap` child class:

```
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)
```

We now explain how to set those three arguments.

- **Nonlinearity `f`:** you can simply pass as a string the name of standard nonlinearities used in CL, such as:

[1] Why do we use `FeatureMap` objects instead of (Python) functions to represent... well, (mathematical) functions? The reason is that we often require additional information about Φ (such as the ambient dimension d, the target dimension m, a method to compute the jacobian $\nabla\Phi$,...). All these parameters and methods are thus conveniently packaged inside the `FeatureMap` objects.

[2] One way to interpret this map is to associate it with a one-layer neural network (without learning the "weights" Ω).

  - `"complexExponential"`, for the complex exponential $f(\cdot) = \exp(\mathrm{i}\cdot)$ (corresponds to the random Fourier features sketch);
  - `"cosine"`, simply the cosine $f(\cdot) = \cos(\cdot)$ (the real part of the random Fourier features sketch);
  - `"universalQuantization"`, for the one-bit square wave with normalization in $\{\pm 1\}$, given by $f(\cdot) = \mathrm{sign} \circ \cos(\cdot)$ and used for quantized sketching [2]. The "complex equivalent" is also available, under the name of `"universalQuantization_complex"`, which corresponds to $f(\cdot) = \mathrm{sign} \circ \cos(\cdot) + \mathrm{i} \cdot \mathrm{sign} \circ \sin(\cdot)$.

  Alternatively, you can pass in any Python function $\mathbb{R} \mapsto \mathbb{C}$ or $\mathbb{R}$. However, since computing the gradient[3] $\nabla\Phi$ requires $\frac{\mathrm{d}f(t)}{\mathrm{d}t}$, you can also pass in a *tuple* of functions $(f, \frac{\mathrm{d}f(t)}{\mathrm{d}t})$ in order to use gradient-based CL methods later.

- **Projections/frequencies `Omega`:** A (d, m) numpy array, typically randomly generated. Without entering into the details, common choices are instantiated by `sk.drawFrequencies(drawType,d,m,Sigma)`, where `drawType` is a string describing the sampling pattern (`Gaussian`, and `FoldedGaussian` or `AdaptedRadius` which perform better in higher dimensions, see [3]) and `Sigma` is the associated scale parameter (in the simplest case, it is a scalar parameter corresponding to the bandwidth σ^2 of the associated Gaussian kernel).

- **Dither `xi`:** this optional parameter expects an $(m,)$ numpy array (not providing anything is equivalent to setting $\boldsymbol{\xi} = \mathbf{0}$). To draw *i.i.d.* values uniformly from $[0, 2\pi]$, you can use `sk.drawDithering(m)`.

To summarize, here is a typical example of the creation of a sketch:

```
import pycle.sketching as sk

# Load the dataset
X = ...
(n,d) = X.shape

# Instantiate the feature map
m = 10*d # Sketch size
Omega = sk.drawFrequencies("FoldedGaussian",d,m,Sigma = 0.01) # Kernel bandwidth = 0.1
Phi = sk.SimpleFeatureMap("complexExponential", Omega) # No dithering used here

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

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

For now, `pycle` features mainly one algorithm called `CLOMPR`. It has been proposed to estimate mixtures of Gaussians [3] and to perform k-means clustering [4]. In `pycle`, it is called as follows:

```
import pycle.compressive_learning as cl

# Beforehand, define the feature map Phi and compute the sketch z

# Bounds for the data
bounds = np.array([X.min(axis=0),X.max(axis=0)])

# Define the task
task_name = ... # See below, either "k-means", "GMM" or "GMM-nondiag"
K = ... # Number of centroids/gaussian modes

# Learn from the sketch
theta = cl.CLOMPR(task_name,z,Phi,K,bounds)

# Do something with the model (prediction, visualization, ...)
```

[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 $\nabla\Phi(\boldsymbol{x}) = \mathrm{diag}\left(f'(\Omega^T\boldsymbol{x} + \boldsymbol{\xi})\right) \cdot \Omega$.

The first argument `task_name` defines the task to solve, which conditions the format of the output `theta`. Currently, the available options are

- `task_name = "k-means"`, used to estimate the centroids of k-means clusters. In this case, `theta = (alpha,C)` is a tuple, where `alpha` contains the weights of the clusters and `C` its centroids (in a (K, d) numpy array).
- `task_name = "GMM"` or `"GMM-nondiag"`, used to fit a mixture of Gaussians to the data. Here, `theta = (w,mus,covs)` is a tuple of three elements, where `w` are the mixture coefficients, and `mus` and `covs` are the centers (resp. covariance matrices) of the Gaussian modes, in a (K, d) numpy array (resp. (K, d, d) numpy array). The difference between the two options is that `"GMM"` assumes diagonal covariance matrices, while `"GMM-nondiag"` improves the fit by considering general covariance matrices, during a post-processing fine-tuning step.

Detailed examples of those tasks are available on the main github page of the `pycle` toolbox. With that and the documentation[4] of the different functions, you should have a good idea of how to use the toolbox in practice. In the next section, some advanced functionalities of the toolbox are described.

3 Advanced features of `pycle`

3.1 Helpful tools from the `utils.py` submodule

Dataset generation tools

Several methods allow to generate synthetic datasets, the most notable being `generatedataset_GMM` for datasets sampled from Gaussian mixture models (with a large set of tunable parameters). Moreover, other toy examples datasets can be generated with `generateCirclesDataset`, `generateSpiralDataset` and `generatedataset_Ksparse`.

Performance metrics

The toolbox provides several metrics to assess the quality of the learned models. For k-means, `SSE` computes the Sum of Squared Errors of a set of centroids. For Gaussian Mixture Models, `loglikelihood_GMM` assesses the log-likelihood of the provided Gaussian mixture on a dataset. Moreover, if a ground-truth GMM is known, `symmKLdivergence_GMM` estimates the (symmetrized) Kullback-Leibler divergence between two GMMs [1].

Visualization tools

To visualize the quality of fit of a GMM to the dataset, `plotGMM` plots the contour curves of the given GMMs density, along with the optional dataset.

3.2 Designing the sampling pattern parameters when drawing the feature map

In `sketching.py`, a strategy to estimate `Sigma` (used for example in `sk.drawFrequencies`) from a small preliminary sketch, described in [1], is implemented under the name `sk.estimateSigma`.

3.3 Sketching with Differential Privacy

As described in [5], a layer of differential privacy can easily be incorporated on top of the sketch. In `pycle`, this is achieved by replacing `computeSketch` with its variant `computeSketch_DP`.

4 Going further with `pycle`

The primary goals for the `pycle` toolbox are that it should be:

- **intuitive to use:** practitioners with minimal 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);

[4] Just type `help(nameOfAFunction) if you want to see all the available options.`

- **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 (personal note: this goal is still a challenging for me, this item is rather wishful thinking).

With this in mind, any bug reports, improvement suggestions, ideas and general comments are more than welcome: don't hesitate to contact me (`vincent.schellekens@uclouvain.be`)! If you use the toolbox in your research, please kindly cite it using the DOI `10.5281/zenodo.3855114`; get BibTeX export files (or others) here: `https://doi.org/10.5281/zenodo.3855114`.

References

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

[2] V. Schellekens and L. Jacques, "Quantized compressive k-means," *IEEE Signal Processing Letters*, vol. 25, no. 8, pp. 1211–1215, 2018.

[3] N. Keriven, A. Bourrier, R. Gribonval, and P. Pérez, "Sketching for large-scale learning of mixture models," *Information and Inference: A Journal of the IMA*, vol. 7, no. 3, pp. 447–508, 2018.

[4] N. Keriven, N. Tremblay, Y. Traonmilin, and R. Gribonval, "Compressive k-means," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6369–6373, IEEE, 2017.

[5] V. Schellekens, A. Chatalic, F. Houssiau, Y.-A. de Montjoye, L. Jacques, and R. Gribonval, "Differentially private compressive k-means," in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 7933–7937, IEEE, 2019.