Programing/ADA project

Sequential Learning of Active Subspaces

Denise Nisell Alexandre Senges

https://github.com/sakex/seq-learning

I. Abstract

Active Subspaces Methods (ASM) are ways to reduce the search space during the optimisation of a function by identifying the geometrical subspaces where it varies the most. While these methods are gaining more and more traction for complex optimization problems, there are still areas where they are difficult to apply due to the need of evaluating gradients. For instance, stochastic simulations (like Markov Chains) are difficult to model using ASMs because it is often impossible to differentiate a function whose output might vary randomly. The paper "Sequential Learning of Active Subspaces" (Nathan Wycoff, Mickaël Binois, Stefan M. Wild; 2019) (hereinafter the "Paper") offers a solution to evaluate ASMs without having to know the gradients by using Gaussian Processes. They offer a closed-form solution that can easily be implemented on top of a Gaussian Process regression.

Their paper was still unpublished at the time we started working on it, they also provided a source code that implemented their formulas written in R as well as C++. The goal of our project is to analyze the original paper's code and port it to Python.

II. Introduction

Our first idea was to work on a project that Alexandre has been developing for quite some time. His project involves training neural networks containing Gated Recurrent Units with genetic algorithms. Therefore, we contacted Antoine to know what he thought about it. He alternatively suggested we work on a new project pertaining to ASMs. That is how we started working on this surprisingly complicated subject.

After having signed a confidentiality agreement with him, Antoine gave us the paper and the code (R and C++). On our first read, we almost understood nothing. After spending much time on it, we still don't understand most of it besides the first-order principles. That is because the subject is quite advanced mathematically, and we had no previous knowledge of the mathematical concepts the paper depends on. Besides, it is likely that it would take most engineers uninitiated to the arts of Active Subspaces and Gaussian Processes a few reads before they would understand the dense formulas and the massive derivations of the paper.

Even if the understanding of all maths involved in those formulas is not requested to translate the project to Python, a thorough understanding of the original code is however very important. That is why we spent a significant amount of time installing the R library, as well as reading the code and playing with it to understand how it worked.

We soon noticed that the R part's main functionality was to call the library hetGP which provides the function mleHomGP, it does a maximum likelihood regression to optimise length scales of Gaussian Processes. Then it uses the model returned from this function and passes it to their own library to execute different tasks, such as identifying the matrix C described in the paper.

Denise is better at Python than C++ while Alexandre really likes C++. That is why we divided the work so Denise would write the python unit tests using the C++ library written by Alexandre. We set up a GitHub private repository, GitHub actions for automated testing, a Dockerfile to easily export the environment and we used GitHub projects as a replacement to Jira to manage our different tasks. The idea was that Denise would work on the front end of the library while Alexandre would implement the backend as well as the DevOps.

In section III, we are going to give a basic understanding of GPs, ASMs, and how the paper achieves to merge these concepts together. In section IV, we will discuss the methodology and the different technologies we used to develop our library. Section V is about the programming technicalities that we used to implement the project, we will also show the multithreaded for_each algorithm we developed. Section VI will be an explanation of our development infrastructure. Finally, in section VII, we will show our results and in section VIII, we will provide some insights and ideas to go further with the project.

III. Concepts

In this section we informally describe Active Subspaces (AS), Gaussian Processes (GP) and how we can use GPs to estimate a functions' AS using the paper's technique.

In the field of machine learning, it is common to want optimise highly dimensional functions (functions with a lot of inputs.) However, as the dimensionality increases, it becomes more and more difficult to solve optimisation problems because the search space becomes exponentially larger, and it would require a huge number of data to be able to showcase all the possible combinations for a machine-learning algorithm to learn.

That is why a few methods to reduce the dimensionality of a function are being researched and developed. A popular technique, the Principal Component Analysis (PCA), reduces the dimensionality of a function by identifying which input variables are highly correlated together and it compresses them into fewer variables called the principal components.

Active Subspaces Methods (ASM), provide a different idea. Instead of reducing the number of input variables, they find what are the directions in which a function varies the most. After having found the subsets in which the function is the most sensible to input variations, we know that our optimum will be in one of those subsets, the search space is thus greatly reduced, and it becomes much simpler to optimise our function with a reduced range of possibilities. The rigorous definition of the active subspace requires to compute the

matrix C, defined as the expected outer product of the gradients of the function f per the following formulas shown in the paper:

$$\mathbf{C} = \mathbb{E}\left[(\nabla f)(\nabla f)^\top\right]$$

$$\mathbf{C} = \int_{\mathcal{X}} (\nabla f)(\nabla f)^\top d\mu.$$

The traditional method to estimate C is to use a Monte Carlo algorithm to calculate gradients at random points and approximate the integral (from the paper):

$$\hat{\mathbf{C}} = \frac{1}{M}\sum_{i=1}^{M}(\nabla f(\mathbf{x}_i))(\nabla f(\mathbf{x}_i))^\top$$

Now, it is oftentimes impossible to compute the gradients of a function, for instance in the case of stochastic simulations. It can also be too expensive to compute a Monte Carlo algorithm on very complex or very high dimensional functions.

In “Sequential Learning of Active Subspaces” (2019), the authors propose to use GPs and their properties as an alternative to the Monte Carlo algorithm. GPs are often described as a distribution over possible functions. Informally, GPs are a non-parametric Bayesian inference method that observes a design X and its response Y = f(X), the GP gives us a range of functions f that might be the data generating process with a certain probability, thus the distribution over functions. GPs can be used to extrapolate which function was likely used to convert our design X to its response Y.

A GP is defined by a mean and a covariance function called kernel. The mean function is usually assumed to be constant while the kernel gives us the covariance between each point pair $k(x, x')$. In order to be valid, the resulting kernel matrix $\Sigma = k(X, X)$ has to be positive definite. Common kernels include the Radial Basis Function (RBF), the constant kernel, and variations of the matern Kernel. Kernels typically have a length scale parameter that can be optimised through maximum likelihood regression.

RBF kernel:

$$k(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j)^2}{2l^2}\right)$$

Kernels have the interesting property that they can be combined together by addition or multiplication.

Coming back to ASMs, if a function is unknown and we only have some random inputs and their response, it is not possible to reliably estimate the gradients of the function. However, using GPs, we can approximate what function is our dataset generated with. We can then use this function to estimate our matrix $\hat{C}$. Now, instead of using a Monte Carlo algorithm to estimate $\hat{C}$, the paper provides a way to exploit the properties of Gaussian Processes to use a closed-form formula to find active subspaces.

The proof and derivation of this formula are way too long and complicated to be discussed here. Fortunately, we only have to implement the resulting equation for which we already have a sample code.

IV. Methodology

The goal of our work was to convert the library written for R to Python. This means that our first step was to compile, install and run the R bundle. Doing so was not straightforward because R and C++ both lack a good dependency management system. In R, linking C++ is libraries is done by using the package “devtools”. The package itself isn’t installed by default, and you will need to run `install.packages("devtools")` **inside the R REPL** to install it. Devtools also needs dependencies that you will need to install through apt (on Linux.) There is very little documentation on the subject online, so the best way to know what to install is to look at the console output and pray it will be enough. For us, running `sudo apt install libxml2-dev libcurl4-openssl-dev` did the trick. After that, we had to install the dependencies for the C++ code and the R library, which meant running `install.packages("RcppArmadillo")` RcppArmadillo requires gfortran which was not installed on my fresh g++10.1 so I rolled back to g++7 with update-alternatives. Furthermore, we had to run `install.packages("lhs")`, `install.packages("hetGP")` and `install.packages("numDeriv")`. Finally, we could run `R CMD INSTALL activegp_1.0.3.tar.gz` to install our package from the source. If R had had a good dependency manager like Yarn or Cargo or even a bad one like PyPi, we would not have gone through so much hassle.

The code finally running, we could dive into it and gain an understanding of what it does. The best place to start is to look at the unit tests. The tests are located in the folder “tests/testthat”. The most relevant test is test_C_est.R because it shows how to estimate the matrix $\hat{C}$. What they do there is that they first generate a design made of random data points of two-dimension X, then they pass these points in a function f(x1, x2) = sin(x1 + x2) to generate the response Y. After that, they use the package “hetGP” to run a Maximum Likelihood regression to estimate the GPs length scales on the data set. The module returns an object of type “model”, with member variables such as the inverse covariance matrix of the kernel they use (K^{-1}) and the length scale parameters that they call theta here. They then pass the model to their own function “C_GP(model)”. This function takes the model as input and returns an estimation of the matrix $\hat{C}$. To test the validity of their code, they compare the output matrix to the true C matrix using the norm of the difference. We know the true value because the function f(x1, x2) = sin(x1 + x2) can trivially be differentiated analytically. They do this test with three different kernels:

- The Radial basis function (or Gaussian) kernel
- The matern 3/2 kernel
- The matern 5/2 kernel

What we can deduce from this test, is that in order to replicate their code in Python and test that it works, we should be able to generate the same dataset in Python, do a GP regression somehow and pass the K^{-1}, as well as the length scales to our C++ code. It should return approximately the same matrix.

V. Programming

There were a few challenges we needed to overcome in our project:

1. Generate the same data as in the original code
2. Be able to run a GP regression in Python like the one they do in R
3. Extract the K^{-1} and length scale parameters
4. Write the C++ API such that it would be easy to use
5. Write Python foreign function interfaces (FFI.)
6. Test and compare our results
7. Optimise the C++ code

Generating the data was pretty straightforward, we used numpy.random.uniform, then we passed the resulting sample to the same formula $y : (x) \rightarrow sin(sum(x))$.

The GP regression which we thought would be trivial in the beginning ended up being the most complicated part of the project because of the way mlehomGP works makes it almost impossible to replicate without rewriting the whole GP regression algorithm. We spent hours reading the actual implementation of the function and reading implementations of its Python counterparts. We investigated many different libraries and we provide examples for those libraries:

1. Scikit learn
2. GPy
3. PymC3

The biggest issue we had was to translate the maths from hetGP to Python because hetGP decided to implement their kernels in curious ways. If you look at their formula for the Gaussian/RBF kernel in their official documentation: cov_gen function, you can see that they use the formula $c(x,y) = exp(-(x-y)^2/theta)$ whereas the formula in most implementations is:

$k(x_i, x_j) = \exp\left(-\frac{d(x_i, x_j)^2}{2l^2}\right)$, l and theta meaning the same thing. Libraries like GPy or SKlearn, typically implement the latest formula, it is not a problem for us because the Paper already accounts for this formula. This means that they adapt the length scale parameters themselves in their code by doing `theta <- sqrt(model$theta / 2)`. We just chose to not do that.

Furthermore, digging into the code, we can see they use a variable “nu_hat”, which is a constant value they use to scale the kernel by multiplying it by nu_hat. They also have a constant trend value “beta0” which is the Maximum Likelihood estimation of the mean function of the GP.

GPy and SKlearn apply a non-opinionated approach to their kernels which means that we had to exactly replicate the one in mlehomGP to be able to get the same results. This work was cumbersome because the documentation is very sparse and it is a very advanced and niche thing to do. We managed to replicate the kernel by using this approach (with sklearn's kernels):

```
ConstantKernel(1e-8) * RBF(length_scale=[1.0, 1.0])) + WhiteKernel(1)
```

We also have an example for GPy and pymc3. These libraries don't implement a way to compute the inverse covariance matrix we need for our $\hat{C}$ estimation to work. So, we need to compute it ourselves. The problem is that as stated in mlehomGP's doc, their K^{-1} is unscaled by "nu_hat". To match that, we first use the L matrix (cholesky decomposition of K) we can get from sklearn and we decompose it to find K^{-1}. After that, use the estimated sigma (the value of the first constant kernel) and multiply K^{-1} by it to unscale it as they do in mlehomGP's. It was a very difficult and long process to figure this trick out, we spent almost a week on it for only 3 lines of code. We opened an issue on GPy's github and a question on Stackoverflow. In the end no one helped us but we managed through perseverance and luck.

We managed to reproduce the results for the RBF kernel, but we never succeeded for the Matern 3/2 and the Matern 5/2. The reason, is that as stated in the paper and in hetGP's code (https://github.com/cran/hetGP/blob/master/R/Covariance_functions.R, line 168 and 251), they use the tensor product variant of the kernels. We weren't able to find any implementation of those for Python and it would have been extremely time-consuming for us to implement it. So we resorted to copy-pasting the matrix and the length scales to Python to test our code. It would be interesting to find or write the tensor version of the kernels in Python and submit a pull request to the interested libraries.

When we wrote the C++ API, we wanted to make it easy to use, we also didn't want to make it restricted to Python. Therefore, we just made a class called ASGP that implements the relevant functions and then we exposed that class to Python.

Writing Foreign Function Interfaces (FFIs) for Python in C++ can be error prone because it requires to wrap C++ code in C functions and then deal with Python.h with raw pointers and such. It is because C++ compilers resort to name mangling and their ABIs are not stable amongst each other. It is well known that two C++ shared libraries compiled with different compilers might not be able to be linked together because of the difference in ABIs. The only way to link C++ binaries together is to actually have a C++ compiler. The D programming language's FFI page resumes this problem quite well[1]: *"Being 100% compatible with C++ means more or less adding a fully functional C++ compiler front end to D. Anecdotal evidence suggests that writing such is a minimum of a 10 man-year project, essentially making a D compiler with such capability unimplementable."*

C has a very stable and battle tested ABI, therefore it is the de facto language to write FFIs, and it has become common for languages to interop together through C FFIs even though they don't use any actual C.

For our own FFIs, we decided to rely on the Boost library. Boost is one of the most highly regarded and expertly designed C++ library projects in the world. A lot of their

functionalities have ended up in the standard library such as threads, unique and shared pointers, the functional programming library, etc.. It also provides a very convenient API to write Python modules in C++ Boost.Python, that comes with an extension: Boost.Python (Numpy), that allows you to interface with Numpy directly in C++. Writing modules becomes as easy as wrapping some code in a macro and calling a few methods. It is also surprisingly easy to make C++ classes available to Python by using the template `class_<T>`

Because it is a programming assignment, we went out of our way to showcase our knowledge of the language, even if some parts could have been written in an easier way. For instance, we use hash tables over char array buffers to preload branches on the different kernel types, in the free function :

```
inline constexpr unsigned hashCovType(const char *str, unsigned index = 0)
```

Another advanced but unnecessary concept we used is static polymorphism. Instead of using branching in every function call like in the original code, or creating three classes that would from a base class, we chose to create one templated class `template<eCovTypes>class GpImpl : public GPMembers` that would pick the right implementation depending on the `eCovTypes` parameter, it is an enum that represents the kernel in use. The GPMembers base class is just a struct containing the data members for GpImpl. This design pattern is called a trait and allows us to decouple the data from the algorithms.

The static polymorphism design pattern allows us to completely remove branching because we determine branches at compile time, furthermore, not inheriting from a base abstract class removes the need for vtables. These optimisations are critical when it comes to reducing branch miss predictions, cache misses, and allowing the compiler to vectorise our code using SIMD instructions.

To reduce code reuse, we wrote a macro `CHOOSE_IMPL(IMPL)` that calls the right implementation given a function name.

We also noticed that at many places in the original code, the authors would call the function std::sqrt(PI), std::sqrt is not a constexpr function which means that it is possible the compiler doesn't optimise those calls away. That's why we implemented a `constexpr sqrt(long double x)` function using the newton_raphson approach to the square root algorithm. Having a look at the disassembled code of both versions on GCC actually showed no difference, so it might have been over-engineering.

We also tried to reorganize the code to prevent recalculations. Compilers often have a hard time optimising away basic operations. For instance the code:

```
double test(double a, double b) {
    return (a - b) + b - a;
}
```

Compiles on https://godbolt.org/ to (with g++ -O3):

```
test(double, double):
    movapd xmm2, xmm0 // Move a into xmm2 (xmm registers allow SIMD instructions)
    movapd xmm3, xmm0 // Move a into xmm3
```

```
        subsd xmm2, xmm1 // xmm1 contains b, so xmm2 = a - b
        addsd xmm2, xmm1 // xmm2 = a - b + b
        movapd xmm0, xmm2 // Moves xmm2 to xmm1
        subsd xmm0, xmm3  // xmm0 = xmm2 - a
        ret   // Returns to the caller
```

We can see the output still does all the operations, when mathematically we know it always should return 0. Worse, we expose ourselves to overflow. It could have been optimised to:

```
test(double, double):
        pxor xmm0, xmm0  // Easiest way to load 0 into an xmm register
        ret
```

That is why we rewrote the original functions in order to prevent recalculating useless parts.

For instance, the line `-6*sqrt(3)*a*b*t - 9*a*t2 - 9*b*t2` was rewritten as `-six_sqrt_3*a*b*t - (a+b)*9.*t2`

To compare outputs we compiled both versions on godbolt:

```
#include <stdio.h>
#include <iostream>
#include <cmath>


double original(double a, double b, double t) {
        double a2 = a*a, b2 = b*b, t2 = t*t, t3 = t2*t;
        return -6*sqrt(3)*a*b*t - 9*a*t2 - 9*b*t2;
}

double ours(double a, double b, double t) {
        double a2 = a*a, b2 = b*b, t2 = t*t, t3 = t2*t;
        double constexpr sqrt_3 = std::sqrt(3.);
        double constexpr six_sqrt_3 = 6.*sqrt_3;
        return -six_sqrt_3*a*b*t - (a+b)*9.*t2;
}
```

Compiles to (-O3, only relevant code, not showing Local Constants.)

```
original(double, double, double):
        movapd xmm3, xmm0
        movsd xmm0, QWORD PTR .LC0[rip]
        movapd xmm4, xmm2
        mulsd xmm4, xmm2
        mulsd xmm0, xmm3
```

```
        mulsd xmm0, xmm1
        mulsd xmm0, xmm2
        movsd xmm2, QWORD PTR .LC1[rip]
        mulsd xmm3, xmm2
        mulsd xmm1, xmm2
        mulsd xmm3, xmm4
        mulsd xmm1, xmm4
        subsd xmm0, xmm3
        subsd xmm0, xmm1
        ret
ours(double, double, double):
        movsd xmm3, QWORD PTR .LC0[rip]
        mulsd xmm3, xmm0
        addsd xmm0, xmm1
        mulsd xmm0, QWORD PTR .LC1[rip]
        mulsd xmm3, xmm1
        mulsd xmm3, xmm2
        mulsd xmm2, xmm2
        mulsd xmm0, xmm2
        subsd xmm3, xmm0
        movapd xmm0, xmm3
        ret
```

Our version clearly uses less instructions than the original one, we can infer it is a performance gain because it reduces the size of the function (less cache needed) and it does less operations.

To optimise the execution speed, we used our own multithreading algorithm instead of relying on openMP. We did that mostly to show our knowledge of the standard library as well as the fact that openMP has a small instantiation overhead that would be felt on such a small algorithm. The multithreaded algorithm is tailored for our scenario of a loop over a range i=0 to n with a subloop j=i to n with two different functions, one for i==j and one for i!=j. You can see our code at “src/helpers/thread_n_m”. We use a mutex to lock two heap assigned counters i and j that are shared amongst threads. When a thread gets access to the lock, it increments the relevant i and j and calls the right callback. All the threads work on it, even the main thread. Since C++11, we can use lambdas, which makes it more convenient to write functional code. We used lambdas as arguments to our function, but std::function, boost::function or raw C function pointers would also work with our template arguments.

The project was already very complex, so we only implemented the estimation of the $\hat{C}$ matrix and not the update function like in the sample code.

VI. Development pipeline

For our development, we used git for version control and https://github.com to host the repo as well as run automated testing with GitHub actions. Setting the environment was a pain, so we made a Docker image to easily share it, the advantage of this approach is that we could just use it to run our tests on GitHub and we didn't have to download every component manually every time.

If you wanted to maintain the repository, you would only need to pull the repository. Then either build it with cmake, or build a docker image. Then, write your changes. Finally, submit a Pull Request, it would need to pass the tests, then we would have to review it and accept to merge it or not.

VII. Results

Unfortunately, we were only able to reproduce the estimation of the matrix $\hat{C}$ with the Radial Basis Function kernel. The original code didn't attempt to make it work with matern kernels either. However the RBF kernel does pass all the tests with GPy, SKlearn and pymc3. Furthermore, the Matern kernels do pass the tests for eigenvalues.

VII. Conclusion

This project was very interesting and it taught us a lot about Active Subspaces and Gaussian Processes. We spent so much time investigating the different libraries and understanding the maths behind GPs, that we acquired a deep understanding we would not have sought in other circumstances. We were very frustrated at not being able to reproduce the results with the other kernels after spending so much time trying. We think it would be worth investigating, however, it would have been too ambitious for us to try and implement a tensor product matern kernel.

1. https://dlang.org/spec/cpp_interface.html