EE596A Submodular Functions
Spring 2016

University of Washington
Dept. of Electrical Engineering

Final Project. Due Wed/Thur June 8/9, 2016 Electronically

Prof: J. Bilmes <bilmes@ee.washington.edu>
TA: K. Wei <kaiwei@uw.edu>

Monday, May 30 2016

All homework is due electronically via the link https://canvas.uw.edu/courses/1039754/assignments. Note that the due dates and times might be in the evening. Please submit a PDF file. Doing your homework by hand and then converting to a PDF file (by say taking high quality photos using a digital camera and then converting that to a PDF file) is fine, as there are many jpg to pdf converters on the web. Some of the problems below will require that you look at some of the lecture slides at our web page (http://j.ee.washington.edu/~bilmes/classes/ee596b_spring_2016/).

Please post all questions you have to our discussion board (https://canvas.uw.edu/courses/1039754/discussion_topics).

1 Minimum Norm Point Implementation for SFM

For your final project, you are to implement the minimum norm point algorithm for general purpose submodular function minimization (SFM). This is an interesting numerical algorithm that was originally developed for finding the vector with a minimum 2-norm in a convex polytope that can be described by a set of vertices. Normally the algorithm would be slow if it was the case that the set of vertices is large. In the submodular case, the number of vertices could be exponential in $|V| = n$, the ground set size, even for simple submodular functions, but thanks to Edmond's greedy algorithm and the properties of submodularity, this is not an impediment at all.

Before we begin, you should be aware of a number of useful readings on the algorithm.

- Our class lecture on the topic which describes the necessary background properties of submodular functions, describes the algorithm itself in great detail, gives a geometric example, proves certain things about the algorithm (i.e., that it converges to the minimum norm point in a finite number of steps and that it correctly produces the minimum norm point in a polytope), that for the submodular case we can use Edmond's greedy algorithm, and lastly, a proof that the minimum norm point in the base polytope B_f of a submodular function f does indeed give you enough information to very easily compute not just the minimum and maximum size minimizers of the function f but in fact the **entire** lattice of minimizers. Please see lecture 17 at http://www.ee.washington.edu/people/faculty/bilmes/classes/ee596b_spring_2016/lecture17.pdf

- The original Wolfe paper, "Finding the nearest point in a polytope", Mathematical Programming 11 (1976) 128-149. Wolfe is the same person as the well known Wolfe from the Frank-Wolfe algorithm (also called the conditional gradient procedure, for constrained convex optimization). However, the min-norm algorithm is not the same algorithm as the conditional gradient (Frank-Wolfe) procedure.

- Fujishige originally realized the importance of the minimum norm point of the base polytope in the 1980 paper "Lexicographically optimal base of a polymatroid with respect to a weight vector". and then in his 1991 book (the second edition of which came out in 2005), he showed that this point can easily produce a minimizer for an general submodular function.

- The 2009 paper "A Submodular Function Minimization Algorithm Based on the Minimum-Norm Base" by Fujishige and Isotani gave more details of the algorithm, and even gave some timing numbers.
- In 2012, Garcia's masters thesis "High-Performance Submodular Function Minimization" gave a full account of the algorithm, including various implementation tricks to make it still faster.
- In 2013, Hui Lin's Ph.D. thesis also has a chapter on a separate implementation of the min-norm algorithm, and gave some empirical real-world examples of submodular functions that suggested that the min-norm complexity could be as bad as $O(n^6)$ in practice
- A lower bound complexity of the min-norm has not been established. However, in 2014, Chakrabarty, Jain, and Kothari in their NIPS 2014 paper "Provable Submodular Minimization using Wolfe's Algorithm" showed a pseudo-polynomial time bound of $O(n^7 g_f^2)$ where $n = |V|$ is the ground set, and g_f is the maximum gain of a particular function f. Note that this is pseudo-polynomial since it depends on the function values. There currently is no known polynomial time complexity analysis for this algorithm.

Your job as part of the final project is to write C++ code for the minimum norm point algorithm. Here are your responsibilities and due dates (for more specifics on the due dates, see our dropbox (https://canvas.uw.edu/courses/1039754/assignments)).

1. Write the code itself, and abide by the interface and data structures for SFM that we define below.
2. Make sure the code compiles (in C++) and runs correctly. The code needs to return the minimum solution and the maximum solution of the minimizers. That is, let

$$\mathcal{W} = \operatorname*{argmin}_{A \subseteq V} f(A) \tag{1}$$

   be the set of minimizers which we know to be a lattice. Then your code needs to return the minimum minimizer $\cap_{W \in \mathcal{W}} W$ and also the maximum minimizer $\cup_{W \in \mathcal{W}} W$.
3. On Wednesday's class, June 8th, we will time the code live during class. While your code is being timed and tested, you will simultaneously give a brief (5 minute, or so) presentation on the code and your experiences in writing it, and anything interesting you learnt, or encountered when writing the code.
4. You need to **turn in all of the code** compressed as a single `.zip` file. Also you need to **turn in a slide deck** for your 5-minute presentation (must be either `.pptx`, `.pdf`, or `.key` presentation).

   Your code itself should be placed in **one single** C++ file (called `minNorm.cc`) that we can compile and link to, and that contains the algorithm and that obeys the interface that is described below.

```
void minNormPoint(submodularFunctions& f,
                  unordered_set<int>& minSolSet,
                  unordered_set<int>& maxSolSet,
                  std::vector<double>& xstar);
 // f is the input submodular function
 // minSolSet is the minimum cardinality solution returned by MN
 // maxSolSet is the maximum cardinality solution returned by MN
 // xstar is the continuous solution of the minimum norm point problem.
```

   Note that the interface uses some standard C++ objects, such as `unordered_set` data types.

5. On Thursday, June 9th, you need to **turn in a short 2-3 page** (PDF format) writeup on your experiences in writing the code. This should include anything interesting you encountered. Also, what did you do to test that your code was correct? Give a short description of what you learned. Mention if you used an external library, what it is, its name, and what precisely you used it for. Mention as well the source of the library and any URLs, etc. for the libraries home page.

Along with the assignment is an extra `.zip` file called `MinNormPoint.zip` that includes additional details, including code stubs, and this documentation.

What we will do is compile your code, and link to the routine above and call it with a number of submodular functions. You do not get to see the submodular functions we will test with (we will know the correct answers). We will do the timing as well (using timing code similar to that described below), will test if it is correct or not.

The code is correct if it passes the following things:

1. The minimum norm point vector x^* must be correct (to numeric precision). We will calculate the relative difference, i.e., if x^* is your vector, and x^{Kai} is our vector, then we calculate $\|x^* - x^{\text{Kai}}\|/\|x^{\text{Kai}}\|$.

2. The minimum and maximum solutions to SFM are correct (i.e., this is a ensuring that the sets returned are correct).

3. The value of the submodular function evaluated at both of the sets returned.

We will also compute and report the timing of your code for each function we test. All of of this while projecting on our screen. Of course, we'll also test that your code compiles correctly :-) We plan to use the `clang` C++ compiler on MacOS/OSX. Please do not use any fancy features of C++, keep it simple.

Below is some additional documentation on the submodular infrastructure that we provide. All our code is in C++, and so yours should be as well (although it would be possible to code in C and then link to the C++, that probably will end up taking more time than learning the simple syntax of C++ that we're using. Note we will not be using any of the fancy features of C++, just simple base class, class members, and single inheritance).

The code uses the unordered set C++ data structure to represent sets. It implements some simple submodular functions, which can be used as templates to define other more complicated submodular functions. Below, we give some implementation details on this.

We define an abstract base class called *submodularFunctions*. All instances of submodular functions are derived from this class. It has the following members and functions.

- groundSet (which is the ground set defining these functions). This is implemented via as C++0x data-structure called *unordered set*.

  You can expect the ground set to be the set of integers 0 through an integer n-1. The function signature suggests maybe not, but the groundset will indeed be $[n] = \{0, 1, 2, \ldots, n-1\}$, which is consistent with the `totalOrder` class provided. Note that this is not absolutely necessary, as the function signature takes any `unordered_set`, but we will be providing groundsets consisting of $[n] = \{0, 1, 2, \ldots, n-1\}$. This is, moreover, necessary so that `totalOrder` works as a total ordering of $[n]$, since it is not currently a total ordering of arbitrary elements.

- The groundSet size (n).

- Constructors and destructors.

- Evaluation functions:

  1. f.eval(unordered set<int> sset), which evaluates f(S).
  2. f.evalgainsadd(unordered set<int> sset, int item), which evaluates $f(i|S)$.

3. f.evalgainsremove(unordered set<int> sset, int item), which evaluates $f(i|S \setminus i)$.
4. f.getExtremePoint(totalOrder<double> ordering, vector<double>& extremePoint): This provides an extreme point of the submodular polyhedron for a given ordering. You will need to implement this greedy algorithm.

We also define some derived class instances. We provide here three simple examples of submodular functions, which can be used as templates in testing other more sophisticated functions. For the actual evaluation of your code, we might use other more sophisticated submodular functions. Provided here are three functions:

- Modular function: $f(X) = \sum_{i \in X} m_i + c$.
- Concave over cardinality function: $f(X) = \phi(|X|)$.
- Concave over modular function: $f(X) = \phi(m(X))$.

We also provide a helper class `totalOrder`, which implements a total ordering of elements (essentially a permutation). It has sophisticated methods like finding random orderings, finding orderings with respect to as vector x, and so on. You can use this if you want. Using all these features, you are required to write the code for the minimum norm point algorithm. Some simple tests you can do to test if your code is working or not:

- Try out $f(X) = \sqrt{|X|}$. This is a very simple function, but MN does not obviously know its answer. You should verify that you get zero as the solution.
- You can also try $f(X) = m(X)$, where m is a not necessarily positive modular function. The MN algorithm should give as solutions $X = \{j : m(j) \leq 0\}$. In fact, in this case, the minimal solution is $X_{\min} = \{j : m(j) < 0\}$, while the maximal solution is $X_{\max} = \{j : m(j) \leq 0\}$.

2 Linear equation solver

Note that you will need to solve a system of linear equations in C++, or matrix inversion of arbitrary dimension. This is part of computing the min norm point in an affine hull of a set of points. Unless you want to, you don't need to implement your own linear solver, you're free to pull one off of the web. For example, you may use Eigen http://eigen.tuxfamily.org/index.php?title=Main_Page which has a nice interface, but it might not be the fastest implementation available. Instead, you might look at Section 5 of Wolfes 1976 paper that talks about a Cholesky factorization approach, or even method D of that paper, as those methods should be faster and still numerically well behaved.

You might also consider the tolerance parameter suggested by Wolfe (e.g., 10^{-10}).

All of these choices should be **fully documented in your writeup that you turn in.**

3 Properly Timing Code in C++

You will find some timing code (in the file `timing_routines.cc`) that you can use (and that we will use) to time (in fairly high resolution) just the amount of time it takes to run the function. Timing code is a bit tricky to get right, however, so we want to offer a few words on this.

Firstly, when you time some code, it needs to last long enough for the timer to get an accurate account of how long it takes, and this is because the timer clock is usually much slower than the machine clock itself (and if an event lasts shorter than a timer clock, then it will just report essentially zero). The timer code basically works like:

1. t_{start} = start timer via the C/C++ function `getrusage`
2. call a routine you want timed
3. t_{end} = end timer via `getrusage`
4. call a routine with the arguments t_{start} and t_{end} which computes the time.

The problem is that if step 2 runs too fast, the timer will just report zero and you won't get anything back. What to do in this case is to run the routine multiple times, i.e.:

1A. t_{start} = start timer via `getrusage`

2A. Repeat k times: call a routine you want timed, like `minNorm`

3A. t_{end} = end timer via `getrusage`

4A. call a routine with the arguments t_{start} and t_{end}

Where you set k to be large enough to get a good estimate of the amount of time (and we assume that the cost of the loop k times is negligible), and then divide the final reported time by k to get the per-routine time cost. How big to set k depends on how fast the routine is and how accurate your timer is. The `getrusage` code in the żip file usually uses a good timer (in linux, and including on osx machines). However, sometimes it is only in the millisecond range (i.e., a 1ms clock) so you need to make sure that your code runs faster than 1ms. It is probably a good idea to set k to make sure that whatever you are timing runs, at the very least, around 5-10 seconds. Of course, if you're timing a very big submodular function that itself takes minutes to solve, you can set $k = 1$.

Lastly, it is important to accurately time code to make sure that the operating system (OS) itself doesn't distort the timing (this kind of problem happens frequently in machine learning papers, while the high-performance computing community usually does it right). Basically, this means that you run steps 1A-4A themselves in a loop (say M times) and then the time you report is the minimum of the M timing results you get back. The reason for this is that if the OS preempts your code, if anything `getrusage` will overestimate the amount of time it takes rather than under-estimate it. This means that the routine itself is actually run Mk times. How big to set M? Usually $M = 10$, and k = whatever it takes to make sure that the code surrounded by the `getrusage` calls lasts long enough to be accurately timed.