Homework #9, EECS 598-006, W20. Due **Thu. Mar. 27**, by 4:00PM

1. [25] **Learning a binary classifier for handwritten digits using 1-norm sparsity regularization**

A previous HW problem designed a binary classifier for two handwritten digit images using a Fair potential regularizer. For this problem you will instead use a 1-norm that truly encourages the feature weights to be sparse:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \Psi(\boldsymbol{x}), \quad \Psi(\boldsymbol{x}) = \mathbf{1}' h_{.}(\boldsymbol{A}\boldsymbol{x}, \delta_h) + \beta \|\boldsymbol{x}\|_1 ,$$

where again h denotes the **Huber hinge loss** function. Because this cost function is **composite**, *i.e.*, not smooth, we must use a **proximal method** rather than a gradient method. **POGM** is particularly well suited to such applications.

Your starting point should be the template code used in a previous HW problem; use $\delta_h = 0.1$ again.
That code uses the `MIRT` package and in that package is `pogm_restart` that you may use for this problem (or you may write your own POGM algorithm). Type `?pogm_restart` in a jupyter notebook or in REPL to see its documentation.
Hint. POGM uses the `mom=:pogm` option. FPGM uses the "primary sequence" `yk` (called $\boldsymbol{z}_k$ in the notes), whereas POGM uses the secondary sequence `xk`. (The are the same for PGM.) Keep that in mind when defining `fun`.

(a) [5] Add JULIA code to perform the following steps. Submit a screenshot of your added code to gradescope.
Apply POGM (with its default gradient-based adaptive restart) to minimize the cost function shown above.
Choose a value for β (try between 0.1 and 1) that leads to better classification accuracy than seen in the previous HW.

(b) [5] Make an image of the feature weights $\hat{\boldsymbol{x}}$ learned by POGM with this 1-norm cost function.

(c) [5] Make a plot that shows the cost function $\Psi(\boldsymbol{x}_k)$ versus iteration k for POGM.
Optional: compare to FPGM using the `mom=:fpgm` option (or write your own FPGM); you should see that POGM converges faster than FPGM, as predicted by the worst-case convergence bounds.
(For my value of β, about 300 iterations was sufficient.)

(d) [5] Report the classification accuracy of your method,

(e) [5] Report the number of nonzero $\hat{\boldsymbol{x}}$ values for the 1-norm method and the previous Fair potential method. The `count` command is useful here.

(f) [0] Optional. If you know of (or devise) a method that you think is faster than POGM for this application, try it out. If it is faster, then email me a notebook with the results. The first submission of any method that is faster than POGM (in wall time) will earn $120 - 40 \max\big((n - 2.5)^2, 2\big)$ extra credit points on this HW, where $n \in \{1, 2, 3, 4\}$ is the number of students who collaborated together on the submission. (Collaboration is allowed.) For a given approach, the first team to email me earns the extra credit for that approach. (Multiple teams can earn extra credit for different approaches.)

2. [15] **CLS vs POGM for LASSO**

The **LASSO** optimization problem for sparse regression is especially useful in under-determined problems and is given by:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{x}\|_1 .$$

Recall that in an earlier HW problem you implemented an algorithm `lasso_cls` for solving this problem as a **constrained least squares problem** with the **gradient projection** method.

This problem compares `lasso_cls` to **POGM** for solving the LASSO problem.

(a) [10] Write a script that applies both `lasso_cls` and POGM to data generated as follows and produces the plot below:

```
using Random: seed!
M,N = 50,99; seed!(0)
xtrue = randn(N) .* (rand(N) .< 0.3) # sparse
A = randn(M,N); y = A * xtrue + 0.1*randn(M) # data
```

Use $\beta = 15$ and $\boldsymbol{x}_0 = \boldsymbol{0}$ here. This is a small problem, so use `opnorm`.
Submit a screenshot of your code to gradescope.

(b) [5] Plot the cost function versus iteration k for both the CLS approach and the POGM approach for 50 iterations.
You should see that POGM converges much faster.

(c) [0] Optional: also compare to PGM=ISTA and FPGM=FISTA.
(This comparison is optional because I expect POGM will be faster based on results in [1].

3. [35] **Image inpainting using wavelet sparsity regularizer**

Image **inpainting** is similar to matrix completion: we are given $M < N$ samples of an image having N pixels and we want to recover the entire image. This is an under-determined problem so clearly regularization is needed. For matrix completion we usually assume the matrix is low rank, but that is a poor model for natural images. In this problem we assume that the **discrete wavelet transform** (**DWT**) of the image is sparse, so we estimate the image using the following optimization problem:

$$\hat{\boldsymbol{x}} = \arg\min_{\boldsymbol{x}} \frac{1}{2} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{y}\|_2^2 + \beta \|\boldsymbol{W}\boldsymbol{x}\|_1,$$

where $\boldsymbol{W}$ denotes a $N \times N$ unitary matrix corresponding to an **orthogonal DWT** and $\boldsymbol{A}$ is a diagonal matrix where the diagonal elements are 1 and 0 indicating the sample locations. Keep in mind that $\boldsymbol{x} = \text{vec}(\boldsymbol{X})$ here where $\boldsymbol{X}$ is the 2D image. Because this course is about optimization, not wavelets, the matrix $\boldsymbol{W}$ is provided in `MIRT.jl` as a `LinearMapAA` object by the `Aodwt` function that uses the `DWT` package. Type `?Aodwt` to see its documentation.

(a) [0] Determine the **proximal operator** for the regularization term $g(\boldsymbol{x}) = \beta \|\boldsymbol{W}\boldsymbol{x}\|_1$.

(b) [10] Write a script that applies 20 iterations of POGM to compute $\hat{\boldsymbol{x}}$ above for $\beta = 0.5$, from data generated as follows. Use $\boldsymbol{x}_0 = \boldsymbol{y}$. Your script should produce the results for the remaining parts. This is a large-scale problem so you must not use `opnorm` or any relatives of it. (Hint: you do not need it because of special properties of $\boldsymbol{A}$ and $\boldsymbol{W}$ here.)
Submit a screenshot of your code to gradescope.

```
using Random: seed!
using Plots; default(markerstrokecolor=:auto)
using MIRT: Aodwt, pogm_restart, jim, ellipse_im
nx,ny = 192,256
Xtrue = ellipse_im(ny, oversample=2)[Int((ny-nx)/2+1):Int(ny-(ny-nx)/2),:]
samp = rand(nx,ny) .< 0.3 # sampling pattern
Ytrue = Xtrue .* samp; seed!(0); sig=0.1; Y = Ytrue + sig * randn(size(Ytrue))
W,scales,mfun = Aodwt((nx,ny)) # orth. discrete wavelet transform (LinearMapAA)
plot(jim(Xtrue, "Xtrue"), jim(Y, "Y"),
    jim(mfun(W,Xtrue), "W*x"), jim(scales, "scales"))
```

(c) [5] Plot the cost function $\Psi(\boldsymbol{x}_k)$ versus iteration k to verify that 20 iterations of POGM suffices.
(I had hoped to have you compare `lasso_cls` to **POGM** for this application, but `lasso_cls` currently does not seem to work with `LinearMapAA` objects. Anyway, you should expect by now that POGM will be much faster.)

(d) [5] Show the estimated image $\hat{\boldsymbol{X}}$ and report its NRMSE.

(e) [5] Examine (for yourself) the wavelet coefficients of the true image $\boldsymbol{X}$ by typing:
`plot(jim(mfun(W,Xtrue), "W*x"), jim(scales, "scales"))`
You will see that the **detail coefficients** for `scales` $\in \{1, 2, 3\}$ are sparse, but the **approximation coefficients**, where `scales .== 0`, are not sparse. Thus, it is better to replace the 1-norm above with a weighted 1-norm that uses 0 weight for the approximation coefficients. Mathematically, we want $R(\boldsymbol{x}) = \|\boldsymbol{D}\boldsymbol{W}\boldsymbol{x}\|_1$ where $\boldsymbol{D}$ is a diagonal matrix with 0 elements along the diagonal in locations corresponding to the approximation coefficients. In JULIA, instead of using the regularizer `norm(W*x,1)` a more appropriate regularizer is `norm(d[:]  .* (W*x), 1)` where `d = scales .> 0)`, because that regularizer shrinks only the detail coefficients and leaves the approximation coefficients unchanged.
Extend your script to also run POGM for this weighted 1-norm regularizer and make the remaining plots.
Submit a screenshot of your code extensions to gradescope.

(f) [5] Make a figure of the inpainted image $\hat{\boldsymbol{X}}$ for both the standard 1-norm and the weighted 1-norm, and report the NRMSE values for both estimates.

(g) [5] You should see that the weighted 1-norm gives lower NRMSE, but you might wonder if the same regularization parameter β is suitable for both regularizers. Perhaps some other value of β will improve the results?
Compute the NRMSE for both the 1-norm case and the weighted 1-norm case for these 11 values of β:
`2 .^ LinRange(-4,0,11)`
Plot the two NRMSE vs β on the same plot.
What do you conclude about choice of β here and the pros and cons of 1-norm vs weighted 1-norm regularization?

Non-graded problem(s)

4. [0] Challenge. Find a **tight** worst-case convergence bound for the **PGM**.

If you find one, please email me. It might exist in the literature somewhere.
If not, most likely it could be computed numerically using the "PESTO" Matlab package:
`http://www.di.ens.fr/~ataylor/share/PESTO_CDC_2017.pdf`
`https://github.com/AdrienTaylor/Performance-Estimation-Toolbox`

The first student to email me a correct, tight, worst-case bound for PGM earns 20 extra credit HW points.
Even more valuable than the points, you will have the satisfaction of knowing that next year's course notes will be more complete.

[1] D. Kim and J. A. Fessler. "Adaptive restart of the optimized gradient method for convex optimization". In: *J. Optim. Theory Appl.* 178.1 (July 2018), 240–63.