load_and_refit

February 21, 2019

1 Step 2: Refit.

In this notebook, we calculate the parameters used for exact CV by refitting the model initially fit in step one, the notebook fit_model_and_save.

For expository purposes this notebook calculates the refit for only one weight vector. To compute exact CV, one would perform the corresponding computation for all leave-k-out weight vectors.

```
In [1]: from copy import deepcopy
        import inspect
        import matplotlib.pyplot as plt
        %matplotlib inline
        import numpy as np
        import sys
        import time
        np.random.seed(3452453)
        import paragami
        from aistats2019_ij_paper import regression_mixture_lib as rm_lib
        from aistats2019_ij_paper import saving_gmm_utils
        from aistats2019_ij_paper import mse_utils
        import plot_utils_lib
In [2]: # Load the initial fit.
        # This file was produced by the notebook ``fit_model_and_save``.
        initial_fit_infile = '../fits/initial_fit.npz'
        full_fit, gmm, regs, metadata = \
            saving_gmm_utils.load_initial_optimum(initial_fit_infile)
        timepoints = metadata['timepoints']
Initializing FitDerivatives.
Using provided t_jac.
Using provided full_hess.
```

First, choose some timepoints to leave out.

The number of points left out (that is, k) is given by num_times , which is 1. The largest time-point we leave out is given by $max_num_timepoints$, which is 7. Because later timepoints are not affected by the smoothing, there is no reason to leave them out.

There are a certain number of ways to leave k out of 7 timepoints, and which_comb chooses one of them in the order given by the function itertools.combinations. Of course, when k=1, which_comb simply chooses which timepoint to leave out. mse_utils.get_indexed_combination maps which_comb to particular timepoints in a consistent way.

Full exact CV would run this script for all 7 choose *k* values of which_comb.

Because we have repeated measurements at each timepoint, leaving out a single timepoint will correspond to leaving out multiple row of the observation matrix. Those rows are determined by mse_utils.get_time_weight, which also returns a weight vector setting these observations' weights to zero.

We now re-optimize with the new weights.

Note that we could either start the optimization at the initial optimum (a "warm start") or do a fresh start from k-means. A fresh start is more time consuming but a more stringent test for the accuracy of the IJ. We calculate both, but report results from the fresh start in the paper. In the notebook examine_and_save_results, you can choose to examine either set of results.

Here, for consistency with the paper, we re-initialize with k-means.

```
gmm.set_regression_params(reg_params_w)
        init_gmm_params = \
            rm_lib.kmeans_init(gmm.transformed_reg_params,
                               gmm.num_components, 50)
        init_x = gmm.gmm_params_pattern.flatten(init_gmm_params, free=True)
        opt_time = time.time()
        gmm_opt, init_x2 = gmm.optimize(init_x, gtol=1e-2)
        print('\tUpdating preconditioner...')
        kl_hess = gmm.update_preconditioner(init_x2)
        print('\tRunning preconditioned optimization...')
        gmm.conditioned_obj.reset()
        reopt, gmm_params_free_w = gmm.optimize_fully(init_x2, verbose=True)
        print(gmm_opt.message)
        opt_time = time.time() - opt_time
        print('Refit time: {} seconds'.format(opt_time))
Iter 0: f = -153.38003431
Iter 1: f = -152.49438715
Iter 2: f = -153.69147895
Iter 3: f = -153.83779915
Iter 4: f = -154.02397812
Iter 5: f = -153.41393391
Iter 6: f = -154.10396420
Iter 7: f = -154.14366282
Iter 8: f = -154.14261201
Iter 9: f = -154.16417745
Iter 10: f = -154.18307547
Iter 11: f = -154.20711481
Iter 12: f = -154.22118064
Iter 13: f = -154.27402715
Iter 14: f = -154.28739474
Iter 15: f = -154.33849929
Iter 16: f = -154.03580241
Iter 17: f = -154.35421130
Iter 18: f = -154.36910489
Iter 19: f = -154.36872458
Iter 20: f = -154.37238982
Iter 21: f = -154.37722095
Iter 22: f = -154.38186985
Iter 23: f = -154.38410992
        Updating preconditioner...
        Running preconditioned optimization...
Preconditioned iteration 1
```

```
Running preconditioned optimization.
Iter 0: f = -154.38410992
Iter 1: f = -154.38423176
Iter 2: f = -154.38584092
Iter 3: f = -154.21889674
Iter 4: f = -154.42200228
Iter 5: f = -154.39603234
Iter 6: f = -154.39957947
Iter 7: f = -154.41374585
Iter 8: f = -154.43397491
Iter 9: f = -154.43484046
Iter 10: f = -154.43484816
Iter 11: f = -154.43484816
Preconditioned iteration 2
  Getting Hessian and preconditioner.
 Running preconditioned optimization.
Iter 12: f = -154.43484816
Iter 13: f = -154.43484816
Converged.
Optimization terminated successfully.
Refit time: 14.35115647315979 seconds
   We now save the results.
In [6]: gmm_params_w = \
            full_fit.comb_params_pattern['mix'].fold(
                gmm_params_free_w, free=True)
        refit_comb_params = {
            'mix': gmm_params_w,
            'reg': reg_params_w }
        refit_comb_params_free = \
            full_fit.comb_params_pattern.flatten(refit_comb_params, free=True)
In [7]: save_filename = \
            '../fits/refit__num_times{}__which_comb{}.npz'.format(
                    args.num_times, args.which_comb)
        print('Saving to {}'.format(save_filename))
        saving_gmm_utils.save_refit(
            outfile=save_filename,
            comb_params_free=refit_comb_params_free,
            comb_params_pattern=full_fit.comb_params_pattern,
            initial_fit_infile=initial_fit_infile,
            time_w=new_time_w,
            lo_inds=lo_inds,
            full_lo_inds=full_lo_inds)
Saving to ../fits/refit__num_times1__which_comb1.npz
```