calculate_prediction_errors

February 21, 2019

1 Step 3: Calculate the IJ and prediction errors.

In this notebook, for a single weight vector, we calculate the IJ itself as well as the prediction errors for exact CV and IJ. This notebook uses the output of the notebooks `load_and_refit` and `fit_model_and_save`.

```
In [1]: import numpy as np
        import paragami
        import vittles
        import scipy as sp
        from scipy import sparse
        import time

        import seaborn as sns
        import pandas as pd

        import matplotlib.pyplot as plt
        %matplotlib inline

        np.random.seed(3452453)

        from aistats2019_ij_paper import regression_lib as reg_lib
        from aistats2019_ij_paper import sensitivity_lib as sens_lib
        from aistats2019_ij_paper import saving_gmm_utils
        from aistats2019_ij_paper import mse_utils

        import plot_utils_lib
```

```
In [2]: # Simulate passing arguments in on the command line.
        class Args():
            def __init__(self):
                pass

        args = Args()
        args.num_times = 1
        args.which_comb = 1
        args.max_num_timepoints = 7
```

```
In [3]: ###############################
        # Load the original fit.

        print('Loading original fit.')
        initial_fit_infile = '../fits/initial_fit.npz'
        full_fit, gmm, regs, initial_metadata = \
            saving_gmm_utils.load_initial_optimum(initial_fit_infile)

        opt_comb_params = full_fit.get_comb_params()
```

```
Loading original fit.
Initializing FitDerivatives.
Using provided t_jac.
Using provided full_hess.
```

```
In [4]: ###############################
        # Load the test data

        test_regression_infile = '../fits/test_regressions.json'
        with open(test_regression_infile) as infile:
            regs_test = reg_lib.Regressions.from_json(infile.read())

        ##########################################
        # Load a refit as specfified by ``args``.

        refit_filename = \
            '../fits/refit__num_times{}__which_comb{}.npz'.format(
                    args.num_times, args.which_comb)

        comb_params_free_refit, comb_params_pattern_refit, refit_metadata = \
            saving_gmm_utils.load_refit(refit_filename)

        time_w = refit_metadata['time_w']
        lo_inds = refit_metadata['lo_inds']
        full_lo_inds = refit_metadata['full_lo_inds']

        assert(comb_params_pattern_refit == full_fit.comb_params_pattern)
        comb_params_refit = comb_params_pattern_refit.fold(
            comb_params_free_refit, free=True)

        time_w = refit_metadata['time_w']
        lo_inds = refit_metadata['lo_inds']
        full_lo_inds = refit_metadata['full_lo_inds']
```

The objects named `comb_params` refer to both the regression and clustering parameters. The name `free` refers to the unconstrained flat value for the parameters as calculated by `paragami`.

```
In [5]: print('Regression pattern: ',
              comb_params_pattern_refit['reg'])
```

```
        print('Clustering pattern: ',
              comb_params_pattern_refit['mix'])
```

```
Regression pattern:  OrderedDict:
	[beta_mean] = NumericArrayPattern (700, 10) (lb=-inf, ub=inf)
	[beta_info] = PatternArray (700,) of PDMatrix 10x10 (diag_lb = 0.0)
	[y_info] = NumericArrayPattern (700,) (lb=0.0, ub=inf)
Clustering pattern:  OrderedDict:
	[centroids] = NumericArrayPattern (10, 9) (lb=-inf, ub=inf)
	[probs] = SimplexArrayPattern (1,) of 10-d simplices
```

1.0.1 Calculate the infinitesimal jackknife.

The `vittles` package makes it easy to calculate linear approximations to the sensitivity of M-estimators to hyperparameters, of which the IJ is a special case. Here, the `HyperparameterSensitivityLinearApproximation` uses the sparse value of H_1 calculated earlier.

Note that H_1 is factorized during the initialization of `weight_sens`, and that it takes relatively little time.

```
In [6]: # Note that if you don't cast the jacobian to a numpy array from
        # a numpy matrix, the output is a 2d-array, causing confusion later.
        weight_sens = vittles.HyperparameterSensitivityLinearApproximation(
            objective_fun=lambda: 0,
            opt_par_value=full_fit.comb_params_free,
            hyper_par_value=regs.time_w,
            hessian_at_opt=sp.sparse.csc_matrix(full_fit.full_hess),
            cross_hess_at_opt=np.array(full_fit.t_jac.todense()))
```

We now use the `weight_sens` object to approximate the “free” value of the combined parameters at `time_w`. The IJ operates in unconstrained space, so we use `paragami` to fold the unconstrained vector back into a dictionary of parameters.

```
In [7]: # Get the infinitesimal jackknife for the refit weight vector.
        lr_time = time.time()
        comb_params_free_lin = \
            weight_sens.predict_opt_par_from_hyper_par(time_w)
        lr_time = time.time() - lr_time
        print('Infinitesimal jackknife time: {}'.format(lr_time))

        comb_params_lin = full_fit.comb_params_pattern.fold(comb_params_free_lin, free=True)
```

```
Infinitesimal jackknife time: 0.0011603832244873047
```

1.0.2 Calculate various prediction errors.

Recall that the prediction error is the difference between the data and the posterior expected cluster centroid for a particular gene. Let us consider the original optimal clustering parameters, `opt_comb_params['mix']`. To get the test set error on gene g for these parameters, we need to do the following steps:

1. Run the regression for gene g in the test set
2. Classify the regression, calculating $\mathbb{E}_{q_z^*}[z_g]$. This is a function of the clustering parameters and the regression line for gene g.
3. Calculate the expected posterior cluster centroid for gene g, which is $\mu_g^* = \sum_k \mathbb{E}_{q_z^*}[z_{gk}]\mu_k$.
4. Because the transformation discards the mean information, compare the de-meaned data to the estimated centroid: $error_{gt} = \left(y_{gt} - \frac{1}{T}\sum_{t'=1}^{T} y_{gt'}\right) - \mu_{gt}^*$.

Note that step one could re-run the regression either with the original weights or the new weights. We found that this decision does not matter qualitatively. Here and in the paper, we simply classify the original regression, but the notebook `examine_and_save_results` can produce results for oth the original and re-weighted regressions.

We will examine prediction error on the time points that are left out, that is, for observations in `full_lo_inds`.

```
In [8]: print('Calculating prediction error.')

        # Get the training set error on the full data.
        train_error = mse_utils.get_lo_err_folded(
            opt_comb_params,
            keep_inds=full_lo_inds,
            mse_regs=regs,
            mse_reg_params=opt_comb_params['reg'],
            gmm=gmm)

        ############
        # Original fit.

        # Get the optimal test set regressions.
        reg_params_test = regs_test.get_optimal_regression_params()

        # Get the test error for the original fit.
        orig_test_error = mse_utils.get_lo_err_folded(
            opt_comb_params,
            keep_inds=full_lo_inds,
            mse_regs=regs_test,
            mse_reg_params=reg_params_test,
            gmm=gmm)

        orig_pred = mse_utils.get_predictions(
            gmm, opt_comb_params['mix'], reg_params_test)

        # Get the test error for the CV refit.
        cv_error = mse_utils.get_lo_err_folded(
            comb_params_refit,
            keep_inds=full_lo_inds,
            mse_regs=regs_test,
            mse_reg_params=reg_params_test,
            gmm=gmm)

        cv_pred = mse_utils.get_predictions(
            gmm, comb_params_refit['mix'], reg_params_test)

        # Get the test error for the IJ approximation.
        ij_error = mse_utils.get_lo_err_folded(
            comb_params_lin,
            keep_inds=full_lo_inds,
            mse_regs=regs_test,
            mse_reg_params=reg_params_test,
            gmm=gmm)

        ij_pred = mse_utils.get_predictions(
            gmm, comb_params_lin['mix'], reg_params_test)
```

```
Calculating prediction error.
```

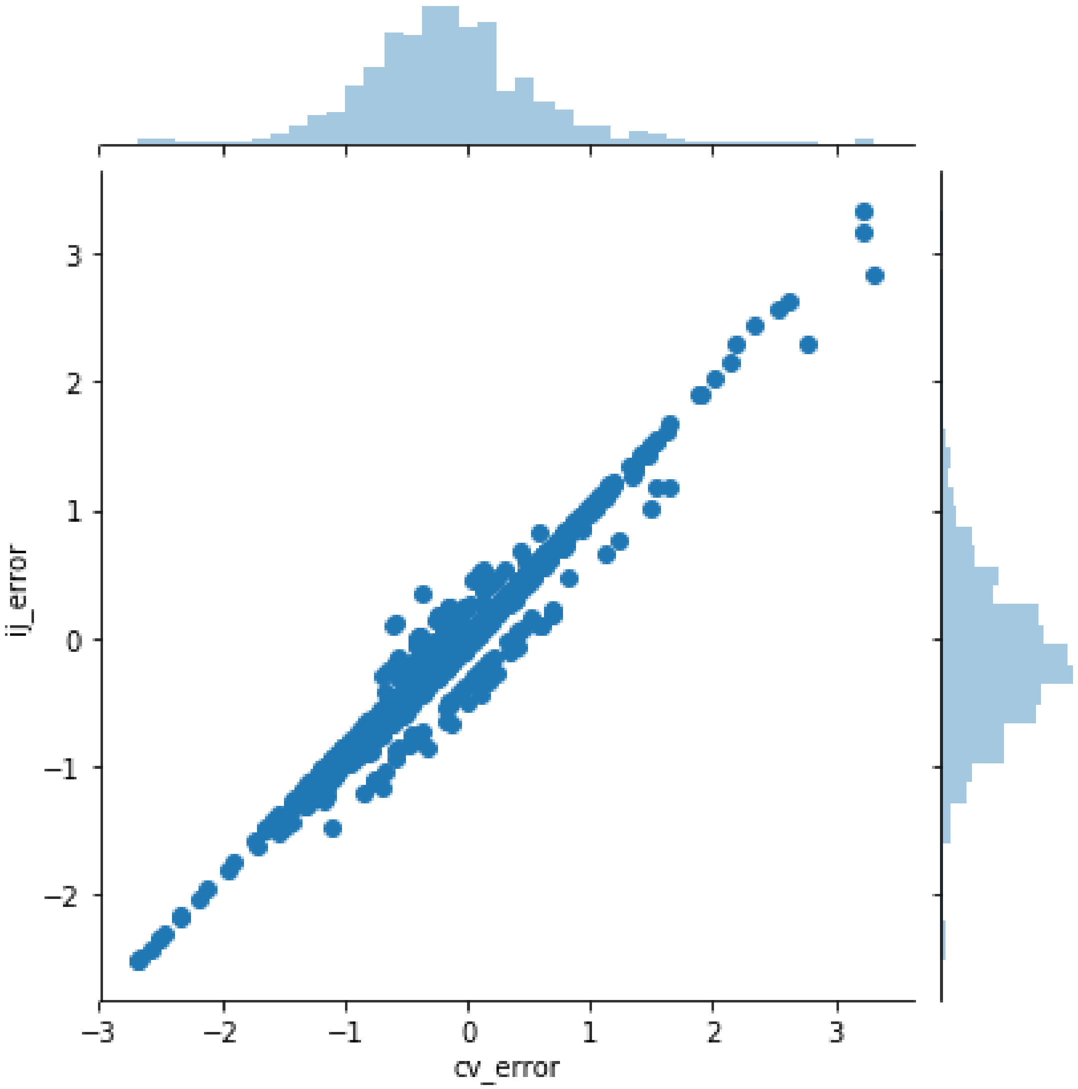
1.0.3 Selected results.

We now make a cursory comparison of the results. For a more detailed analysis, including the results that went into the paper, see the notebook `examine_and_save_results`.

```
In [9]: cv_excess_error = cv_error - orig_test_error
        ij_excess_error = ij_error - orig_test_error

        def GetColDf(col):
            return pd.DataFrame(
                {'cv_error': cv_error[:, col],
                 'cv_excess': cv_excess_error[:, col],
                 'ij_error': ij_error[:, col],
                 'ij_excess': ij_excess_error[:, col],
                 'col': col})

        result = pd.concat([ GetColDf(col) for col in range(len(full_lo_inds)) ])
```

If we simply look at the point-by-point error, CV and IJ are highly correlated.

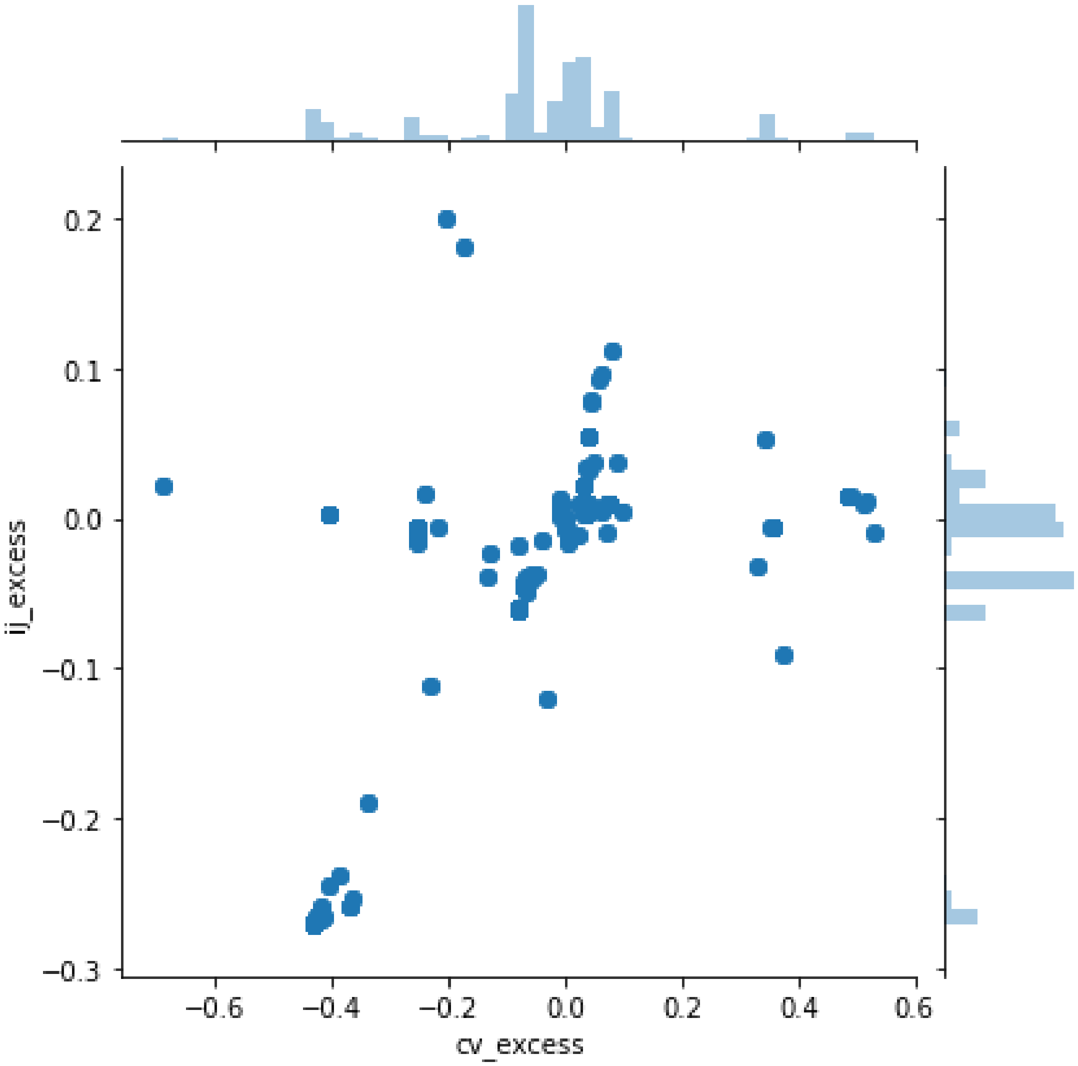
```
In [10]: sns.jointplot(x='cv_error', y='ij_error', data=result);
```



However, this is because the error in each point is dominated by the error at the original optimum. To meaningfully compare the IJ to CV, we should compare the difference between the IJ and CV error and the error at the original optimum. The distribution of these "difference-in-difference" errors is shown in the next plot.

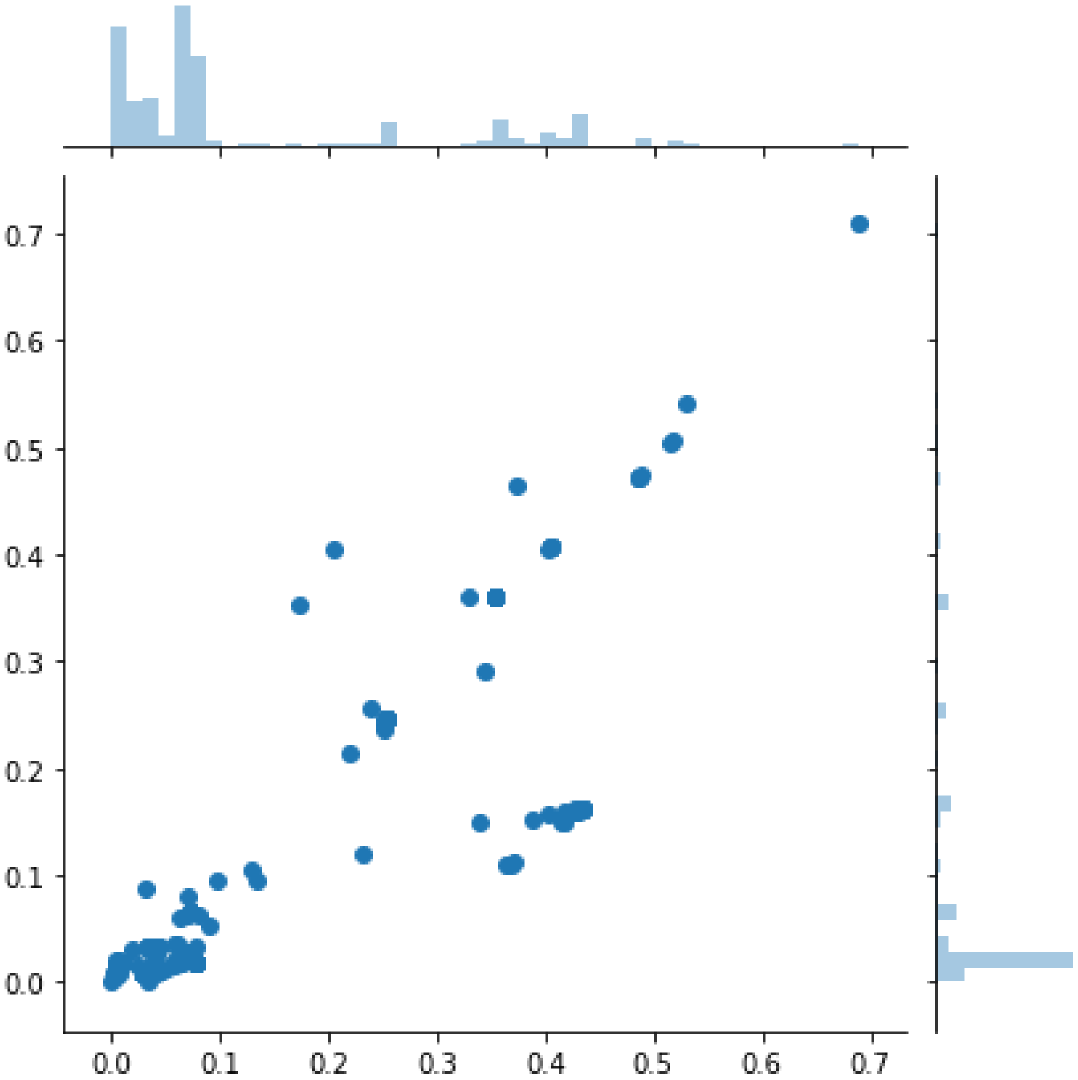
Some clear outliers can be seen. However, note that, in this case, overplotting makes IJ looks worse than it is – in the histograms you can see that most differences are very small.

```
In [11]: sns.jointplot(x='cv_excess', y='ij_excess', data=result);
```



As you might expect from a linear approximation, the IJ does the worst when the predicted change for CV is large.

```
In [12]: misfit = np.max(np.abs(cv_excess_error - ij_excess_error), axis=1)
         abs_cv_excess_error = np.max(np.abs(cv_excess_error), axis=1)

         sns.jointplot(abs_cv_excess_error, misfit)

Out[12]: <seaborn.axisgrid.JointGrid at 0x7f3f74fe1908>
```



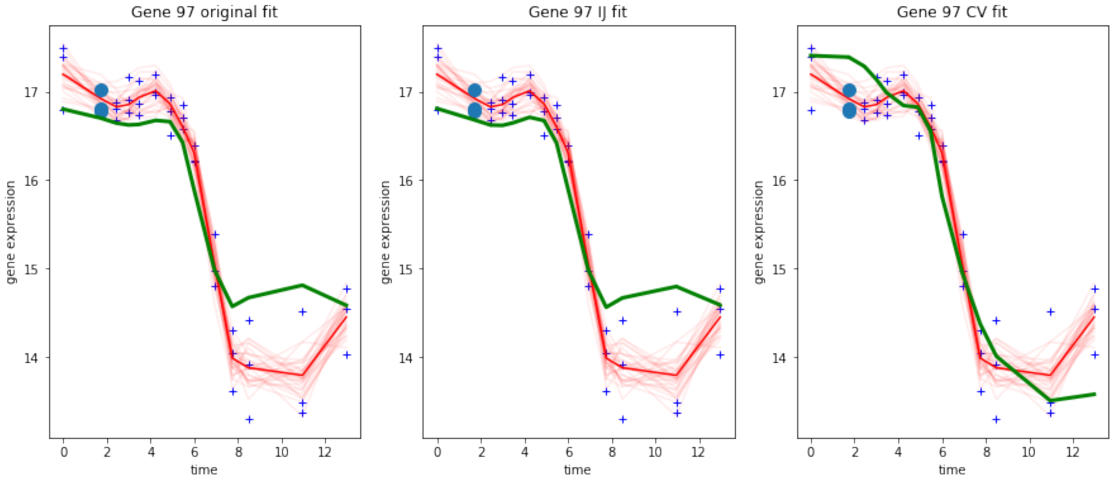
Finally, we visualize some of the genes where IJ badly misestimates the CV error. Clearly, in these cases, re-fitting with the left-out points (shown with large dots) produced large changes that the IJ did not capture. In general, it appears that the IJ errs relative to CV by not moving far enough from the original optimum.

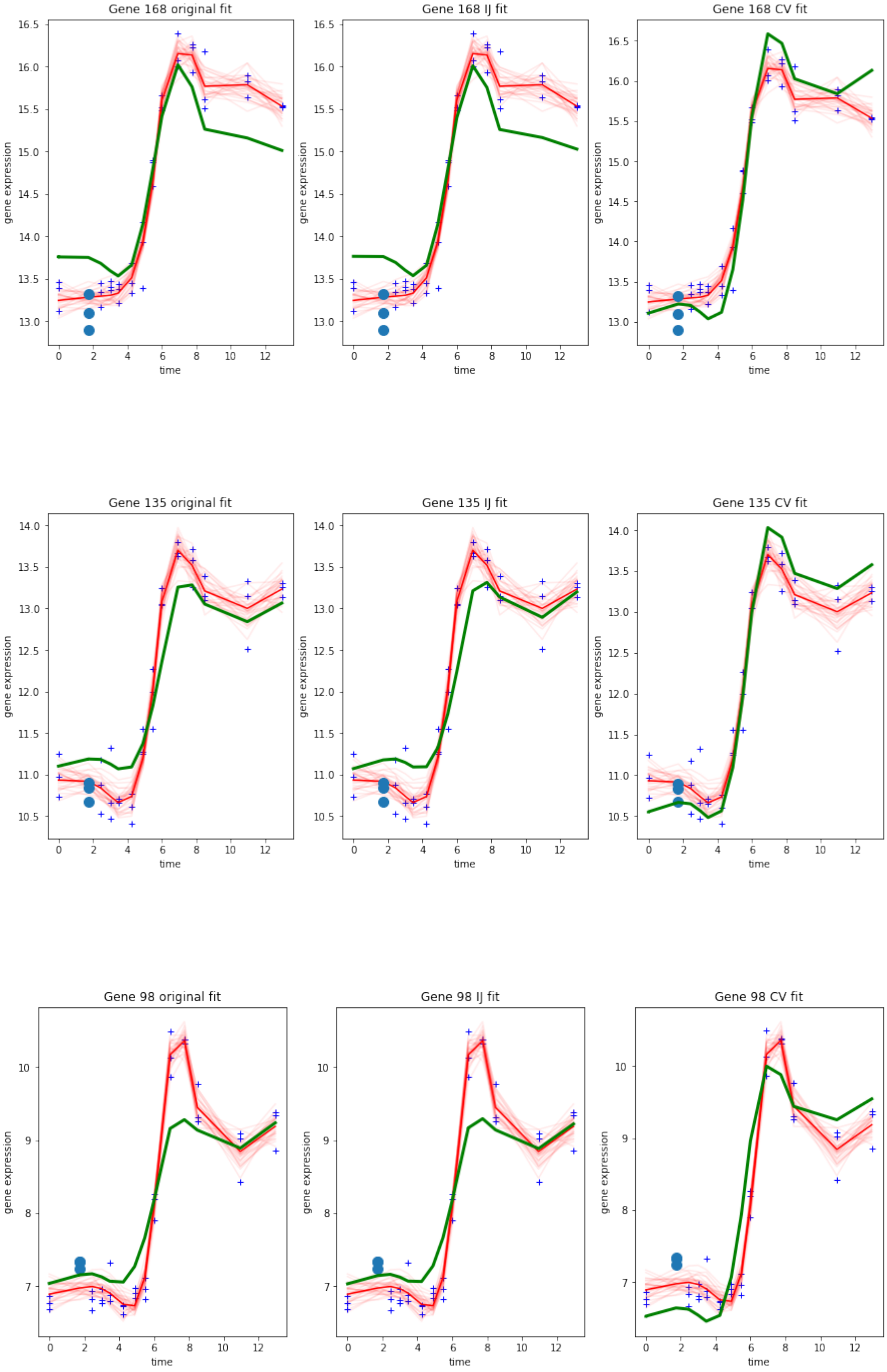
Despite the poor fit on these extreme genes, we stress that most genes exhibited small changes in both CV and IJ. For these genes, IJ performs well enough to capture salient aspects of the estimated out-of-sample error. For more detailed analysis of this point, see the notebook `examine_and_save_results`.

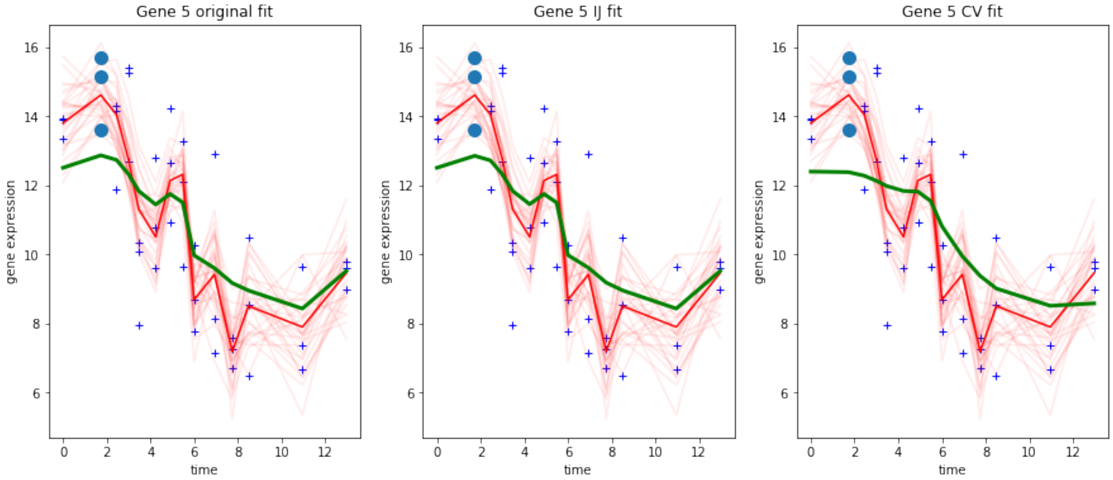
```
In [13]: timepoints = initial_metadata['timepoints']
         timepoints_stretch = np.sqrt(timepoints)

         def PlotGenePredictions(gene_ind):
             _, figs = plt.subplots(1, 3, figsize=(15,6))
```

```
        for i in range(3):
            np.random.seed(42)
            plot_utils_lib.PlotRegressionLine(
                timepoints_stretch, regs_test, reg_params_test, gene_ind, this_plot=figs[i]
            figs[i].plot(timepoints_stretch[full_lo_inds],
                         regs_test.y[gene_ind, full_lo_inds], 'o', markersize=10)

        plot_utils_lib.PlotPredictionLine(
            timepoints_stretch, regs_test, orig_pred, gene_ind, this_plot=figs[0])
        figs[0].set_title('Gene {} original fit'.format(gene_ind))

        plot_utils_lib.PlotPredictionLine(
            timepoints_stretch, regs_test, ij_pred, gene_ind, this_plot=figs[1])
        figs[1].set_title('Gene {} IJ fit'.format(gene_ind))

        plot_utils_lib.PlotPredictionLine(
            timepoints_stretch, regs_test, cv_pred, gene_ind, this_plot=figs[2])
        figs[2].set_title('Gene {} CV fit'.format(gene_ind))
```

```
In [14]: worst_fits = np.argsort(-1 * misfit)

         for gene in worst_fits[0:5]:
             PlotGenePredictions(gene)
```




Gene 168 original fit
Gene 168 IJ fit
Gene 168 CV fit
16.5
16.0
15.5
15.0
14.5
14.0
13.5
13.0
gene expression
0
2
4
6
8
10
12
time
Gene 135 original fit
Gene 135 IJ fit
Gene 135 CV fit
14.0
13.5
13.0
12.5
12.0
11.5
11.0
10.5
gene expression
time
Gene 98 original fit
Gene 98 IJ fit
Gene 98 CV fit
10
9
8
7
gene expression
time


Gene 5 original fit
Gene 5 IJ fit
Gene 5 CV fit
gene expression
time
16
14
12
10
8
6
0
2
4
6
8
10
12