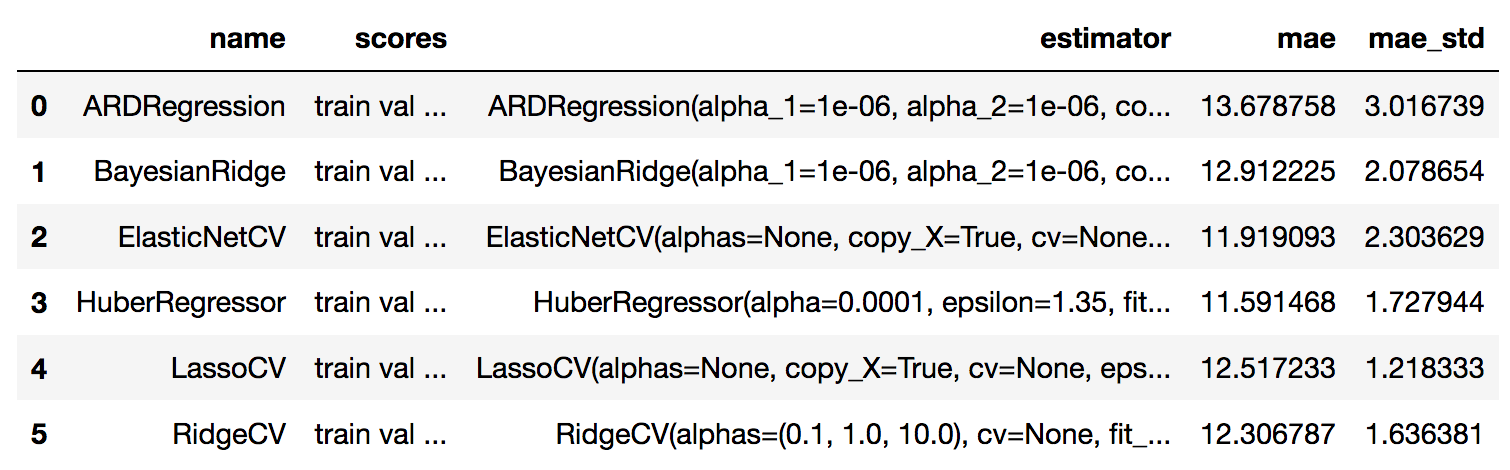
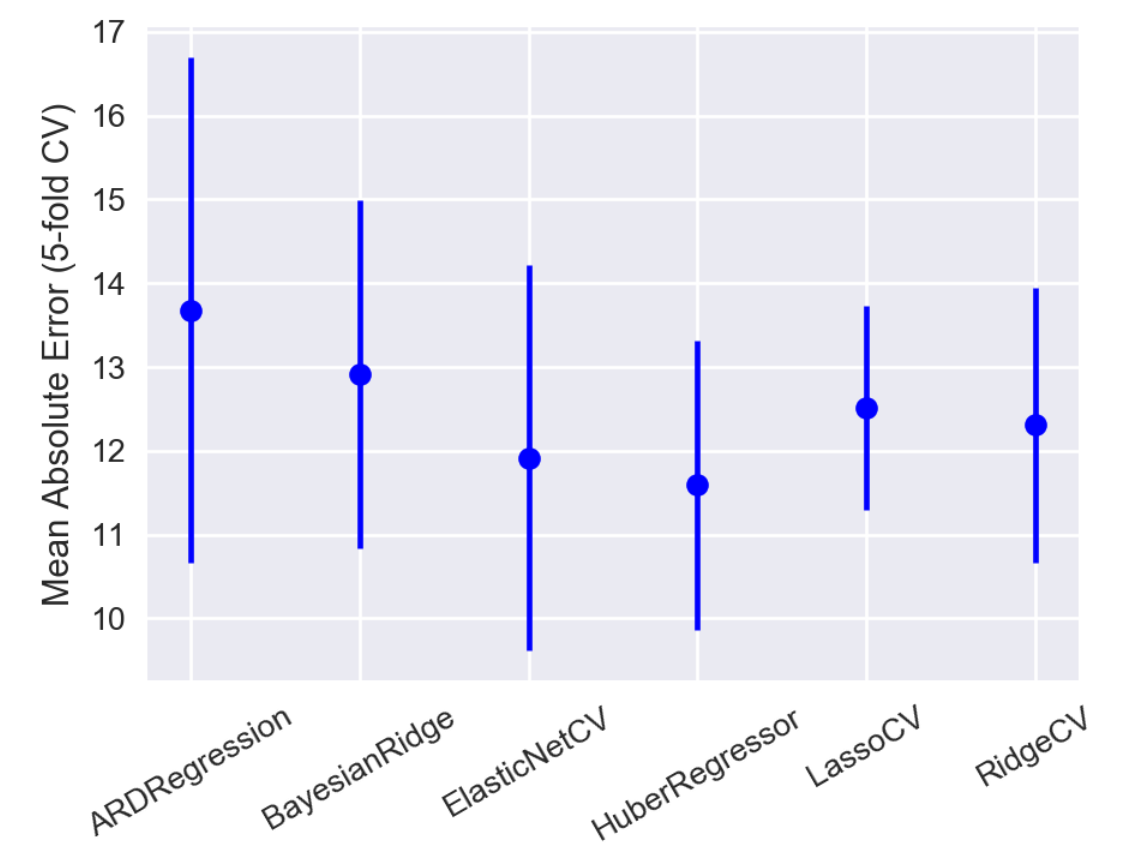
**Modeling Update**

**Week 1/2**

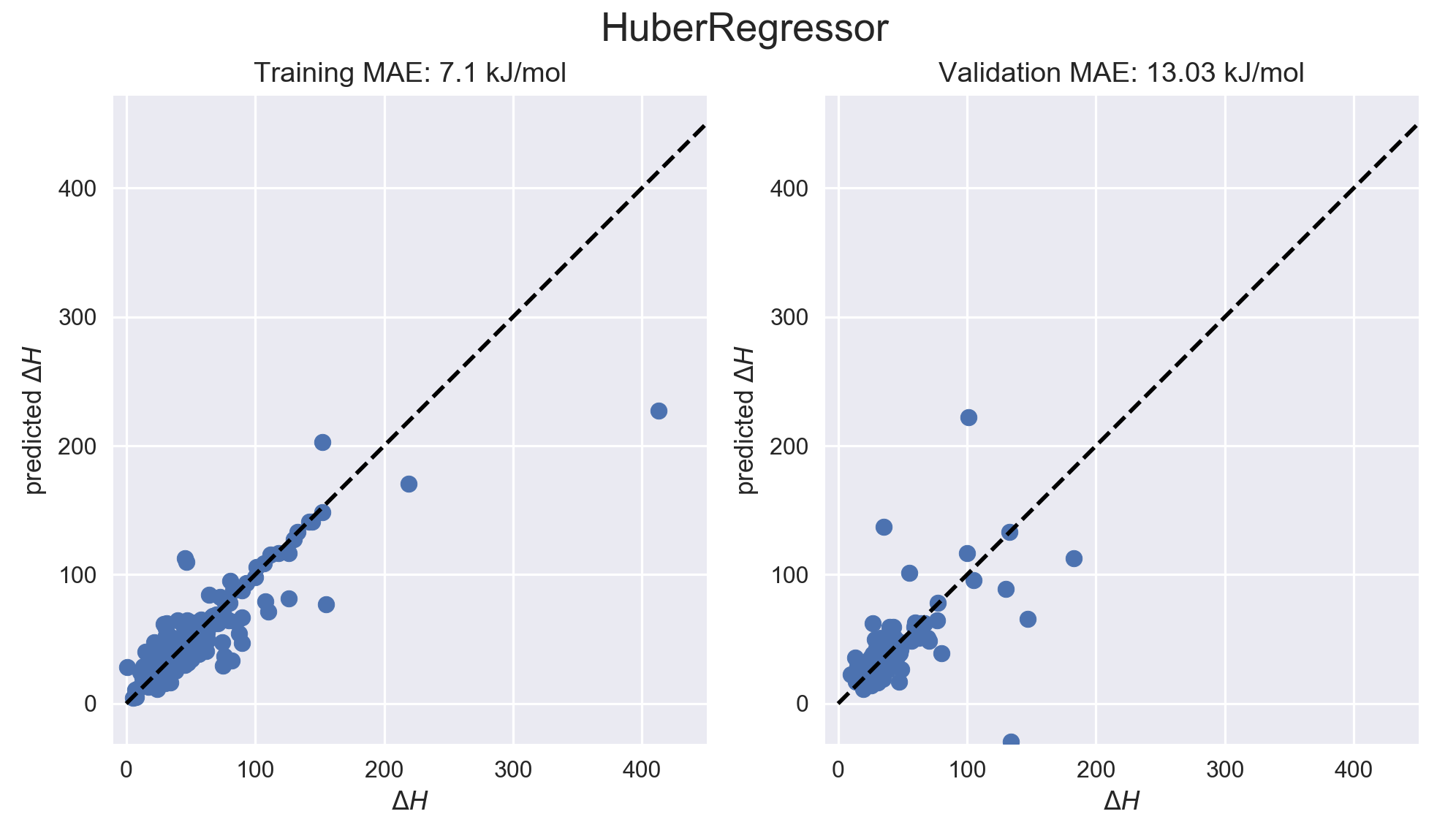
My first step was to "tool out" the notebook so that model evaluation is a single callable function. This way, I can iterate over "all the models" in sklearn (**linear\_model**.\_\_all\_\_) and store training/validation results in a dataframe. That looks like this:



"Estimator" is an actual fitted model object that I can call to run predictions on any data. Anything ending with "CV" is actually running its own internal 3-fold CV (I think you had passed a 10-fold CV in the original code) and choosing the best fit from that. "MAE" is the average of 5-fold cross-validated MAE on the validation set - so for this set of models (all of the regressors in the linear\_model module), the results look like this:

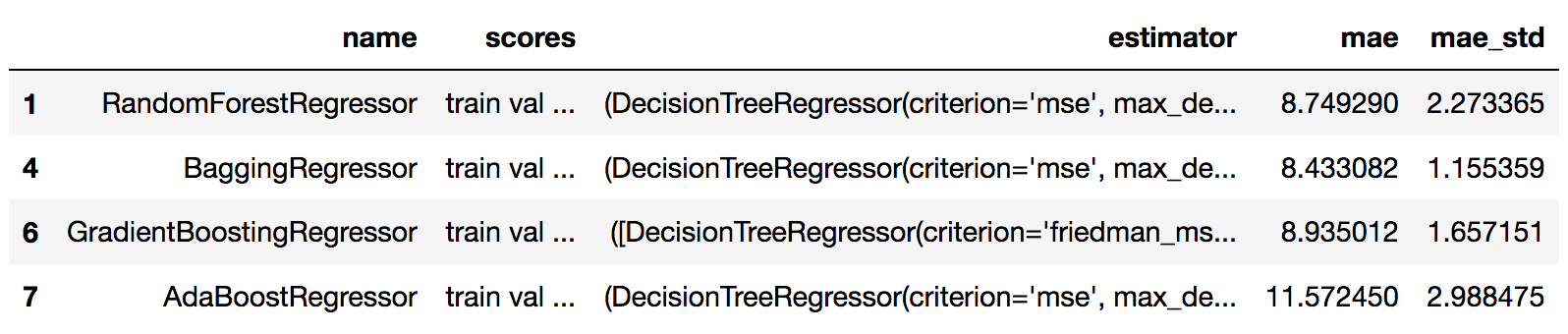


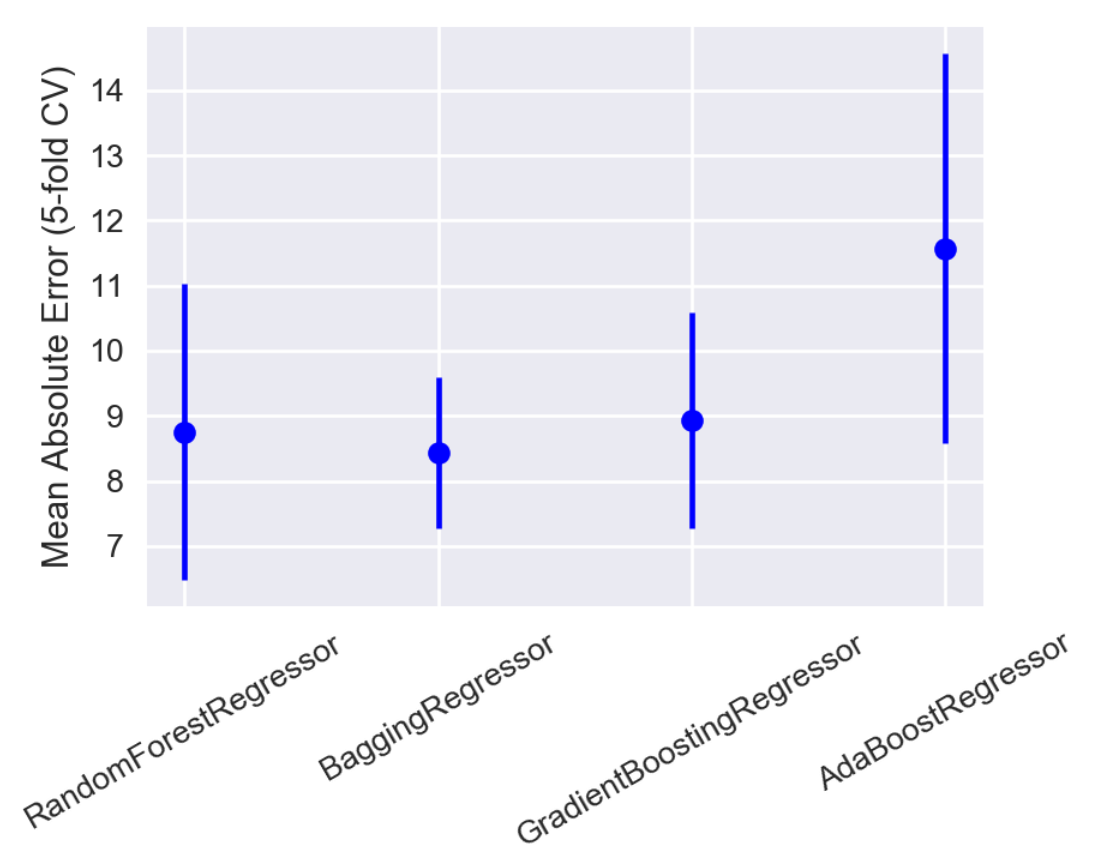
We can take a look at the training and validation parity plots for *e.g.* the HuberRegressor:



Regression trains well, but on validation it can result in some pretty ugly outliers. In the original notebook, RandomForest was the “best” as measured by validation MAE. Next we’ll iterate over some ensemble regressors.

In these model tables, the "scores" column is actually a sub-dataframe that has the train/val scores (including pearson, spearman, etc.) for every model fit during the 5-fold CV.

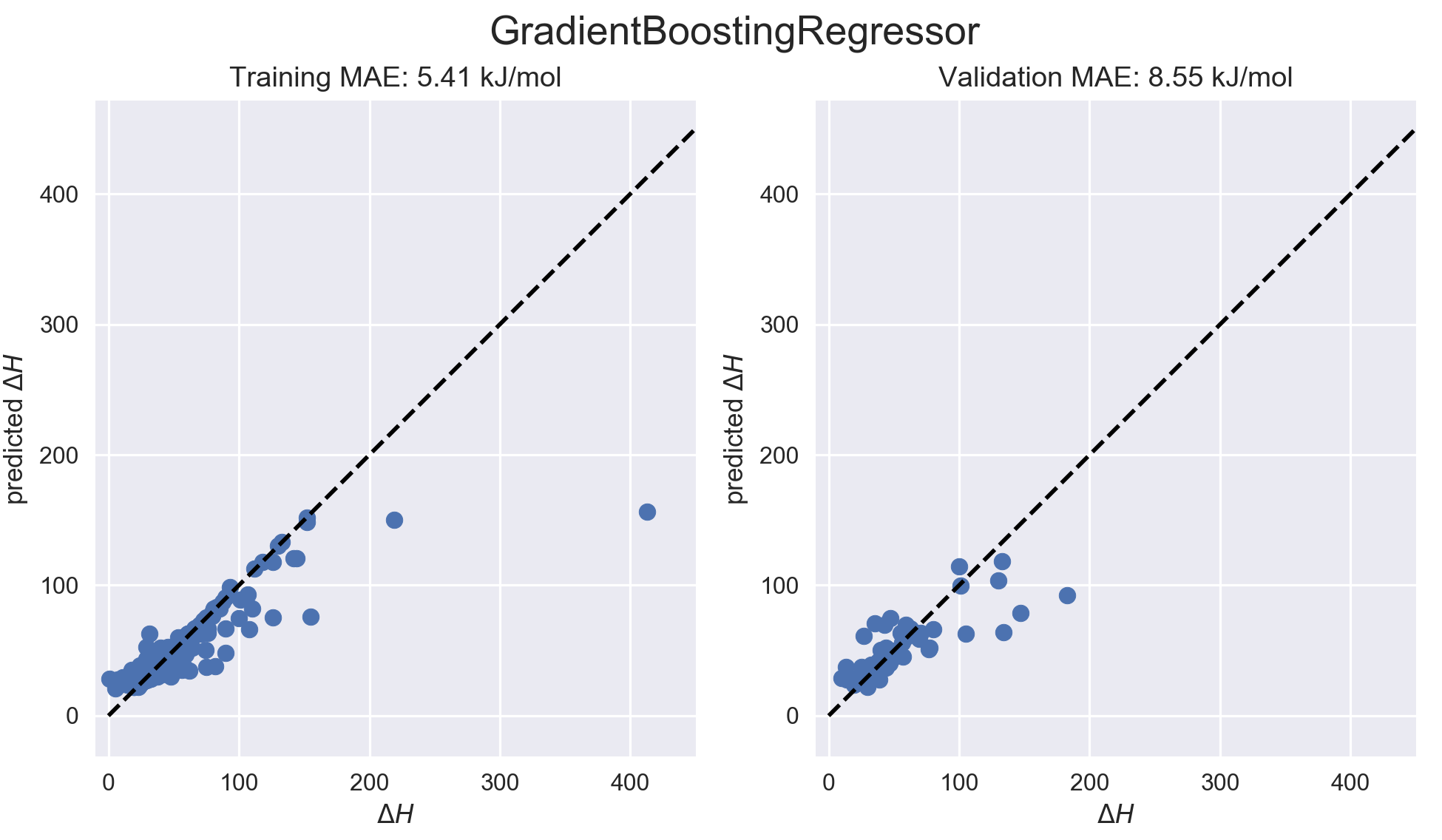




Every time I run this, the “best” model changes. In one run, GradientBoosting was best, and it tends to have a smaller standard deviation in its validation accuracy.

*Gradient Boosting*

It’s also an interesting model: its predictions are an average of an ensemble of ~100 decision trees. Each subsequent tree is fit on the “negative gradient of the loss function”; in other words, instead of fitting 100 random trees, it fits 1 tree, then another tree designed specifically to reduce the loss of the first tree, then another tree designed to reduce the loss of the first two, etc.



Validation performance is much better on all of these ensemble models. What’s interesting about Gradient Boosting is that it struggles with the high ∆H points; seems to underpredict them.

Gradient Boosting requires some optimization of the hyperparameters. I ran a GridSearchCV on the following space:

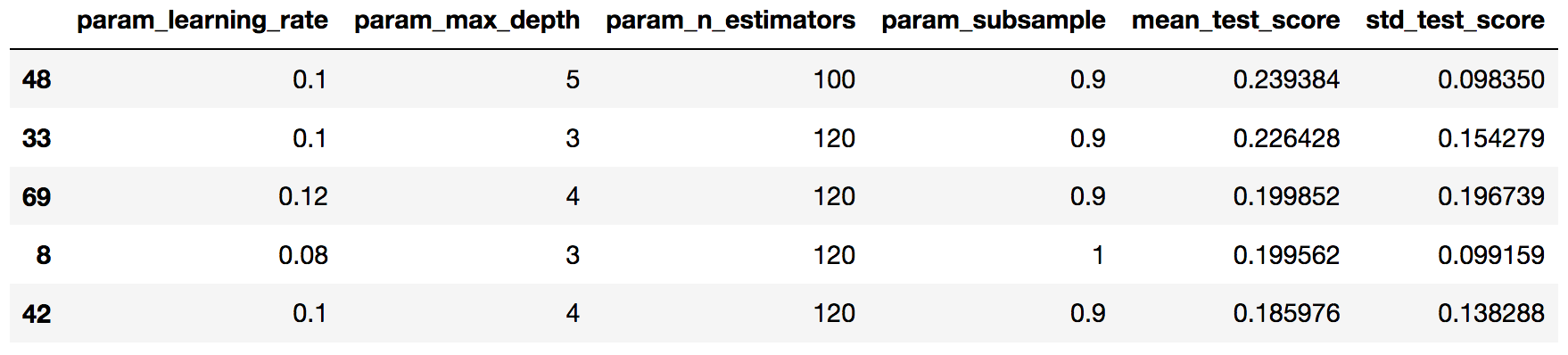
parameters = {'max\_depth': [3, 4, 5],

'learning\_rate': [0.08, 0.1, 0.12],

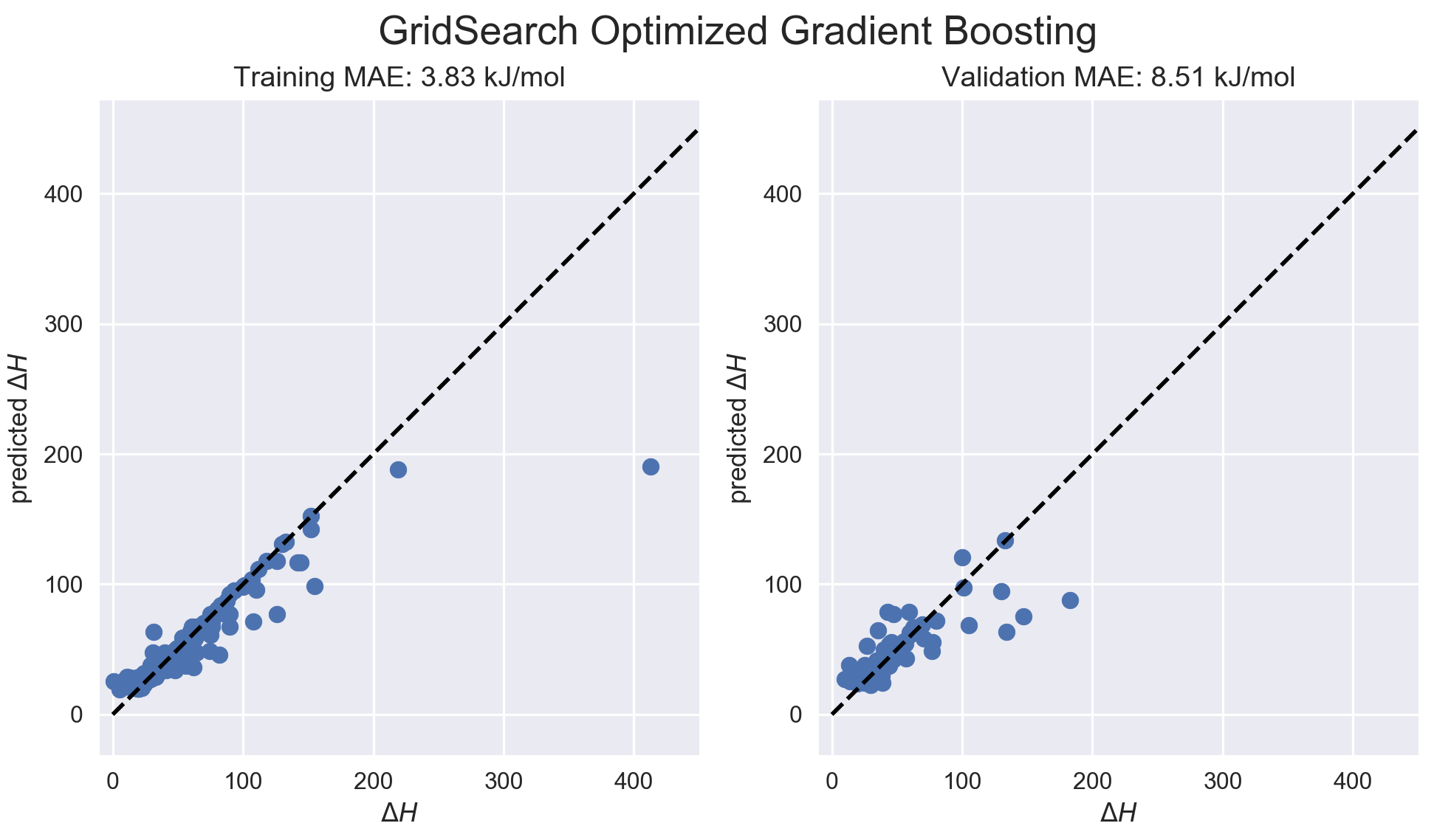
'n\_estimators': [80, 100, 120],

'subsample': [0.9, 0.95, 1]}

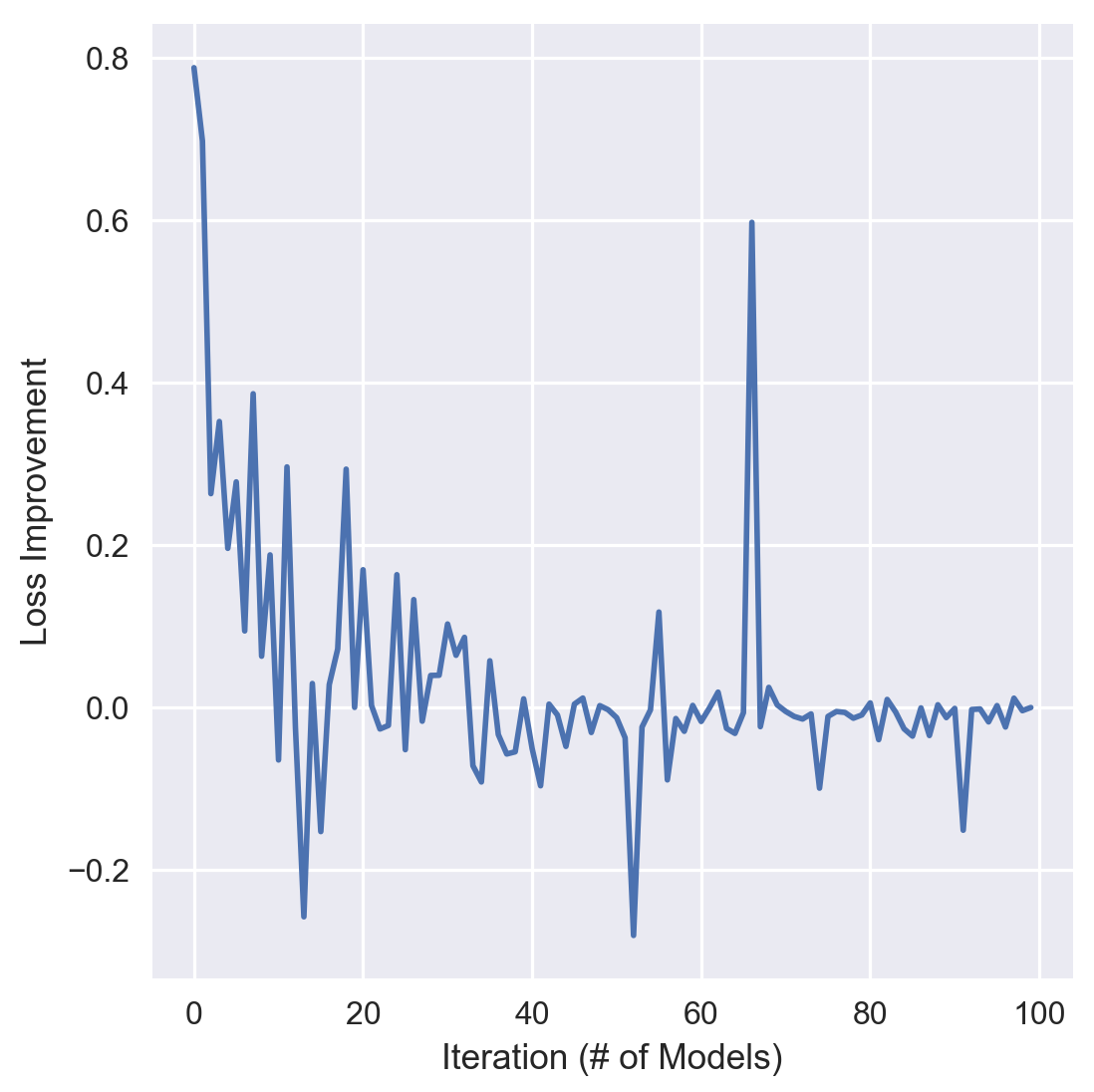
Max\_depth is the depth of the decision trees in the ensemble, learning rate indicates how strongly to follow the loss gradient at each iteration, n\_estimators is both the number of iterations of the training and the final number of trained trees in the ensemble, and subsample turns the algorithm into “stochastic gradient boosting”, meaning some fraction of the training data is used to fit each tree (here, 90%+).



Mean\_test\_score is the mean R2 metric from 4-fold CV at each set of hyperparameters. The mean\_test\_scores are all well within their neighbors’ large margins of error (std\_test\_score), so the lessons learned here aren’t very clear. 90% subsampling seemed to be the clearest benefit, a high number of estimators was good, but deeper trees enabled a lower n\_estimators. Learning rate seemed pretty arbitrary. Here’s what the parity plots look like for a fit on the original 80/20 train/val split used for the other parity plots:

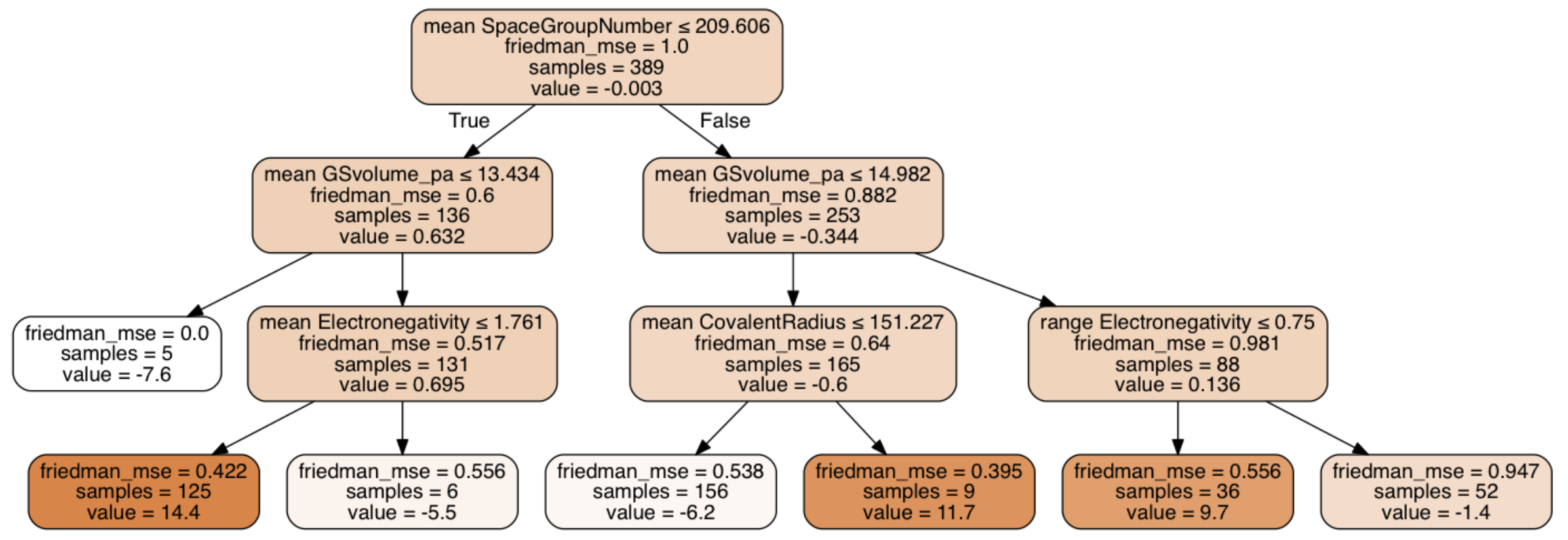


Ok, basically unchanged. To analyze how this is trained and what the features mean, there are a few things we can do. First, we can track how the loss function improved as we added models to the ensemble:



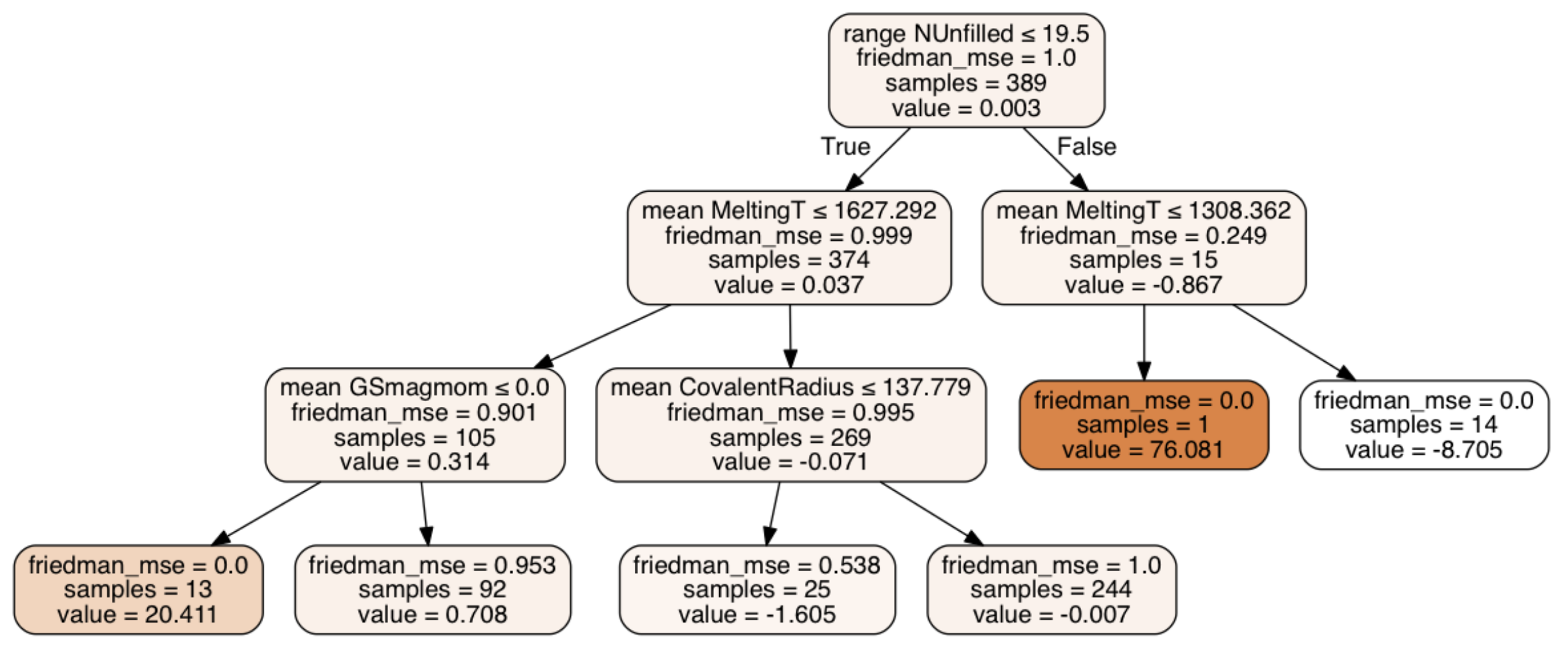
In this plot, a value >0 means the tree added at that step improved the model, a value <0 means the added tree actually made the model worse. At this point it would be useful to visualize individual trees.

It gets really convoluted to look at decision trees with depth of 5, so I used the parameters for the max\_depth=3 model. Here’s the first tree:



It splits first on “mean SpaceGroupNumber”, then “mean GSVolume\_pa”, and subsequent layers incorporate Electronegativity and CovalentRadius. The “value” of each leaf cell indicates the amount this tree will add to the running total of the prediction from the previous trees. A prediction from a gradient-boosted regressor is calculated by the following equation:

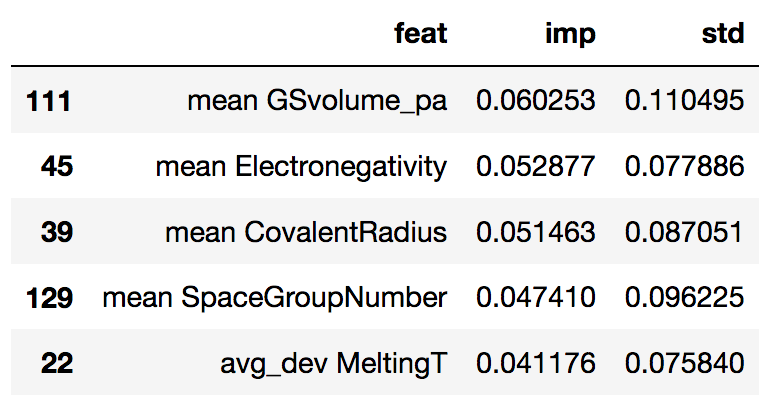
A prediction starts with the median of the training set (a 0th order model), then adds on “corrections” from each of the subsequent trees fitted to the residuals.



Here’s an example of a tree where one leaf is basically devoted to “correcting” for one sample with a very high ∆H. This would be an example of over-fitting.

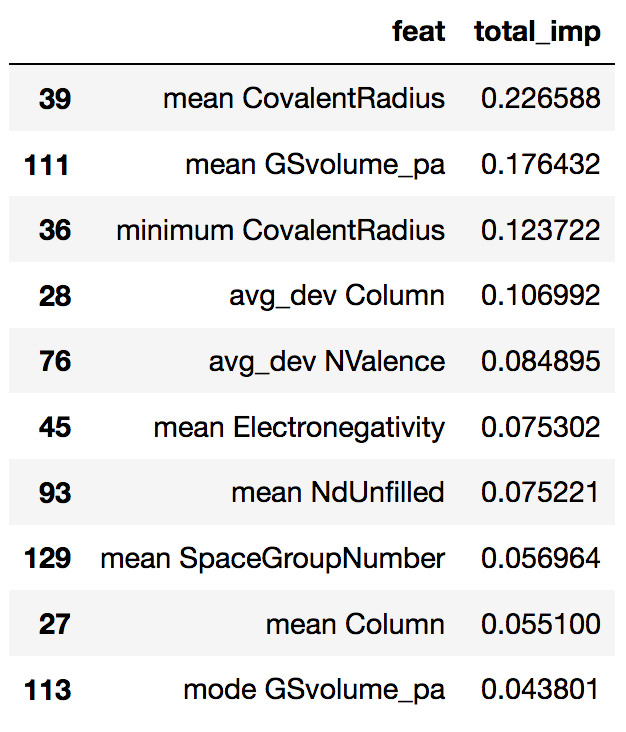
*Feature Importance*

The following table lists the top 5 features in order of importance, determined by their mean decrease in impurity:



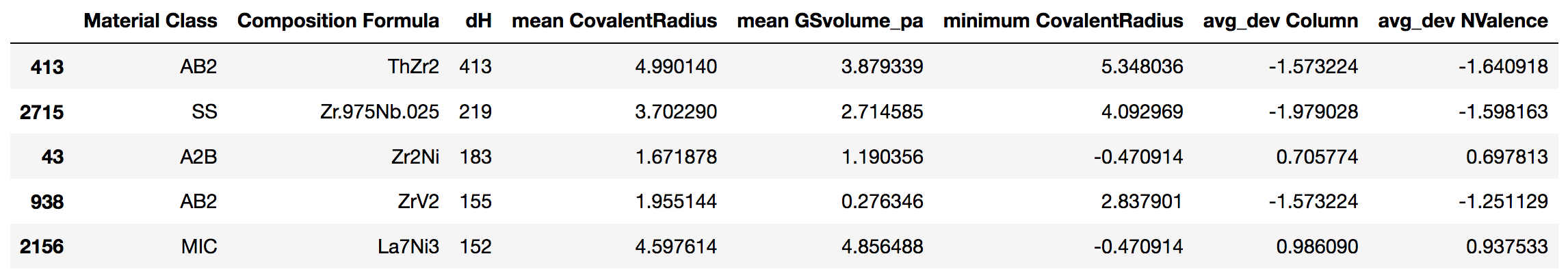
where “imp” is a relative importance, “std” is the standard deviation of that importance (notably larger than the values themselves). These features generally show up at the top of the individual trees.

We can do the same for the regular Random Forest model, then sum the importances over both models to get a more general idea of the most important features.



Covalent radius and GSvolume show up often.

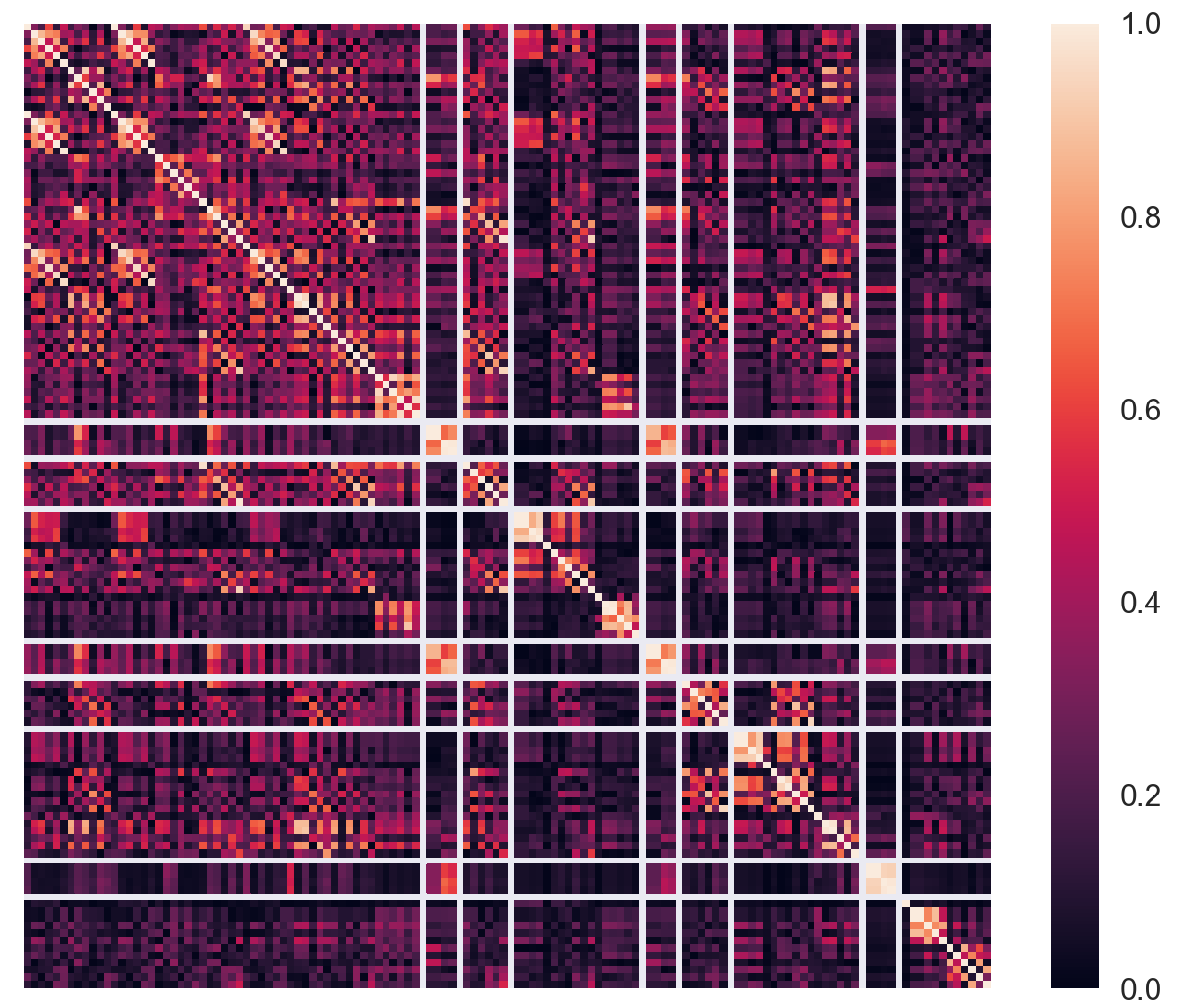
Finally, here is a table of the top five materials with highest ∆H, and the standardized values of the five most important features:

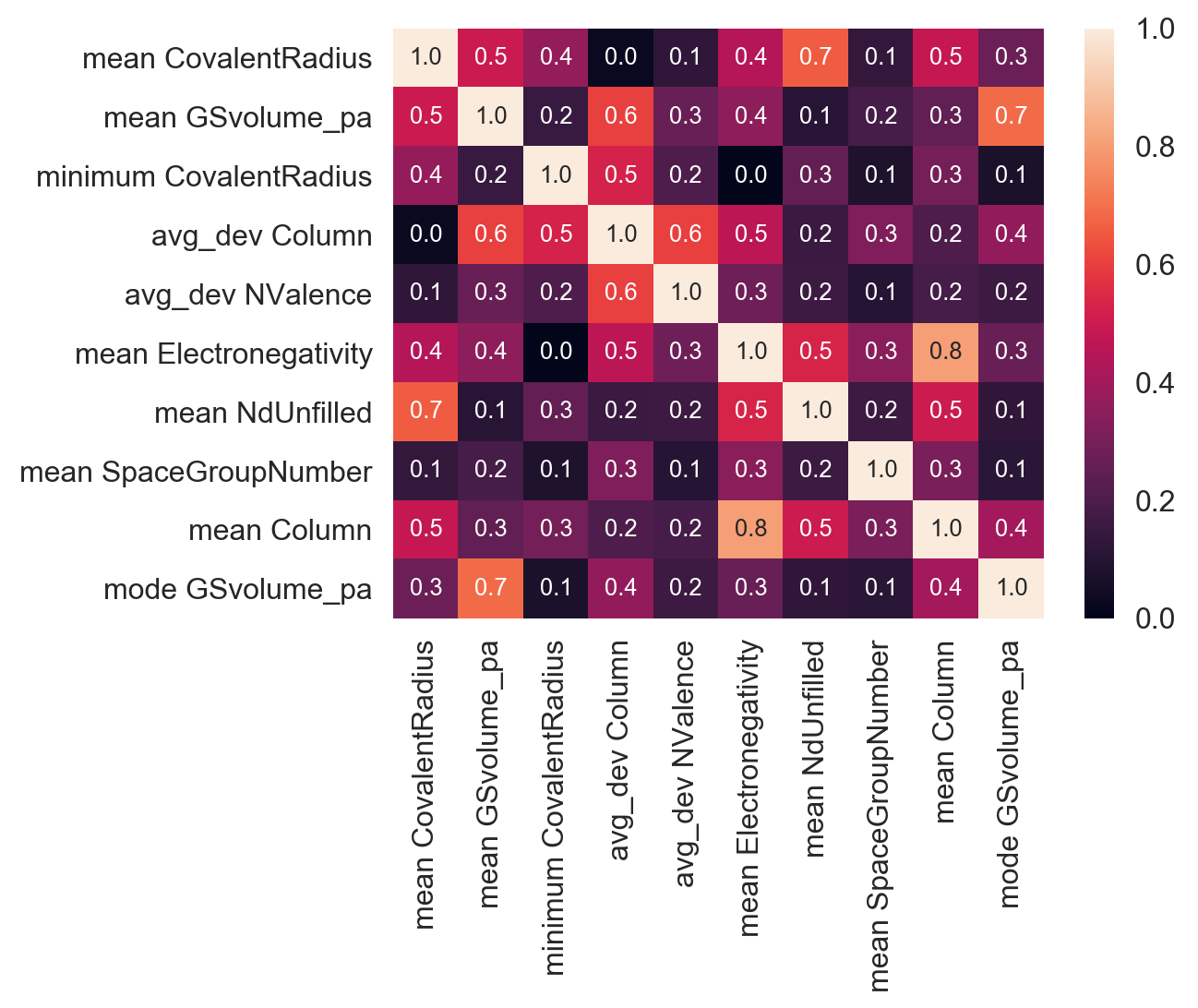


There may be some strong feature correlations here.

*Magpie Features*

The absolute feature importance is quite low for this model. Below is the correlation heat map of the Magpie features for this materials dataset, and the heat map for the top ten most important features:





These really are not that strongly correlated.