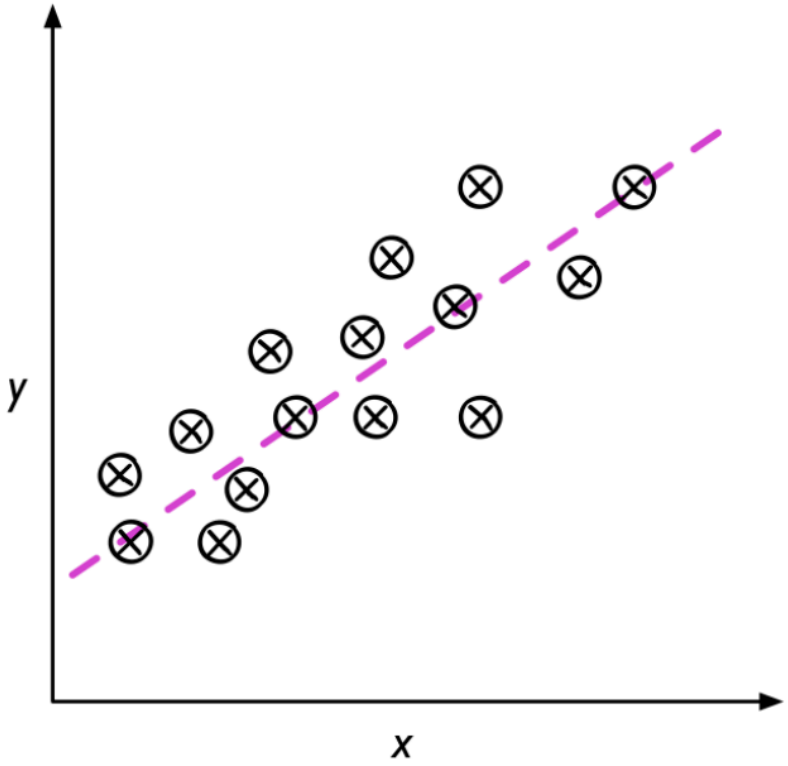
Goals for Today:

Supervised Learning

- Training/Testing
- Scikit Learn
- Linear Regression
- Regularization
- Partial Least Squares



Dow Dataset - Review

Recall from last time:

Removed Outliers: We used PCA to identify and remove periods when unit operations were offline

Redundant Descriptors: Even after removing outliers we still had many strongly correlated features (x variables).

This is potentially a problem:

i) Numerically unstable (i.e., rank deficient) due to multiple near-equivalent solutions.

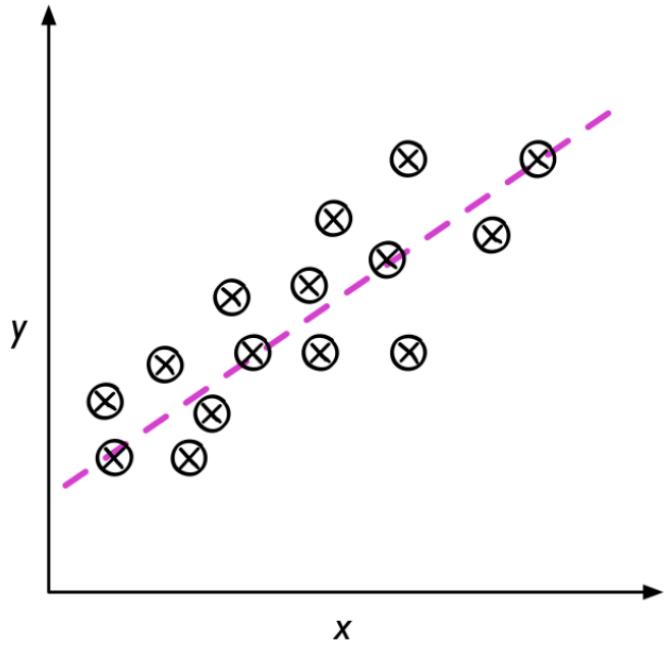
ii) Poor transferability (i.e., overfitting)

iii) Poor interpretability (i.e. due to i and ii, the parameters of the model will be uncertain)

Today we'll cover some of the methods for dealing with too many features in the context of the simplest supervised learning models (linear regression)

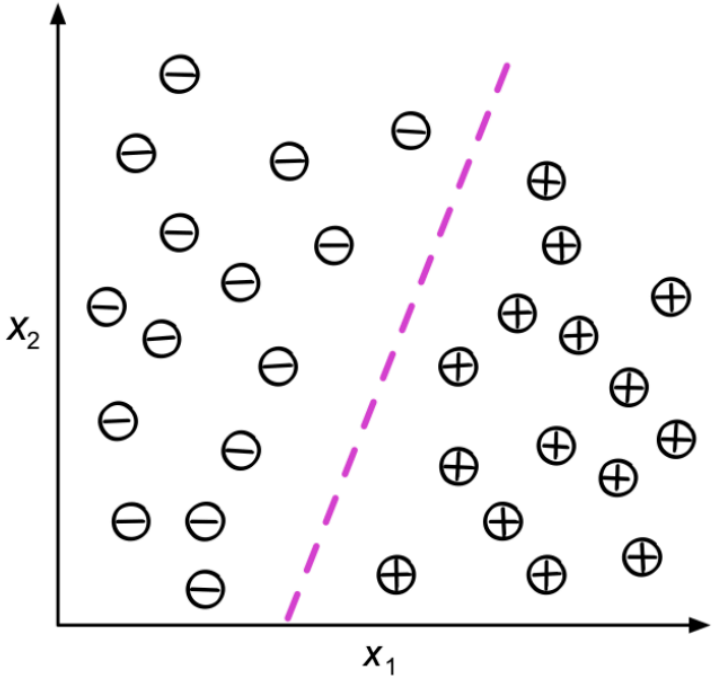
Supervised Learning - Recall

Regression



Regression models are used to explain relationships b/w features & continuous target variables

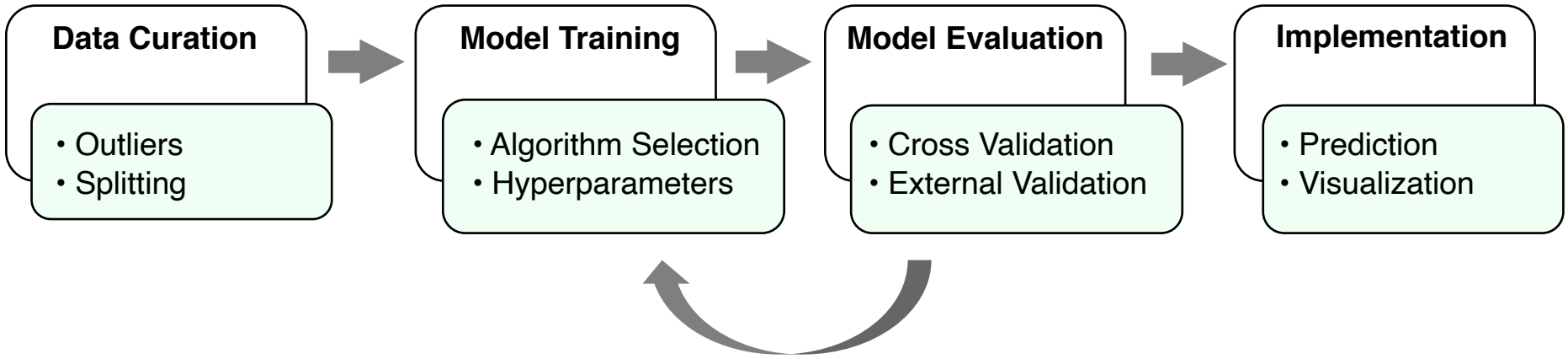
Classification



Classification models are used to distinguish b/w categorical target variables based on features

Typical goal: predict future (i.e., **y** at future times or for samples not in training data)

Supervised Learning - Workflow



- Labeled data is required (supplied by collaborator or part of your work)
- Training data and testing data are distinct (covered next)
- Validation on training data **is not the same thing** as predicting on unseen data.
- Domain knowledge plays an important role in all of these steps (curating the data, choosing the model, and evaluating if it makes sense)

Over the next month we will focus on the middle two boxes

Supervised Learning – Training Data

Training Data: The set of (x,y) samples that we will use to train our model

Dow Example:

Training Samples

Feature Variables

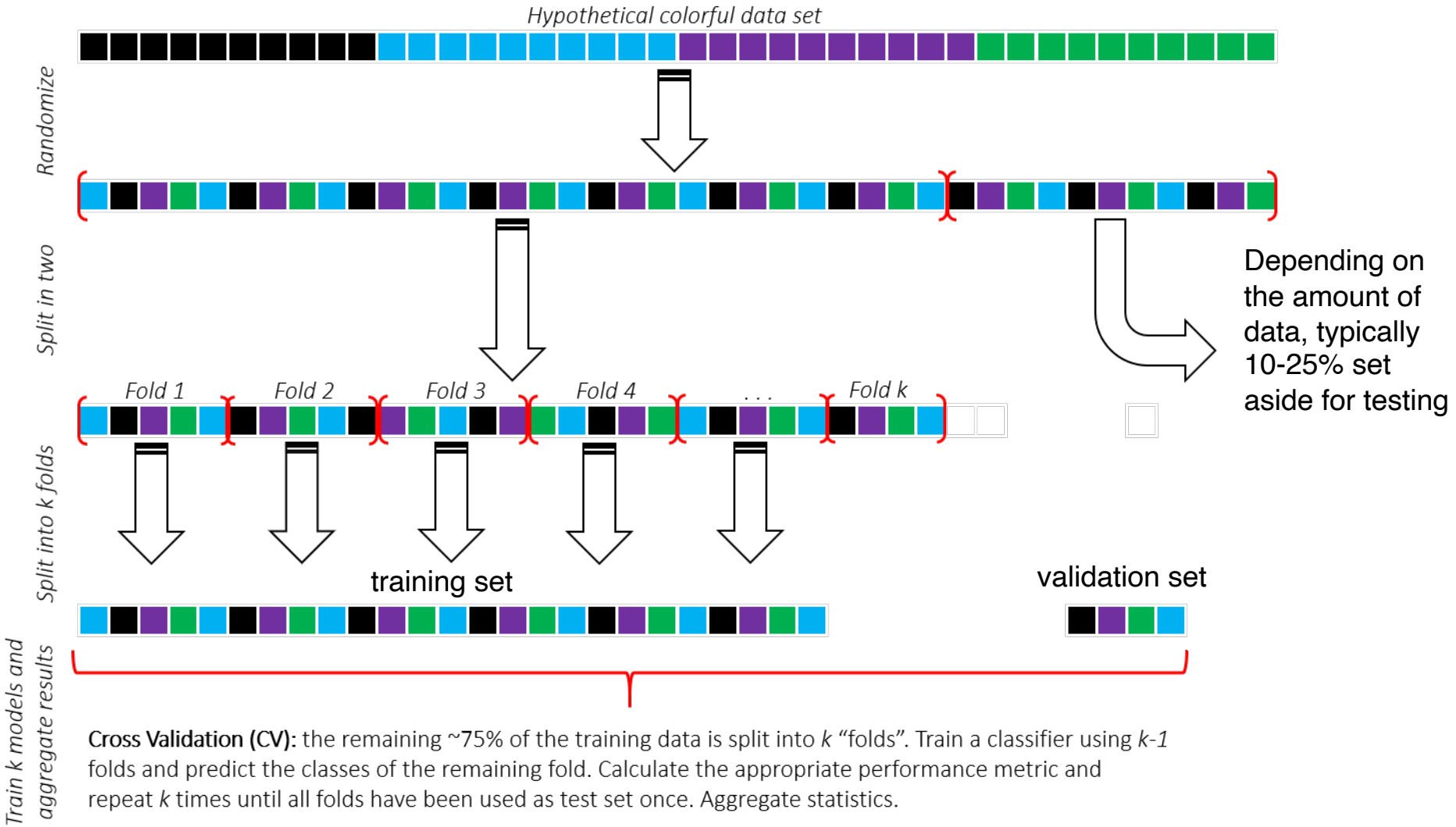
Date	x1	x2	x3	x4	x5	x6	x7 \
12/1/2015 0:00	327.813	45.7920	2095.06	2156.01	98.5005	95.4674	54.3476
12/1/2015 1:00	322.970	46.1643	2101.00	2182.90	98.0014	94.9673	54.2247
12/1/2015 2:00	319.674	45.9927	2102.96	2151.39	98.8229	96.0785	54.6130
12/1/2015 3:00	327.223	46.0960	2101.37	2172.14	98.7733	96.1223	54.9153
12/1/2015 4:00	331.177	45.8493	2114.06	2157.77	99.3231	94.7521	54.0925

Date	x40	y
12/1/2015 0:00	99.9614	1.77833
12/1/2015 1:00	99.8637	1.76964
12/1/2015 2:00	100.2490	1.76095
12/1/2015 3:00	100.3200	1.75226
12/1/2015 4:00	100.6590	1.74357

Target Variables

The underlying assumption is that our training data is **representative** of the **y** population we want to predict on.

Training/validation/testing concepts:

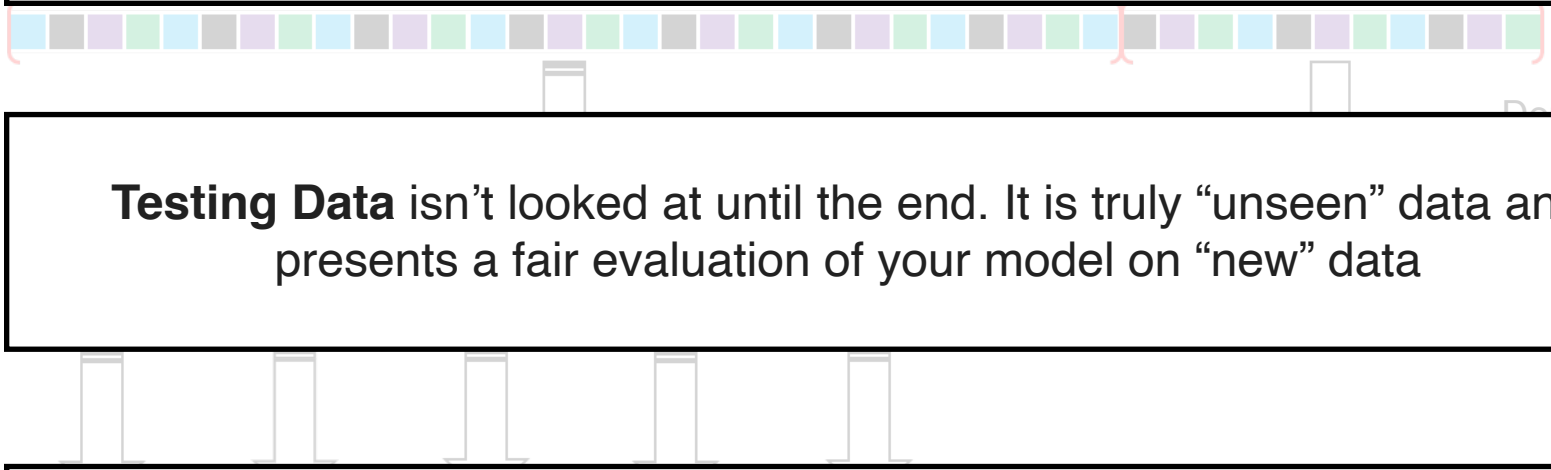


Cross Validation (CV): the remaining ~75% of the training data is split into *k* "folds". Train a classifier using *k-1* folds and predict the classes of the remaining fold. Calculate the appropriate performance metric and repeat *k* times until all folds have been used as test set once. Aggregate statistics.

Supervised Learning – Training Data

Training/validation/testing concepts:

Training Data is used to parameterize your model. This is the data that you can look at and tune with respect to.



Testing Data isn't looked at until the end. It is truly "unseen" data and presents a fair evaluation of your model on "new" data

Validation Data is used in the place of testing data during the training process. You can look at it, even though it doesn't directly enter the parameterization.

Cross Validation (CV): the remaining ~75% of the training data is split into *k* "folds". Train a classifier using *k-1* folds and predict the classes of the remaining fold. Calculate the appropriate performance metric and repeat *k* times until all folds have been used as test set once. Aggregate statistics.

Supervised Learning – Linear Regression

In linear regression we parameterize a model of the form:

$$\mathbf{y}_p = \beta_0 + \mathbf{x}_1\beta_1 + \ldots + \mathbf{x}_n\beta_n = \mathbf{x}\beta$$

*Implied "1" in first element of **x** matrix

The least squares solution corresponds to selecting parameters such that the square of the residuals is minimized:

$$\arg\min_{\beta} \sum_i \left(y_i - \mathbf{x}_i\beta\right)^2$$

This can be solved by taking the derivative and solving for the extremum. Or more concisely:

$$\mathbf{x}\beta = \mathbf{y} \xrightarrow{*} \beta = \left(\mathbf{x}^{\mathrm{T}}\mathbf{x}\right)^{-1}\mathbf{x}^{\mathrm{T}}\mathbf{y}$$

*For tall rank deficient matrices $\mathbf{x}^{-1}$ doesn't exist but the pseudo-inverse $(\mathbf{x}^{\mathrm{T}}\mathbf{x})^{-1}$ yields the least squares solution.

Relevant evaluation metrics:

Mean squared error (MSE):

$$\mathrm{MSE} = \frac{1}{N}\sum_{i}^{N}\left(y_i - y_{i,p}\right)^2$$

$y_{i,p}$ is the predicted value, y_i is the target value

This is the quantity that we seek to minimize in an ordinary least squares regression problem. Reports on average error, but is biased towards minimizing large deviations.

Coefficient of Determination (R^2):

$$R^2 = 1 - \frac{\sum_i^N \left(y_i - y_{i,p}\right)^2}{\sum_i^N \left(y_i - \mu_y\right)^2}$$

μ_y is the mean of the target value

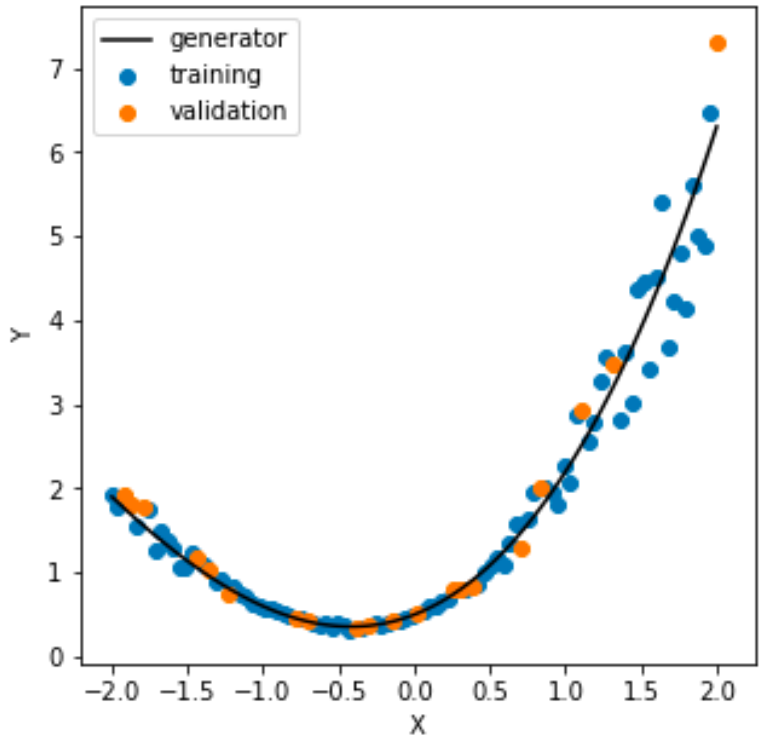
This is a number within (–inf,1). Quantity on the right is the squared error normalized by the variance of the target data, and can be interpreted as the fraction of unexplained variance in the predictions.

Residual plots: $y_{i,p}$ (or t_i, or x_i) vs (y_i – $y_{i,p}$) plots that can reveal systematic bias.

Illustrative Dataset

We'll illustrate the basics of linear regression and some common variants using an artificial dataset with more x variables than are required by the task. We'll return to the Dow dataset afterward.

```
# Define generating function
x = np.linspace(-2,2.0,100)
Y = 0.7*x + 0.9*x**(2.0) +0.1*x**(3.0)
Y = Y*0.5*(np.random.rand(len(Y))-0.5) + Y + 0.5

# Make matrix of X features
X = np.ones([len(Y),31])
for i in range(10):
  for j in range(1,4):
    X[:,i*3+j] = x**(j) + \
    (np.random.rand(len(Y))-0.5)*0.01

# Make training and validation sets
inds = list(range(len(Y)))
random.shuffle(inds)
t_inds = inds[:80]
v_inds = inds[-20:]
y_train = Y[t_inds]
y_val   = Y[v_inds]
x_train = X[t_inds]
x_val   = X[v_inds]
```



We've generated a dataset based on $\mathbf{y} = 0.5 + 0.7\mathbf{x} + 0.9\mathbf{x}^2 + 0.1\mathbf{x}^3$

We've also created 27 additional highly correlated **x** variables to confound naïve fitting attempts.

Ordinary Least Squares Regression

The current problem has many redundant variables and so using ordinary least squares regression is a bad idea.

Nevertheless, nothing is stopping us from attempting:

```
# fit the ordinary least squares model
beta = np.linalg.solve(np.dot(x_train.T,x_train),np.dot(x_train.T,y_train))

# Make predictions on training data
p_train = np.dot(x_train,beta)
p_val = np.dot(x_val,beta)
mse = np.mean((p_train - y_train)**(2.0))
r2  = 1 - mse/np.mean((y_train-np.mean(y_train))**(2.0))
```

Recall that we know the analytic solution:

$$\beta = \left(\mathbf{x}^{\mathrm{T}}\mathbf{x}\right)^{-1}\mathbf{x}^{\mathrm{T}}\mathbf{y}$$

The corresponding predictions are:

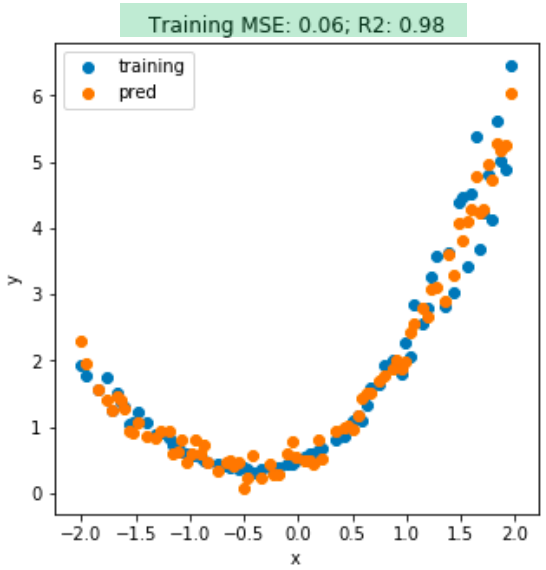
$$\mathbf{x}\beta = \mathbf{y}_p$$

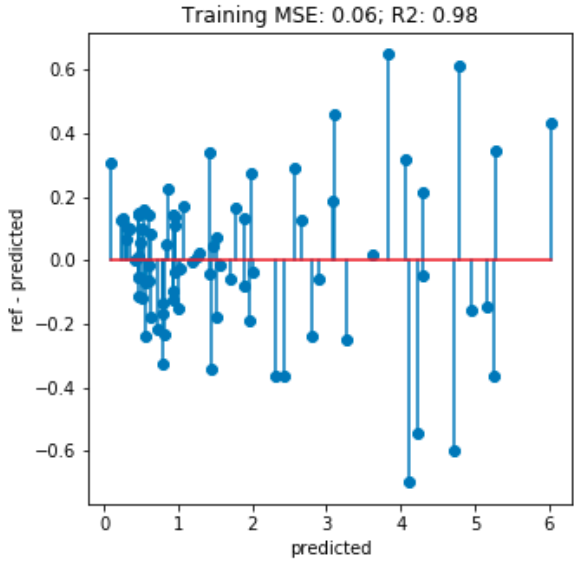
Ordinary Least Squares Regression

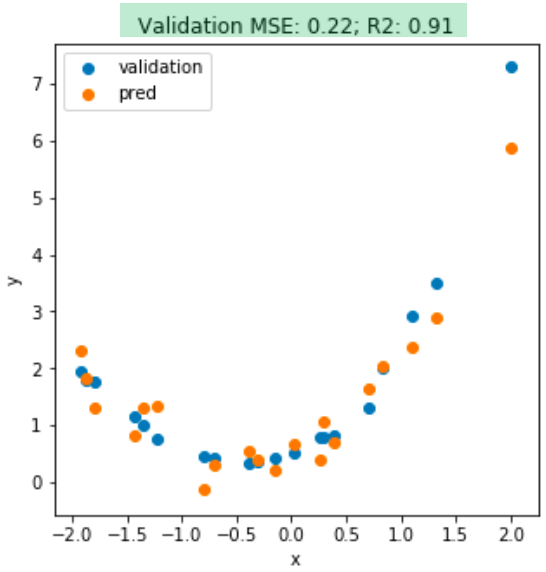
- Even though we are fitting a linear model, the predictions exhibit significant scatter. This is clear visual evidence that we are fitting noise

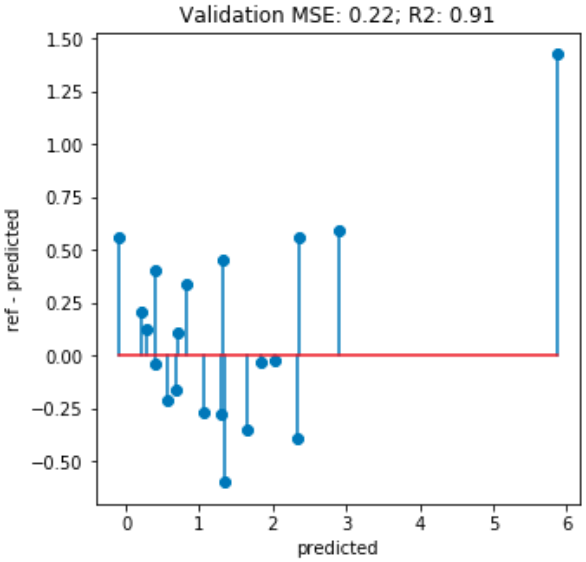
- The MSE and R^2 on training data is significantly lower than on the validation data (a tell-tale sign of overfitting)

- Residual plots show no evidence of bias (expected for least-squares optimization with representative training set)





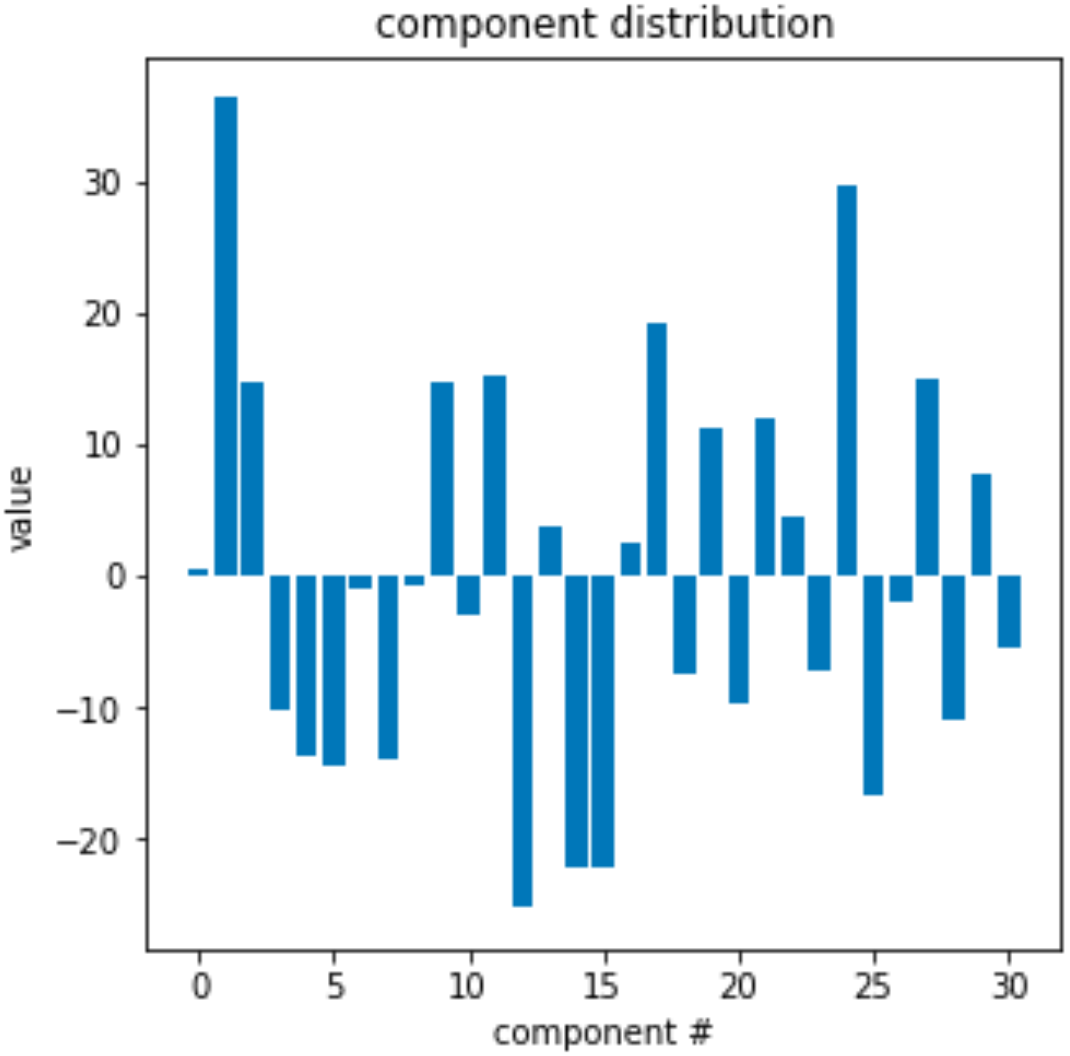




Ordinary Least Squares Regression

• We clearly do not recover the known generating function.

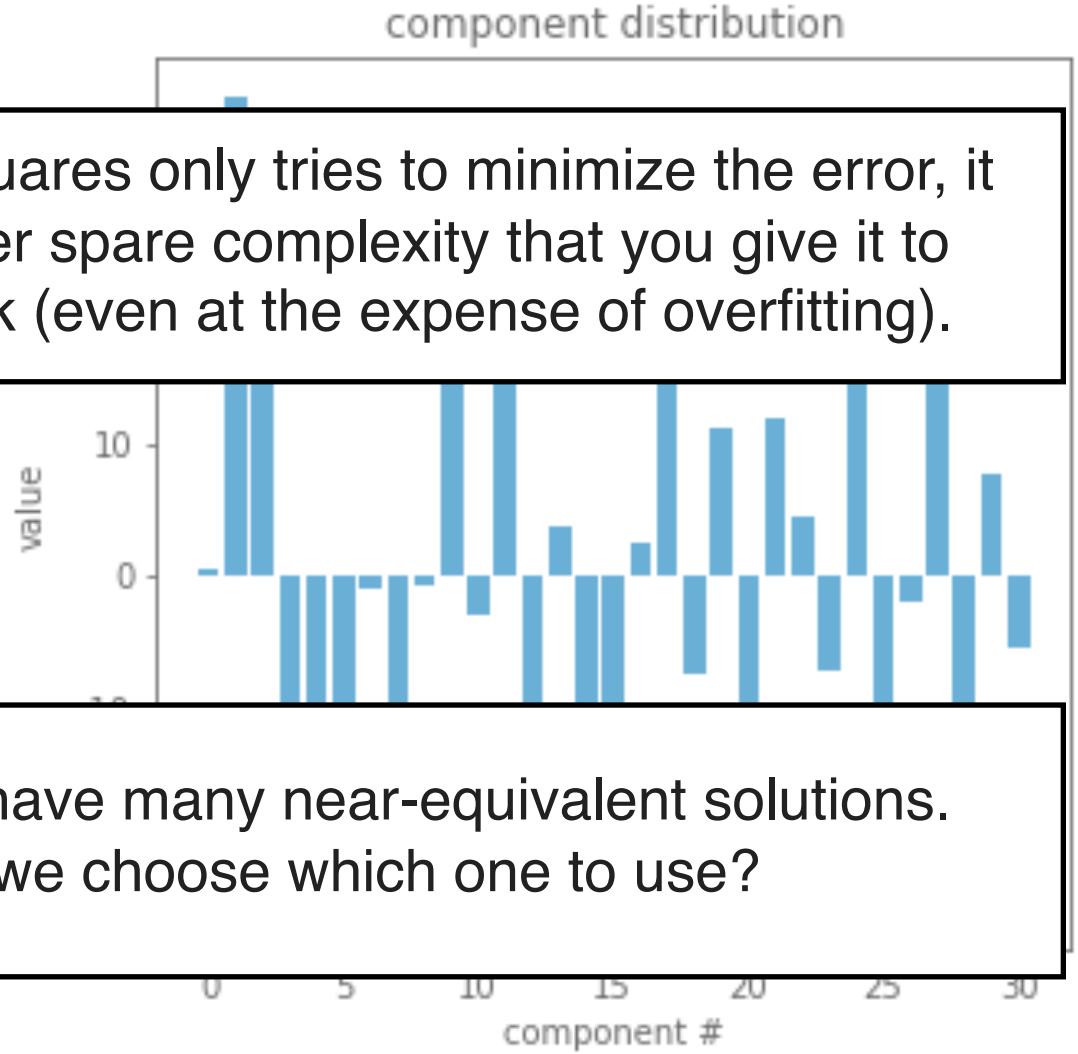
• Worse, we see many large components that seem to finely balance each other (this is how the predictions obtain the noisy appearance).



Plotting the components of the beta vector.

Ordinary Least Squares Regression

Ordinary least squares only tries to minimize the error, it will use whatever spare complexity that you give it to achieve this task (even at the expense of overfitting).



We essentially have many near-equivalent solutions. How do we choose which one to use?

Plotting the components of the beta vector.

Regularization Concept

Regularization is the concept of reducing model complexity to reduce overfitting. In this approach we modify the model and/or objective function in order to penalize complexity.

Among the simplest approaches to semi-automatically simplifying models is through penalization of the L_1 and/or L_2 norm in the objective function:

$$\arg\min_{\beta} \frac{1}{N} \sum_{i} \left(y_i - \mathbf{x}_i \beta\right)^2 + \alpha ||\beta||_p$$

Where α is a parameter and $||\beta||_p$ is the L_p norm: $||\beta||_p = \left(\sum_{j}^{N_\beta} |\beta_j|^p \right)^{\frac{1}{p}}$

p=1 for L_1 regularization and p=2 for L_2 regularization (also known as **Lasso*** and **Ridge** regularization, respectively)

*for lasso an additional division by N_β is typically made in the objective function

Regularization Concept

$$\arg\min_{\beta} \frac{1}{N} \sum_{i} \left(y_i - \mathbf{x}_i \beta\right)^2 + \alpha ||\beta||_p$$

- From a mechanistic perspective L_1 and L_2 penalize large coefficient vectors.

- Theoretical justifications can be made for these regularizations, but they are more commonly employed as a heuristic, with the value of alpha being determined by performance on validation data.

- All else being equal, L_1 tends to more aggressively zero out parameters, whereas L_2 favors small but relatively more distributed values.

- Analytic solutions exist for Linear Lasso regression and linear Ridge regression, but we won't implement them ourselves. Instead we'll use the implementations provided by **scikit learn**.

Scikit Learn

Scikit learn is an open-source python library of non-deep learning ML models.

First, let's illustrate how to use scikit learn in the context of linear regression (a problem I've already showed you how to implement yourself).

```
from sklearn import linear_model # contains all of the linear regression models
from sklearn.metrics import mean_squared_error, r2_score # error metrics
lin = linear_model.LinearRegression(fit_intercept=False) # call the constructor
lin.fit(x_train,y_train)
```

• The basic syntax of scikit learn is to call the constructor of the model you want to fit (just with the relevant parameters specified, no data).

• After you have initialized an instance of the model, you will train it using the `model.fit(data)` method, and access various quantities associated with the model using other methods:

```
# scikit learn results
print("model coefficients: {}".format(lin.coef_))
print("method result: {}".format(lin.predict(x_train)))
print("mse: {}".format(mean_squared_error(y_train,lin.predict(x_train))))
print("r2: {}".format(r2_score(y_train,lin.predict(x_train))))
```

Lasso/L_1 Regularization

To perform Lasso regularization in scikit learn, the relevant model is `Lasso`

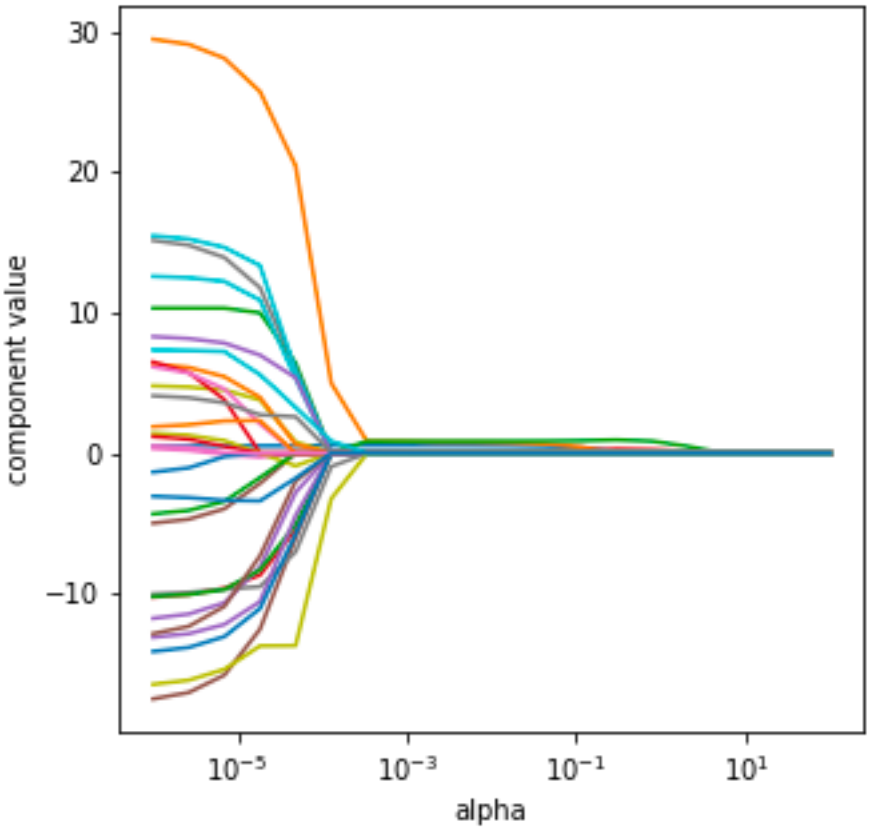
Since *a priori* we don't know what regularization parameter to use, we can use results on our validation data to choose this parameter:

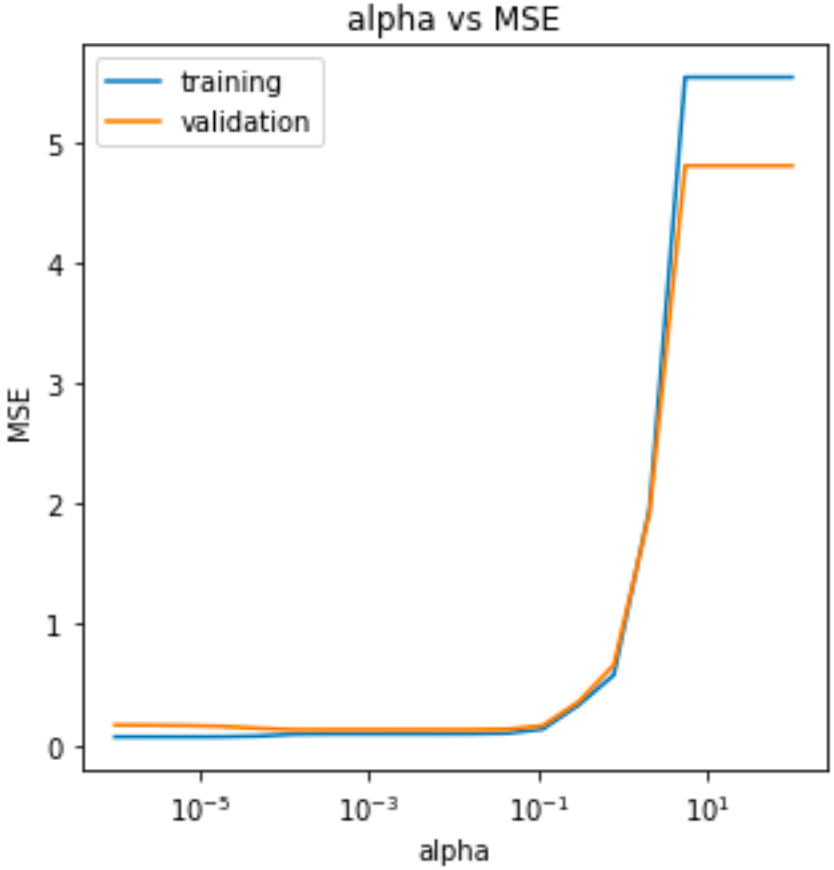
```
alphas = np.logspace(-6,2,20)
betas  = []
t_mses = []
v_mses = []
for i in alphas:
  lin = linear_model.Lasso(alpha=i,max_iter=1000000,tol=0.001,fit_intercept=False)
  lin.fit(x_train,y_train)
  betas  += [lin.coef_]
  t_mses += [mean_squared_error(y_train,lin.predict(x_train))]
  v_mses += [mean_squared_error(y_val,lin.predict(x_val))]
```

• Here I have performed a "hyperparameter" search over potential alpha values, while retaining the training and prediction MSE for each fit.

• The LassoCV model provides similar functionality when using cross-validation rather than an explicit validation set.

Lasso/L_1 Regularization





As $\alpha \rightarrow 0$ we recover ordinary least squares. For larger α, the components in beta decrease.

We incur almost no penalty in our training MSE until $\alpha > 0.1$. For $\alpha \sim 0.01$ the validation MSE is minimized (we can afford to simplify our model).

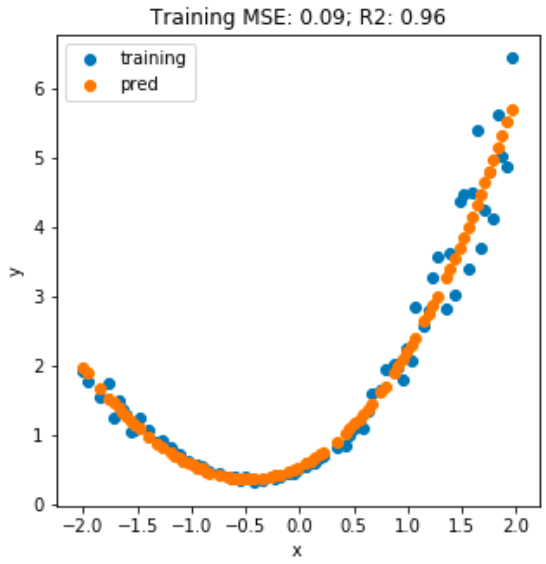
Lasso/L_1 Regularization

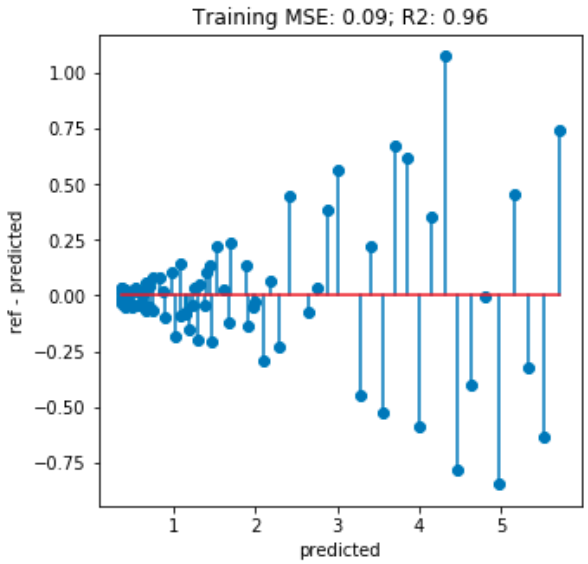
We have chosen the best alpha based on performance on the validation set and plotted its predictions.

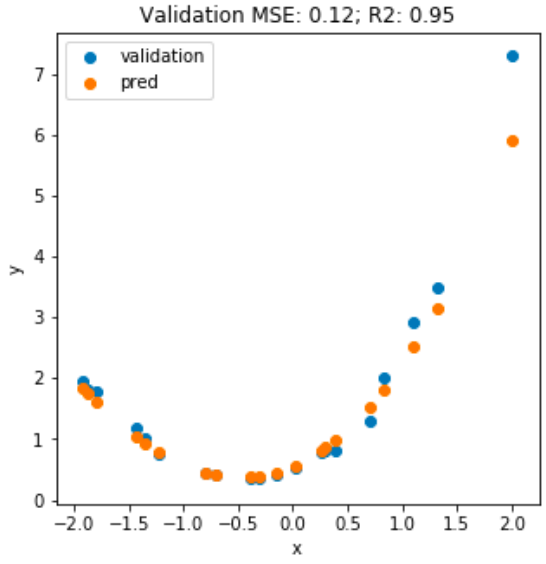
The training and validation MSE and R^2 are very similar

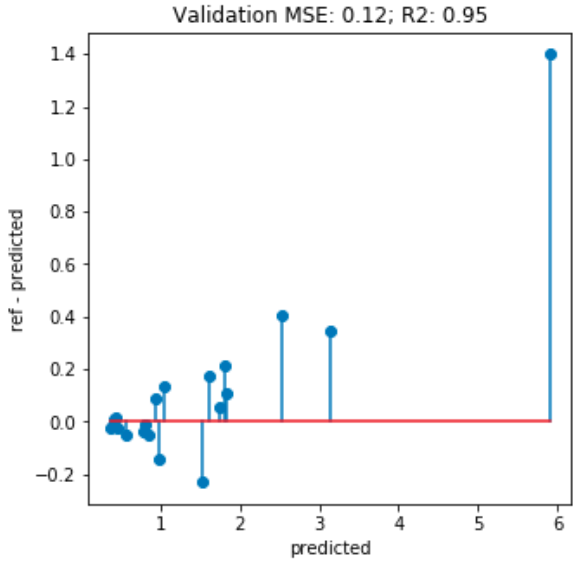
The noise is nearly gone, we have fit the "signal"

The residual plots still exhibit no systematic bias, despite the regularization constraint (expected)









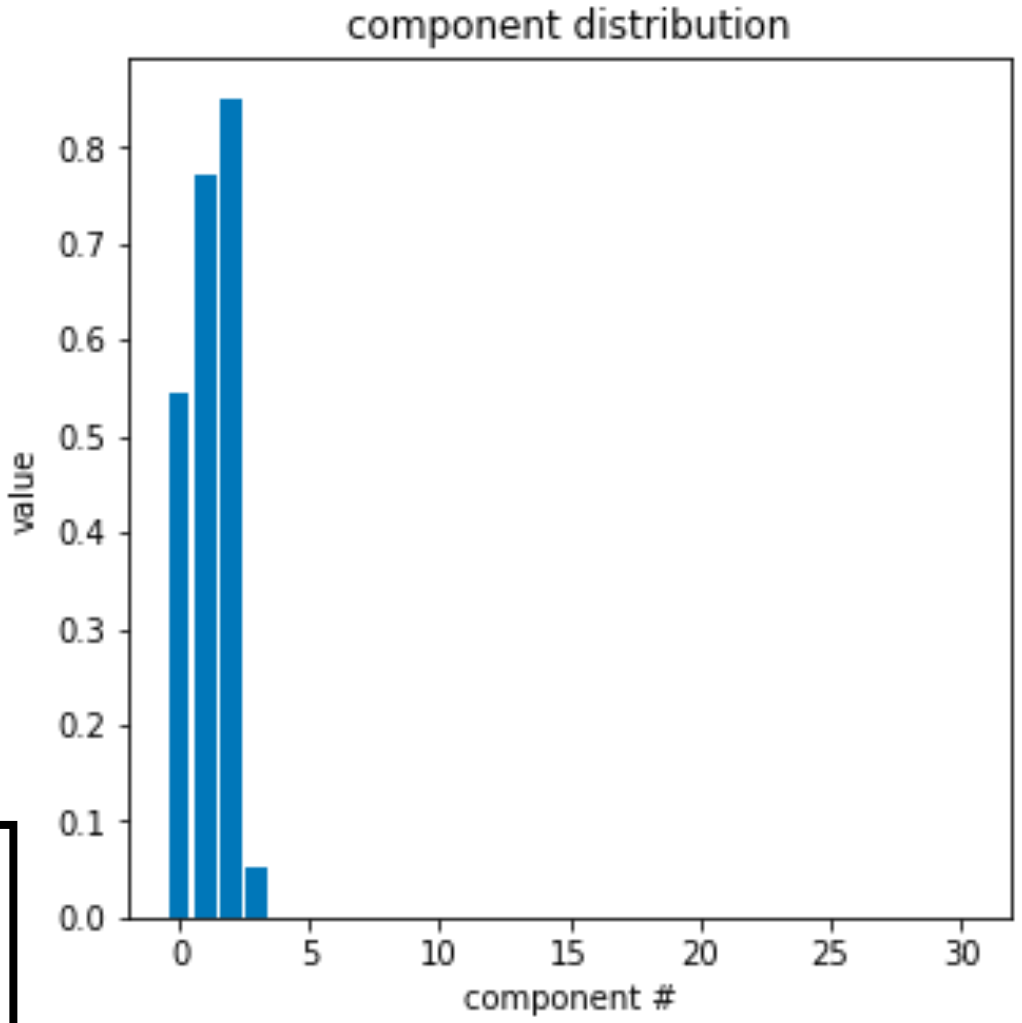
Lasso/L_1 Regularization

- Lasso with hyperparameter optimization has automatically determined that only four non-zero components are required to fit the underlying data.

- Lasso is also displaying its preference for zeroing out components rather than distributing values across correlated variables.

Recall that the generating function was:

$$\mathbf{y} = 0.5 + 0.7\mathbf{x} + 0.9\mathbf{x}^2 + 0.1\mathbf{x}^3$$



Plotting the components of the beta vector.

Ridge/L_2 Regularization

To perform Ridge regularization in scikit learn, the relevant model is `Ridge`

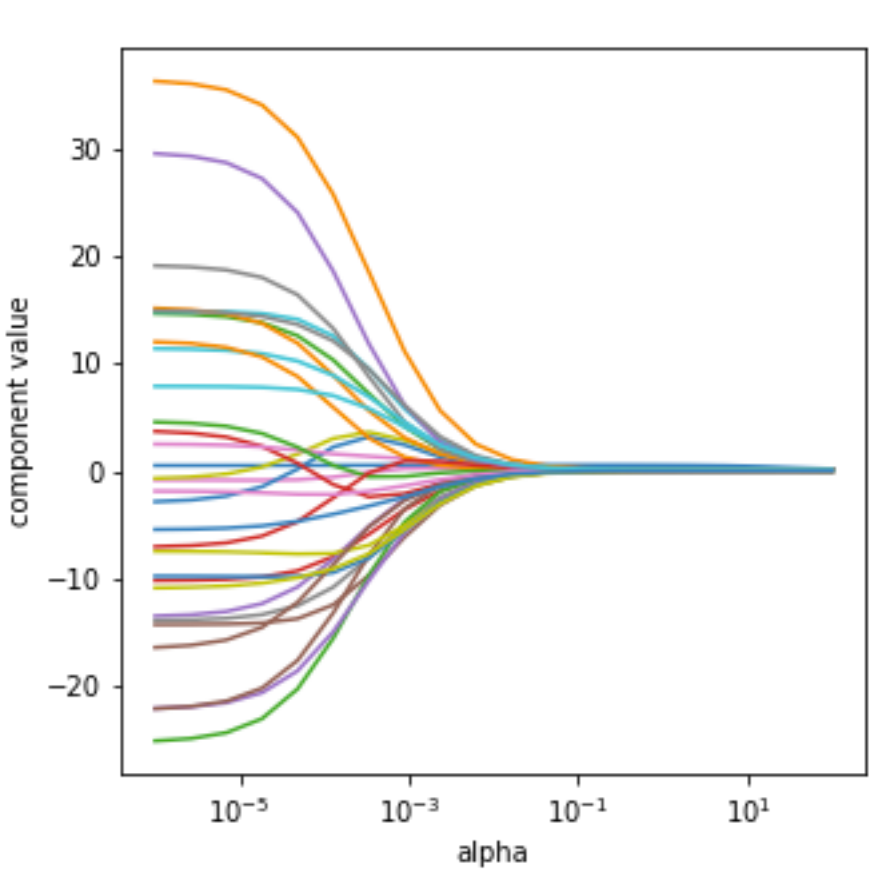
Since *a priori* we don't know what regularization parameter to use, we can use results on our validation data to choose this parameter (same as before):

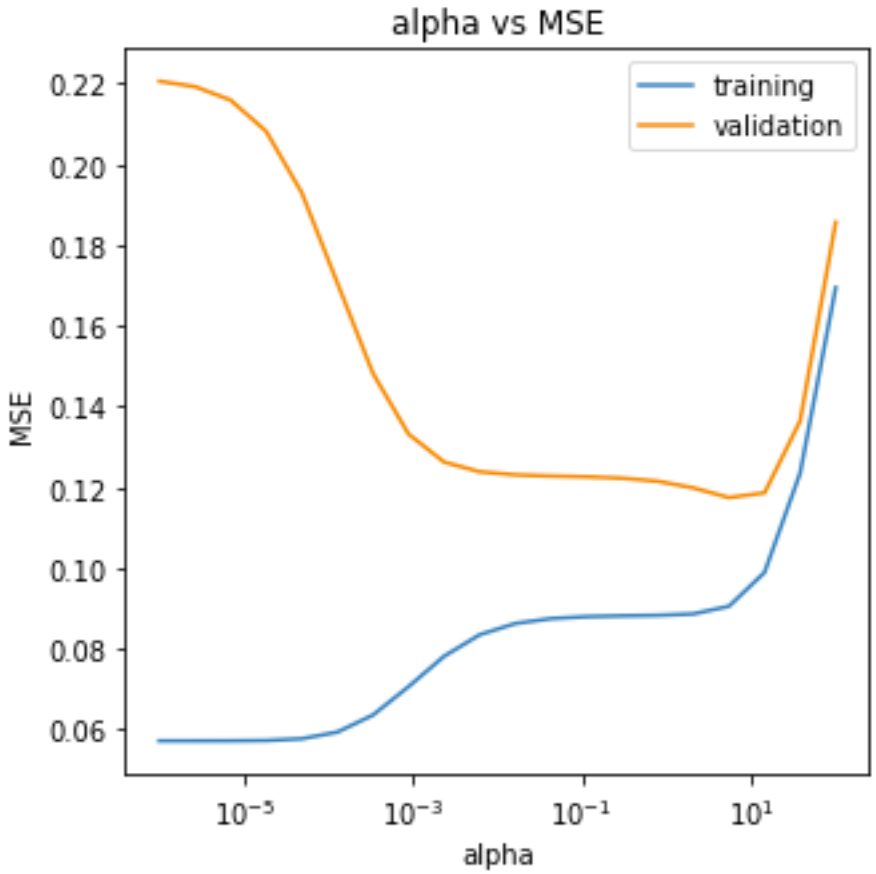
```
alphas = np.logspace(-6,2,20)
betas  = []
t_mses = []
v_mses = []
for i in alphas:
  lin = linear_model.Ridge(alpha=i,fit_intercept=False)
  lin.fit(x_train,y_train)
  betas  += [lin.coef_]
  t_mses += [mean_squared_error(y_train,lin.predict(x_train))]
  v_mses += [mean_squared_error(y_val,lin.predict(x_val))]
```

• Here I have performed a "hyperparameter" search over potential alpha values, while retaining the training and prediction MSE for each fit.

• The RidgeCV model provides similar functionality when using cross-validation rather than an explicit validation set.

Ridge/L_2 Regularization





As alpha → 0 we recover ordinary least squares. For larger alpha values, the components in beta decrease.

We incur almost no penalty in our training MSE until $\alpha > 10^{-4}$. For $\alpha \sim 10$ the validation MSE is minimized (**very different from initial turn in training MSE**).

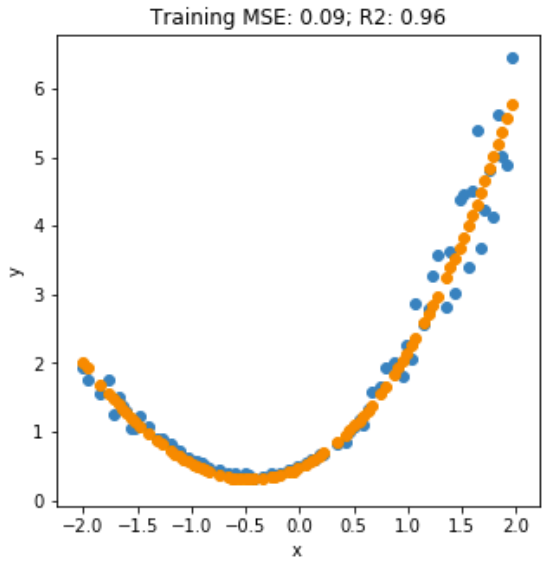
Ridge/L_2 Regularization

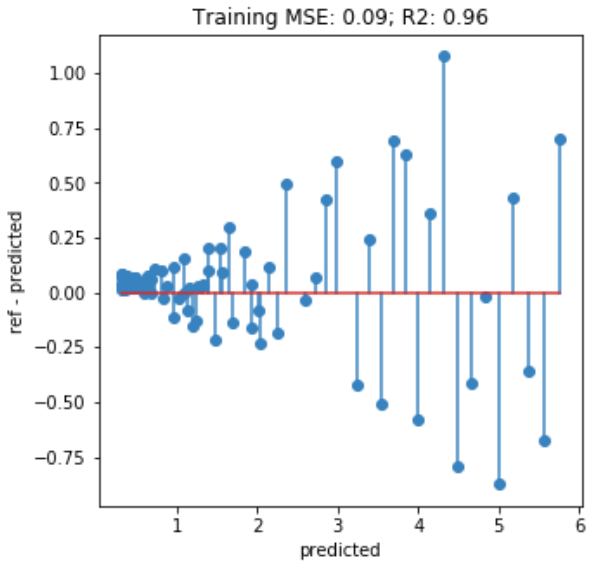
We have chosen the best alpha based on performance on the validation set and plotted its predictions.

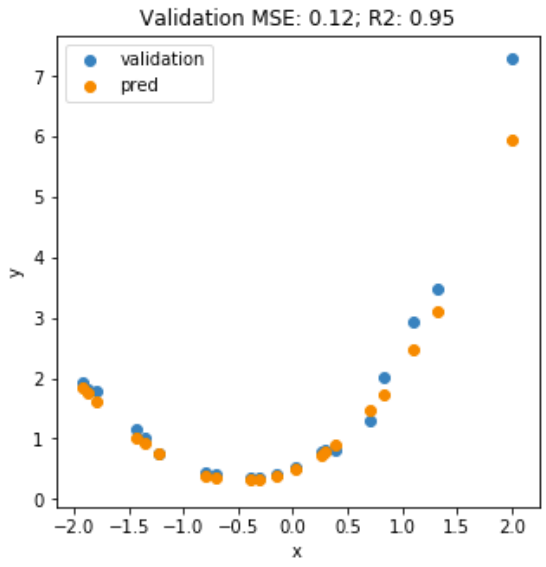
The training and validation MSE and R^2 are very similar

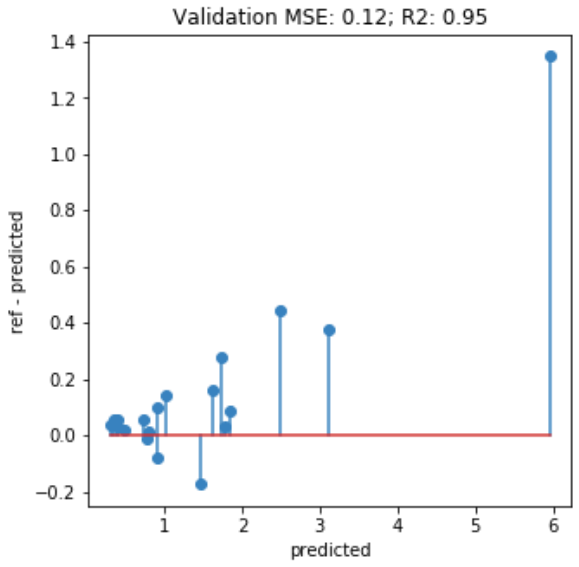
The noise is nearly gone, we have fit the "signal"

The residual plots still exhibit no systematic bias, despite the regularization constraint (expected)









Ridge/L_2 Regularization

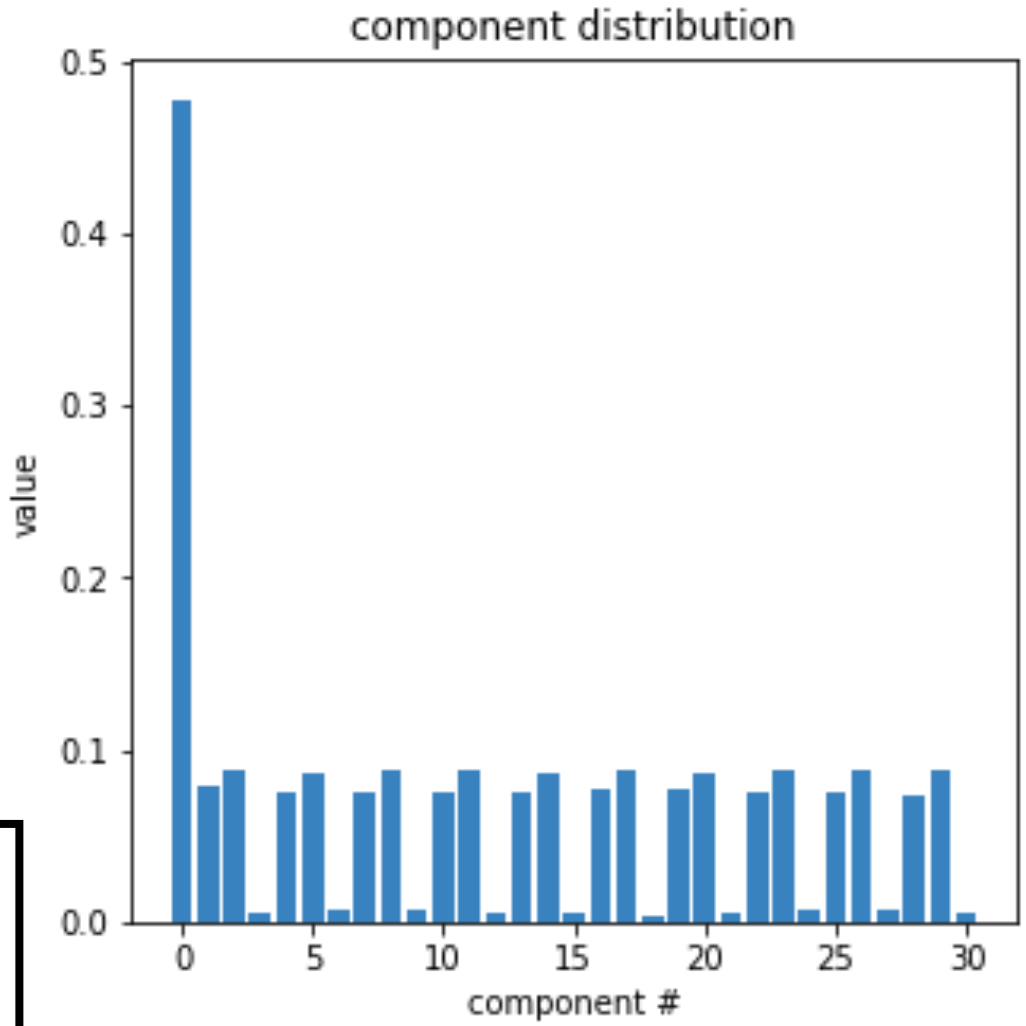
- Ridge favors more variables with smaller overall coefficients.

- The algorithm recognizes that that there are redundant variables, and puts 1/N on each of them (as opposed to lasso which consolidates).

- The intercept variable is distinct from the rest and close to the 0.5 value in the generating function.

Recall that the generating function was:

$$\mathbf{y} = 0.5 + 0.7\mathbf{x} + 0.9\mathbf{x}^2 + 0.1\mathbf{x}^3$$



Plotting the components of the beta vector.

Partial Least Squares

Partial least squares (PLS) linear regression is commonly used when you have many predictor variables (**X**) relative to independent variables (**y**)

- In PLS we seek the ordered components in **X** that maximize the variance in **y**

- In PCA we seek the ordered components in **X** that maximize the variance in **x**.

- The evaluation of the components for PLS isn't analytic like PCA (which was just the eigendecomposition of the centered covariance matrix). For the most common algorithm* :

The first component is simply the normalized covariance of **X** and **y** ($\mathbf{X}^T\mathbf{y}/|\mathbf{X}^T\mathbf{y}|$), you then subtract the projection of this component on **X** from **X** and repeat to gather the other components (or terminate once you get a null result [happens with collinear columns]).

There is a little more subtlety but this gives you the essential idea.

PLS in Scikit Learn

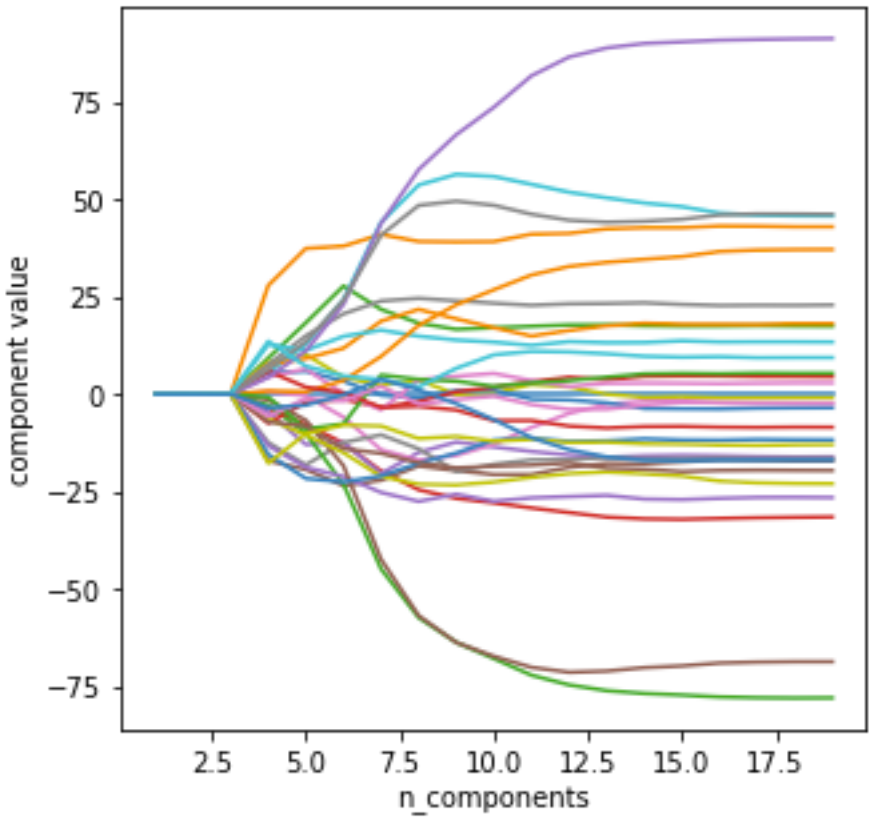
To perform PLS linear regression in scikit learn, the relevant model is `PLSRegression` within the `cross_decomposition` submodule.

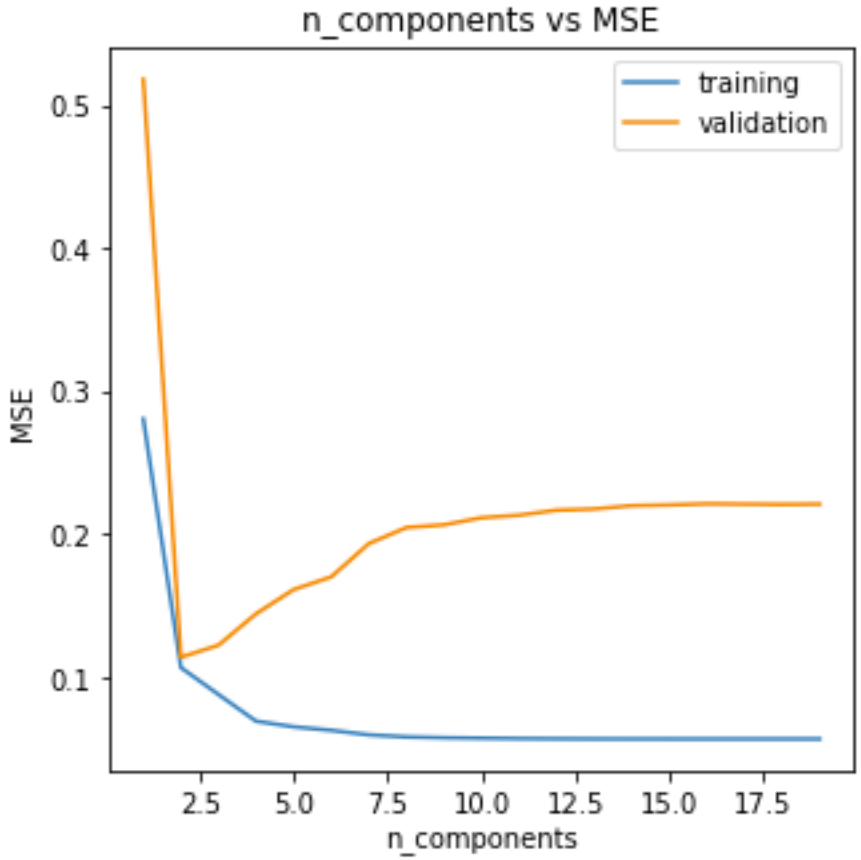
Since *a priori* we don't know how many components to use in the regression (akin to choosing alpha), we can use results on our validation data to choose this parameter (same as before):

```
# Do a hyperparameter optimization on n_components
n_comp = range(1,20)
betas  = []
t_mses = []
v_mses = []
for i in n_comp:
  lin = PLSRegression(n_components=i) # call the constructor
  lin.fit(x_train,y_train)
  betas  += [lin.coef_]
  t_mses += [mean_squared_error(y_train,lin.predict(x_train))]
  v_mses += [mean_squared_error(y_val,lin.predict(x_val))]
```

- I have performed a "hyperparameter" search over potential `n_components` values, while retaining the training and prediction MSE for each fit.

PLS Linear Regression





As `n_components`→N_x we recover ordinary least squares. For small numbers of components we alpha values, the components in beta decrease.

The difference between 2 vs 3 components is small and we have a small validation set. Experimentation is desirable (and inexpensive).

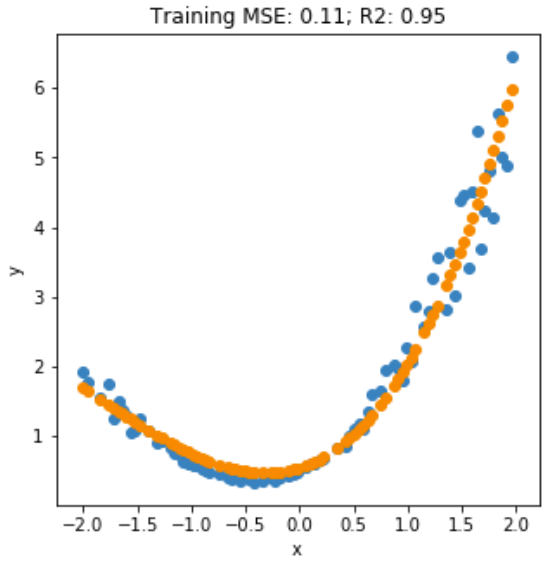
PLS Linear Regression

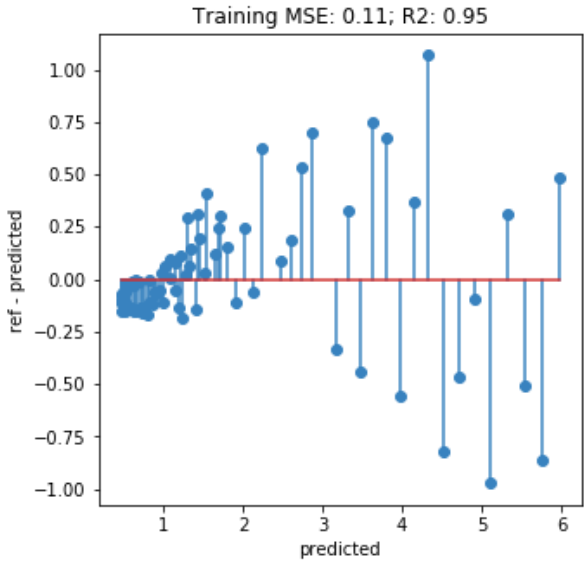
I'm show the result for regression with the first two components (best on validation)

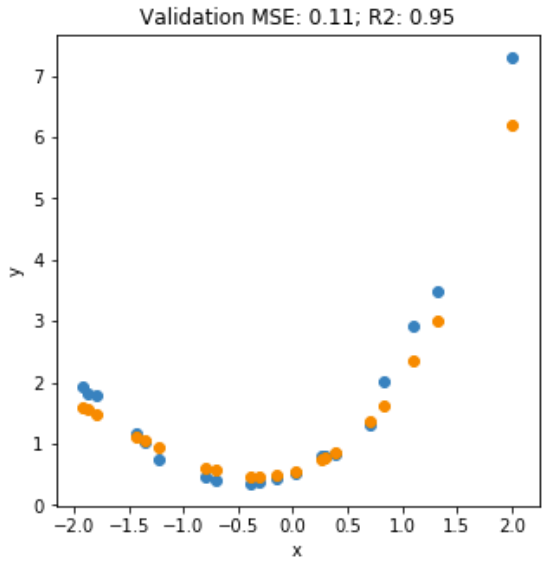
The training and validation MSE and R^2 are identical (similar to regularization results)

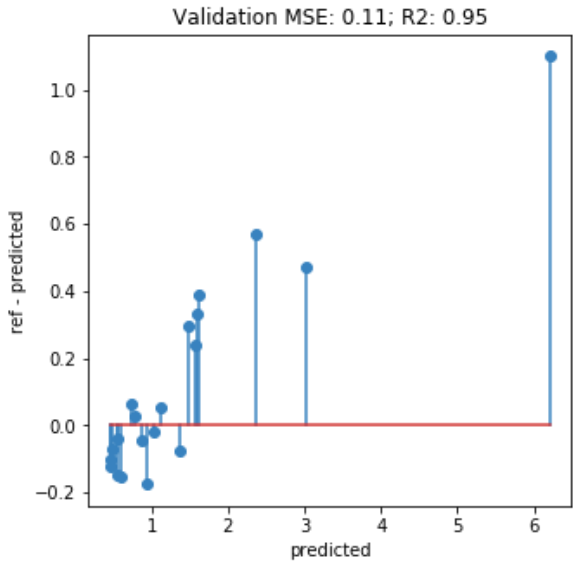
With only two components, a systematic bias is visible

The residual plots clearly reveal the systematic bias (errors are no long symmetrically distributed).









PLS Linear Regression

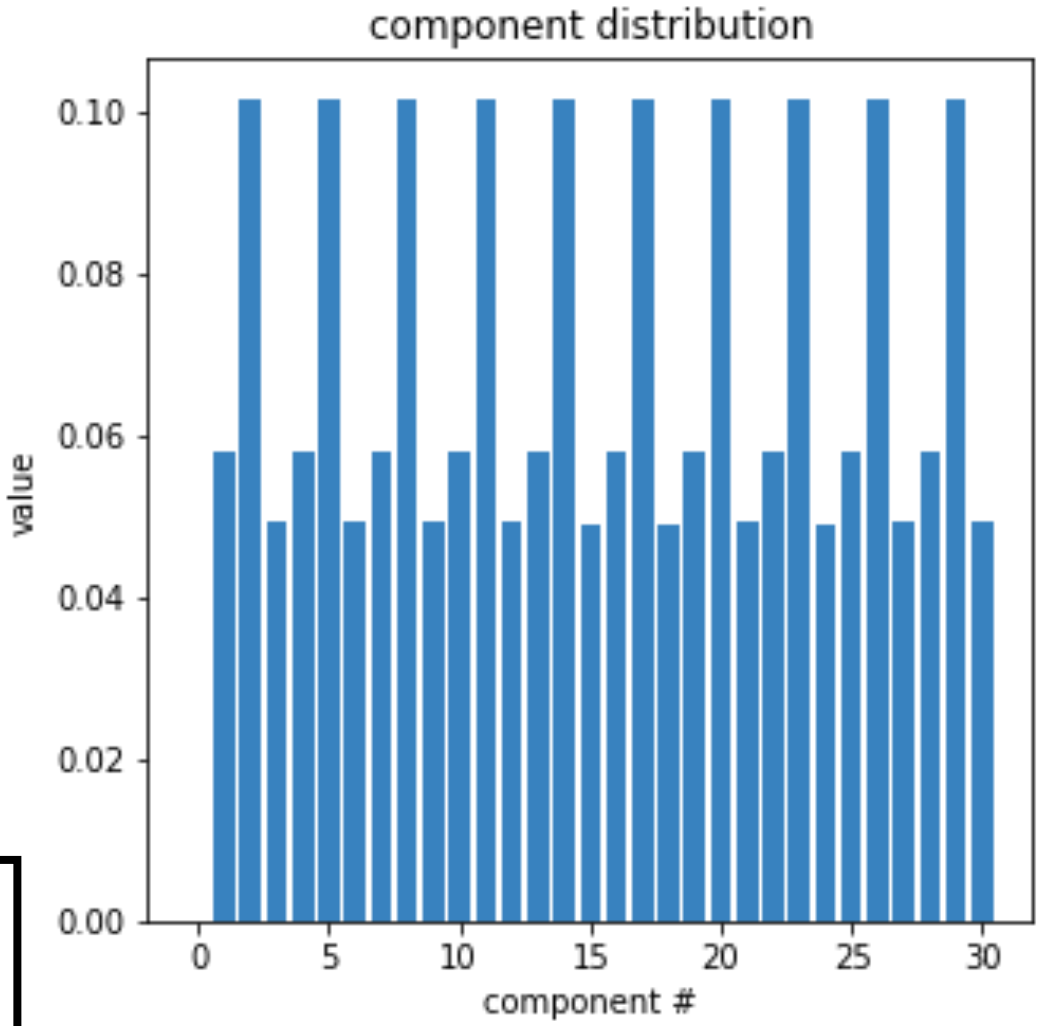
• PLS (with two components) is very similar to ridge.

***Note: data is internally shifted during PLS so intercept is missing**

• The algorithm recognizes that that there are redundant variables, and puts 1/N on each of them (as opposed to lasso which consolidates).

Recall that the generating function was:

$$\mathbf{y} = 0.5 + 0.7\mathbf{x} + 0.9\mathbf{x}^2 + 0.1\mathbf{x}^3$$



Plotting the components of the beta vector.

Dow Dataset – Linear Regression

Let's now explore how our several flavors of linear regression perform on the Dow dataset that we have been working with.

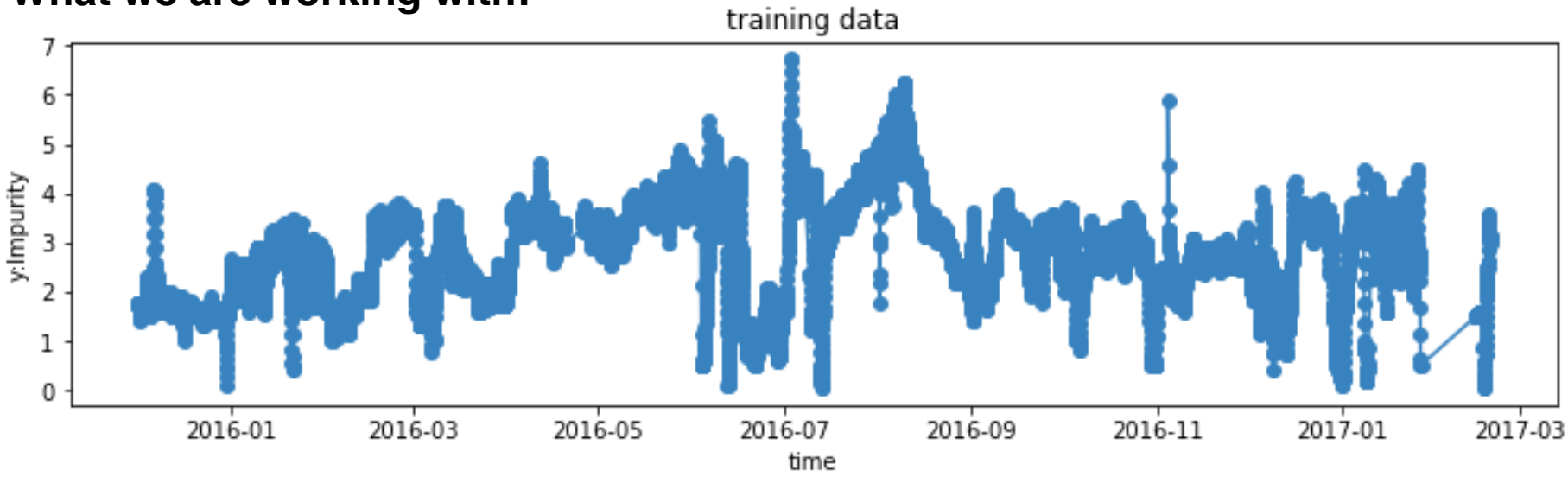
- We'll use the cleaned data that we developed previously as a training set, and data from following year as a validation set.
- We'll set our goal to be predicting impurity fluctuations up to 7%, under the assumption that this would trigger an intervention.

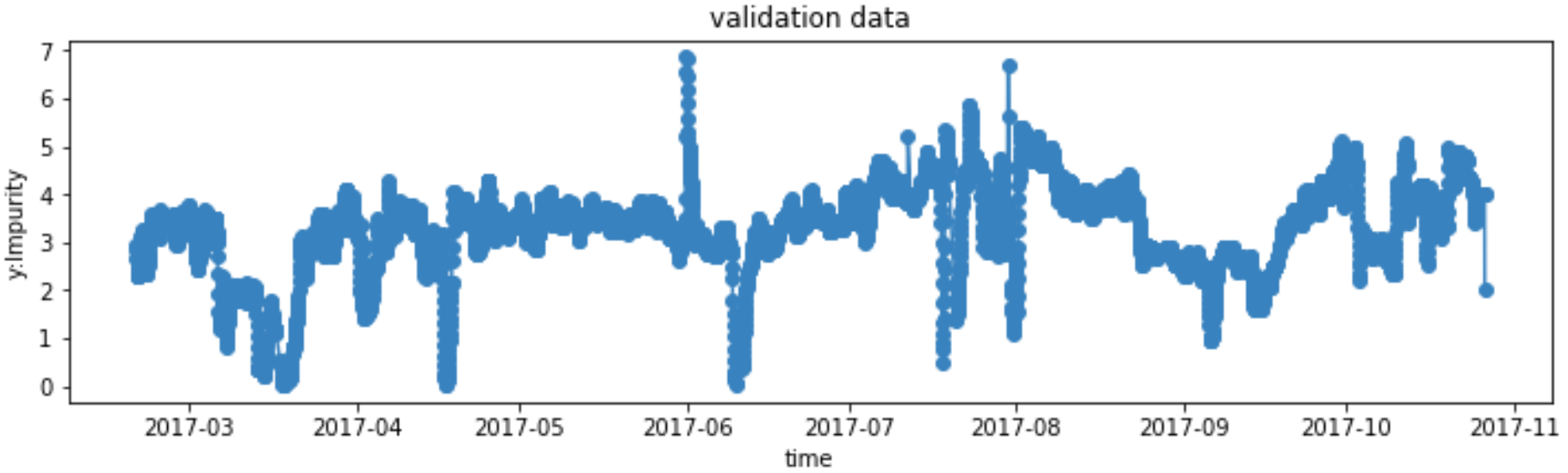
Load the data:

```
train = pd.read_excel("training_clean.xlsx")
val = pd.read_excel("validation_clean.xlsx")

# We will only focus on predicting impurity levels up to a threshold
# of 7, under the assumption that this would trigger intervention.
train = train[(train["y:Impurity"]>0) & (train["y:Impurity"]<7) ]
val = val[(val["y:Impurity"]>0) & (val["y:Impurity"]<7)]

# We'll also include the variables related to the input feed and outlet impurity
# as training variables. The outlet impurity is distinct from the y:impurity variable
xcols = list(val.columns[1:43]) + list(val.columns[-2:])

# Define the training and validation X and Y arrays
train_X = train.loc[:,xcols].to_numpy()
train_Y = train.iloc[:,-3].to_numpy()
val_X  = val.loc[:,xcols].to_numpy()
val_Y  = val.iloc[:,-3].to_numpy()
```

Dow Dataset – Linear Regression

What we are working with:





Dow Dataset – Ordinary Least Squares

As before, we'll use ordinary least squares as our starting point for comparison with the regularized and PLS methods.

Training is merely two lines of code:

```
lin = linear_model.LinearRegression() # call the constructor
lin.fit(train_X,train_Y) # train the model
```

Using our trained model to predict y for arbitrary x-values is only one line:

```
y_train = lin.predict(train_X) # model predictions on the training data
y_val = lin.predict(val_X) # model predictions on the validation data
```

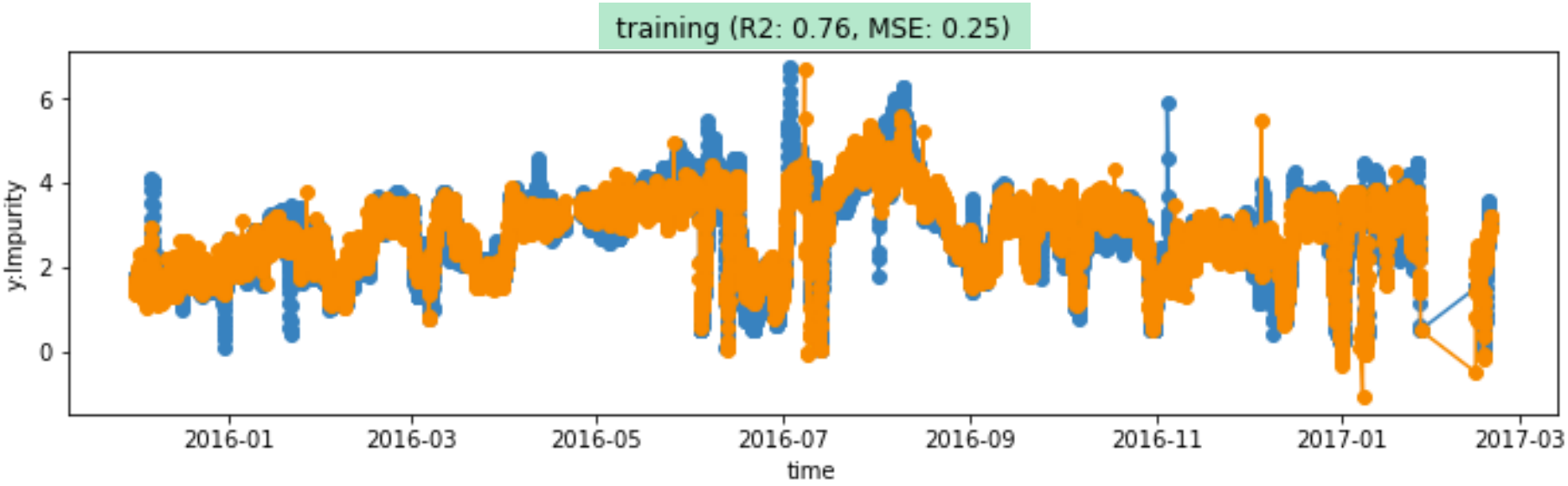
Calculating MSE and R^2 are likewise one line commands:

```
r2 = r2_score(train.iloc[:,-3],lin.predict(train_X)) # r2 metric
mse = mean_squared_error(train.iloc[:,-3],lin.predict(train_X)) # mse metric
```

These commands are the same for any of the scikit learn models, with the exception of the very first model initialization call.

I won't show them or the plotting commands in the rest of the presentation, but you can see the notebook for reference on how the figures were generated. c

Dow Dataset – Ordinary Least Squares

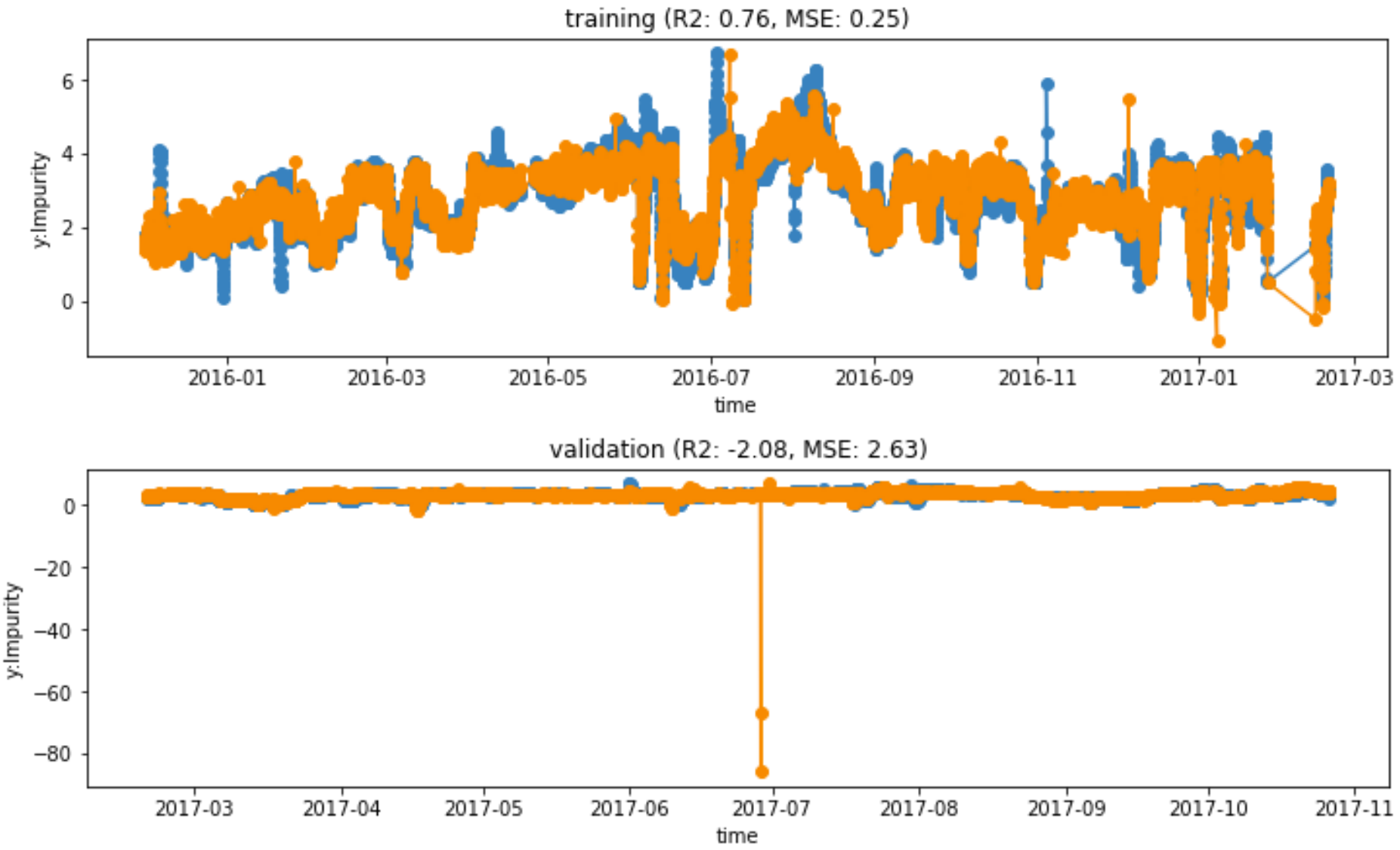


Q: Can we do better than OLS on the training data?

A: Not with a linear model. OLS is the best any linear model can do on the training data, because minimizing MSE is its only objective.

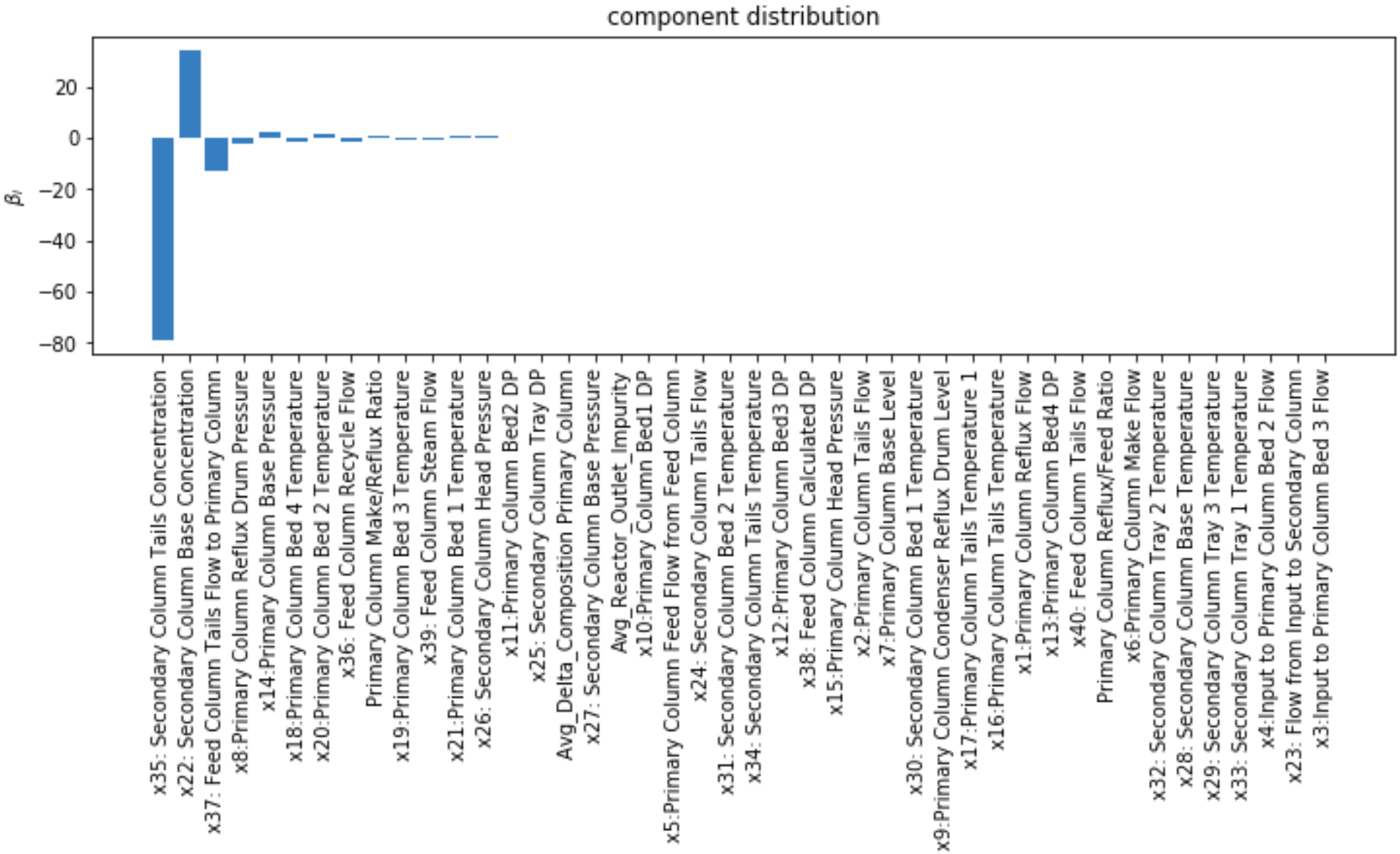
All of the other methods must necessarily have an MSE bounded from below by the OLS value. Minimizing the difference between the validation and training MSE is where they show superiority.

Dow Dataset – Ordinary Least Squares



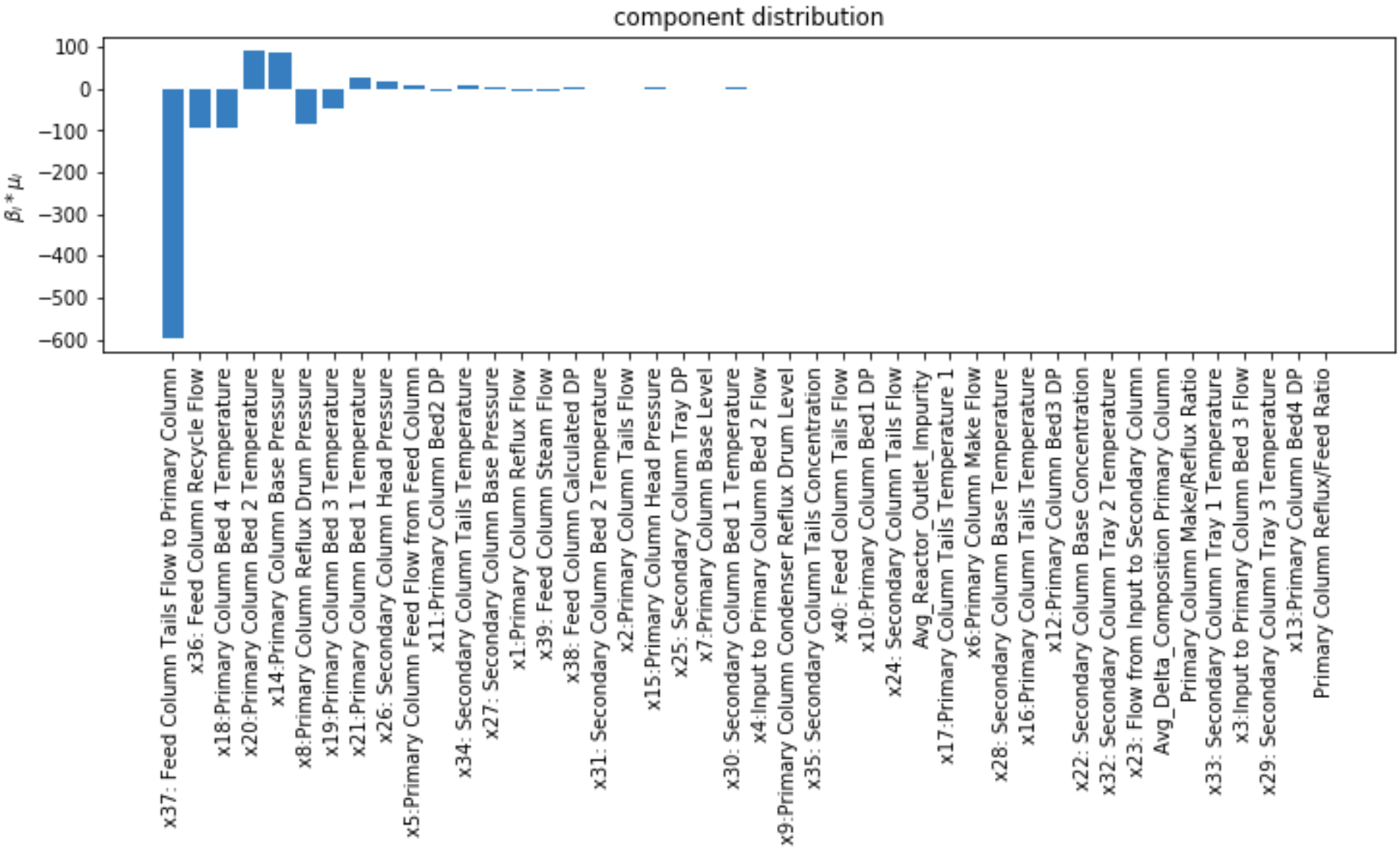
Disaster! The model is clearly overfit to the training data.

Dow Dataset – Ordinary Least Squares



Small number of variables with large contributions. A minor fluctuation in a sensor can easily throw our prediction WAY off.

Dow Dataset – Ordinary Least Squares



Multiplying the coefficients by the mean of each variable can give complementary information about the relative contribution of each variable to the prediction (e.g., units could affect the scaling of components).

Dow Dataset – Lasso Regression

With Lasso/L_1 regularization we have a "hyperparameter" α that we need to choose. How do we do this?

We will use our dedicated validation data (LassoCV can be used for cross-validation in circumstances without a dedicated validation dataset):

```
# Do a hyperparameter optimization on alpha
alphas = np.logspace(-6,2,20)
mse_train = []
mse_valid = []
r2_train  = []
r2_valid  = []
alphas = np.logspace(-6, 6, 13)
for i in alphas:
  lin = linear_model.Lasso(alpha=i,max_iter=1000,tol=0.001) # call the constructor
  lin.fit(train_X,train_Y) # train the model
  mse_train += [mean_squared_error(train.iloc[:,-3],lin.predict(train_X))]
  mse_valid += [mean_squared_error(val.iloc[:,-3],lin.predict(val_X))]
  r2_train += [r2_score(train.iloc[:,-3],lin.predict(train_X))]
  r2_valid += [r2_score(val.iloc[:,-3],lin.predict(val_X))]

# Fit on best
print("best alpha: {}".format(alphas[np.argmin(mse_valid)]))
lin = linear_model.Lasso(alpha=alphas[np.argmin(mse_valid)],max_iter=1000)
lin.fit(train_X,train_Y)
```

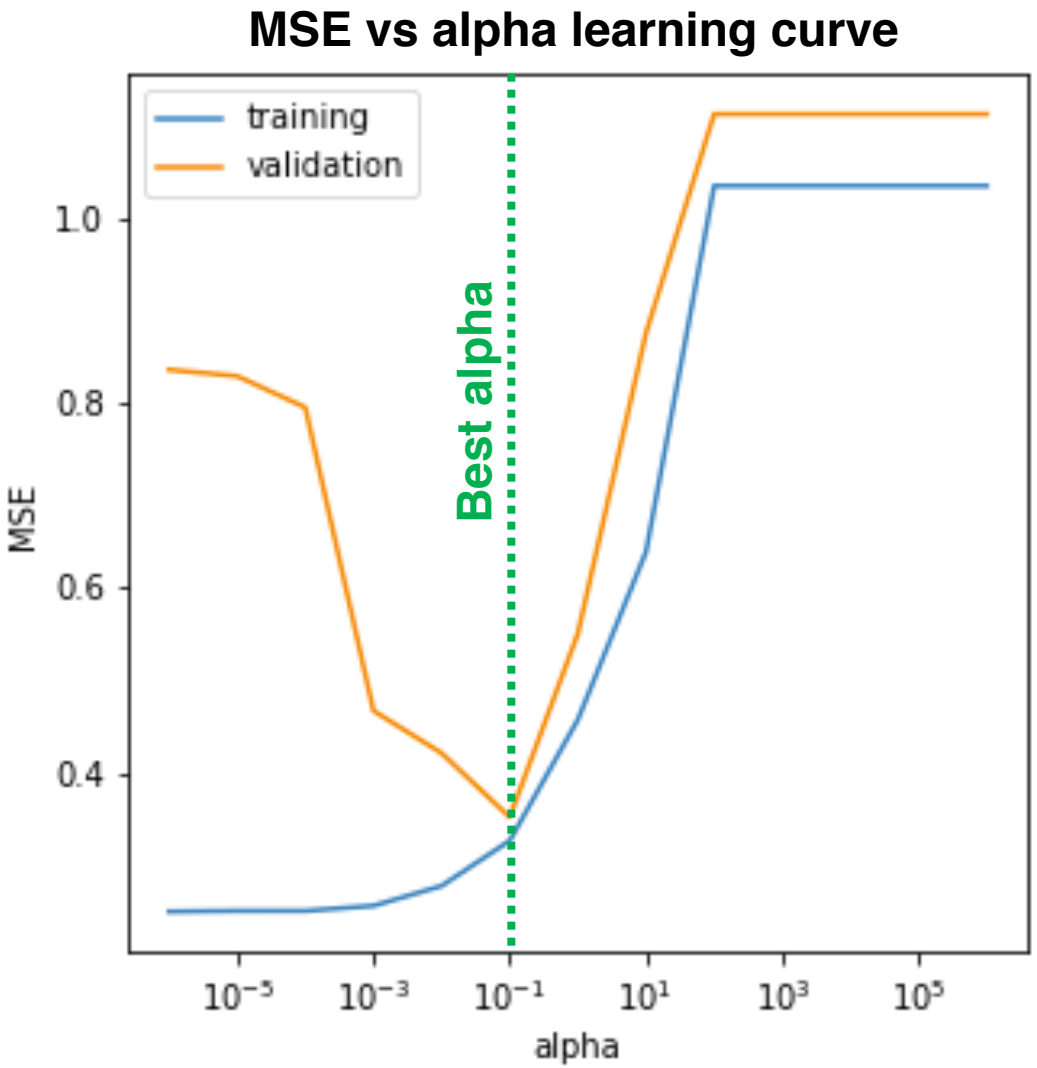
As we move to more complex models, hyperparameter optimization becomes onerous, but linear regression models are very cheap to train.

Dow Dataset – Lasso Regression

This is a typical learning curve for models with many collinear variables: **the training MSE gets worse with increasing alpha, but the validation MSE exhibits a minimum.**

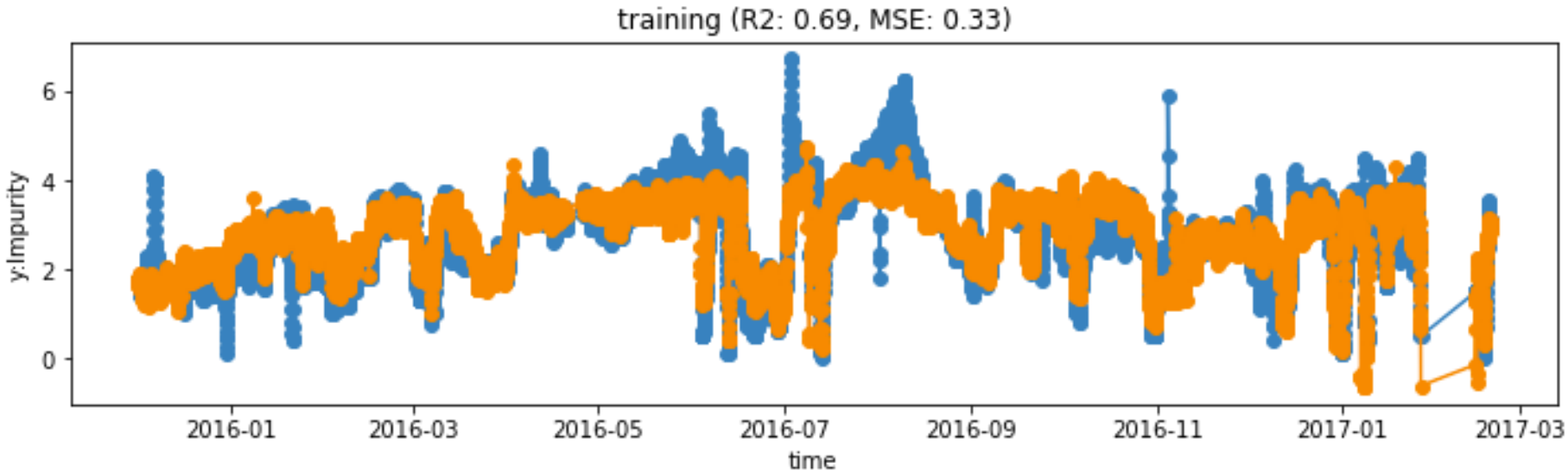
After a certain point, alpha starts impairing our model's ability to learn the relationship between **X** and **y**.

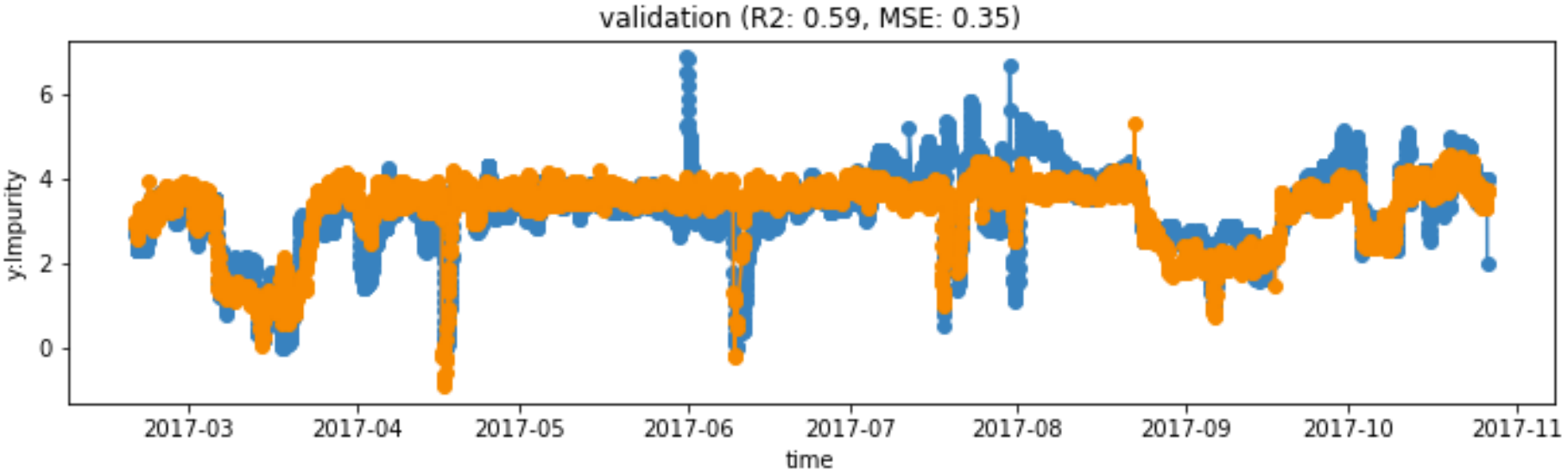
We could have used a finer alpha spacing if we wanted to get an even better model



*corresponding R2 plot has identical behavior

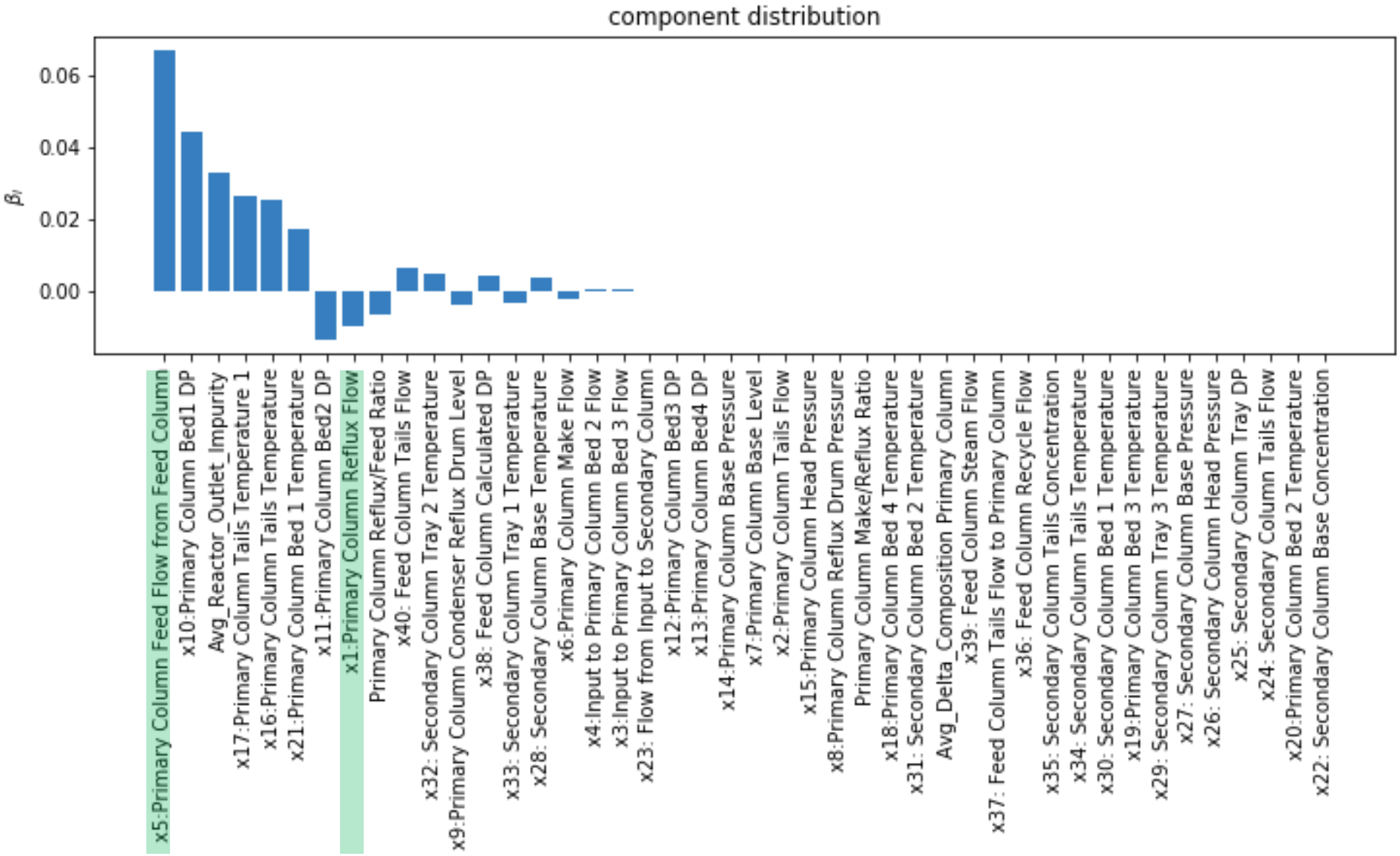
Dow Dataset – Lasso Regression





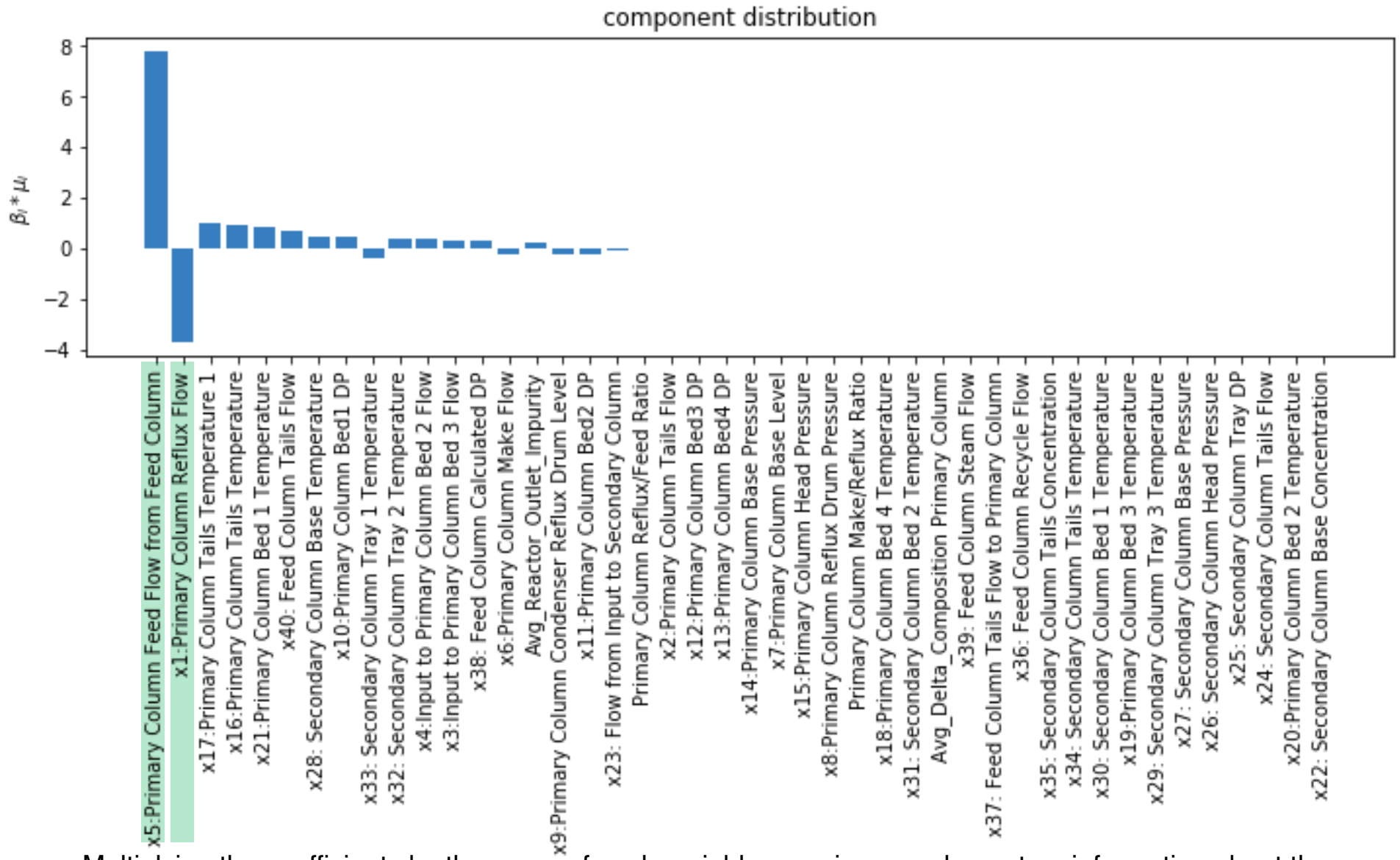
The behavior on the validation dataset is MUCH better than for OLS.

Dow Dataset – Lasso Regression



Overall coefficients are much smaller than for OLS. More variables are utilized, but still less than half of those available.

Dow Dataset – Lasso Regression



Multiplying the coefficients by the mean of each variable can give complementary information about the relative contribution of each variable to the prediction (e.g., units could affect the scaling of components).

Dow Dataset – Ridge Regression

With Ridge/L_2 regularization we have a "hyperparameter" α that we need to choose. How do we do this?

We will use our dedicated validation data (RidgeCV can be used for cross-validation in circumstances without a dedicated validation dataset):

```
# Do a hyperparameter optimization on alpha
alphas = np.logspace(-6,2,20)
mse_train = []
mse_valid = []
r2_train  = []
r2_valid  = []
alphas = np.logspace(-6, 6, 13)
for i in alphas:
  lin = linear_model.Ridge(alpha=i) # call the constructor
  lin.fit(train_X,train_Y) # train the model
  mse_train += [mean_squared_error(train.iloc[:,-3],lin.predict(train_X))]
  mse_valid += [mean_squared_error(val.iloc[:,-3],lin.predict(val_X))]
  r2_train += [r2_score(train.iloc[:,-3],lin.predict(train_X))]
  r2_valid += [r2_score(val.iloc[:,-3],lin.predict(val_X))]

# Fit on best
print("best alpha: {}".format(alphas[np.argmin(mse_valid)]))
lin = linear_model.Ridge(alpha=alphas[np.argmin(mse_valid)])
lin.fit(train_X,train_Y)
```

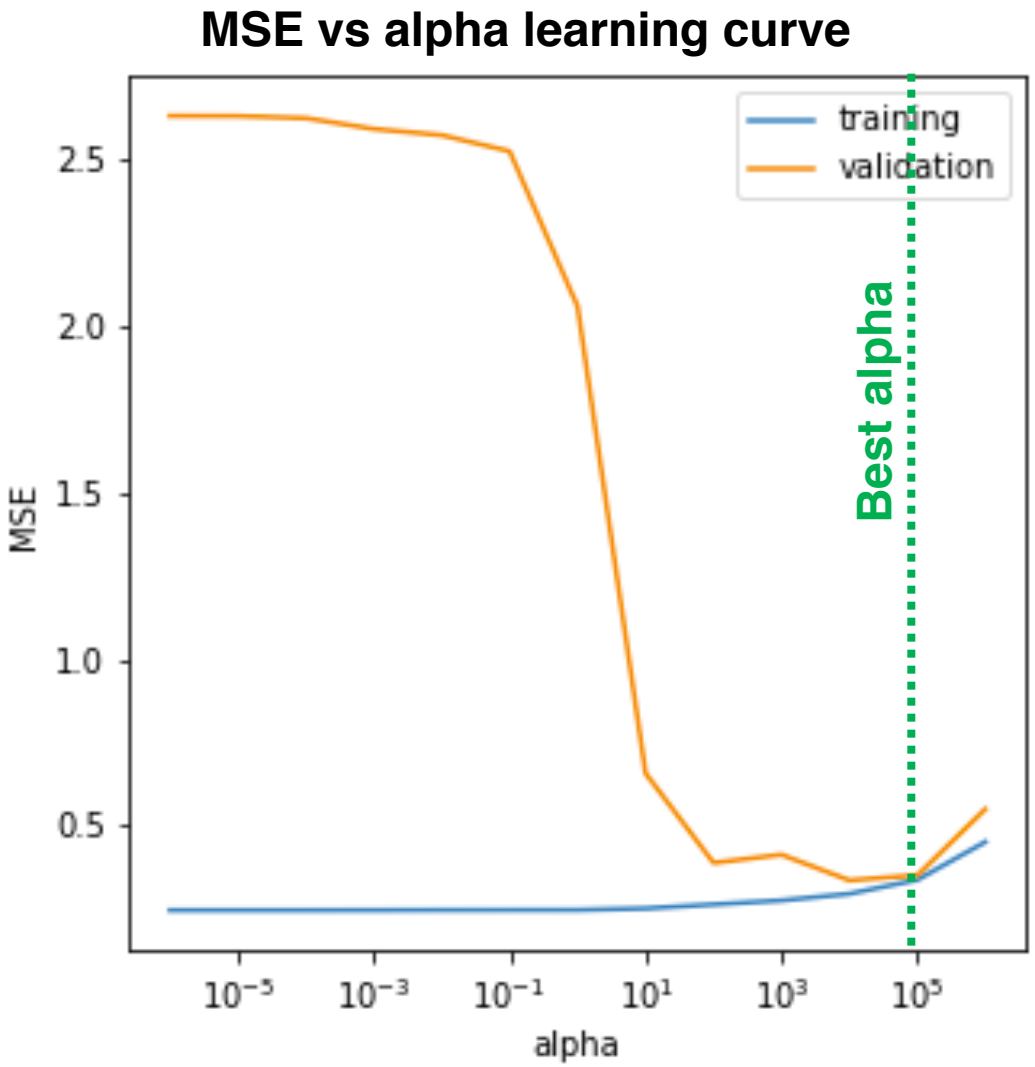
As we move to more complex models, hyperparameter optimization becomes onerous, but linear regression models are very cheap to train.

Dow Dataset – Ridge Regression

This is a typical learning curve for models with many collinear variables: **the training MSE gets worse with increasing alpha, but the validation MSE exhibits a minimum.**

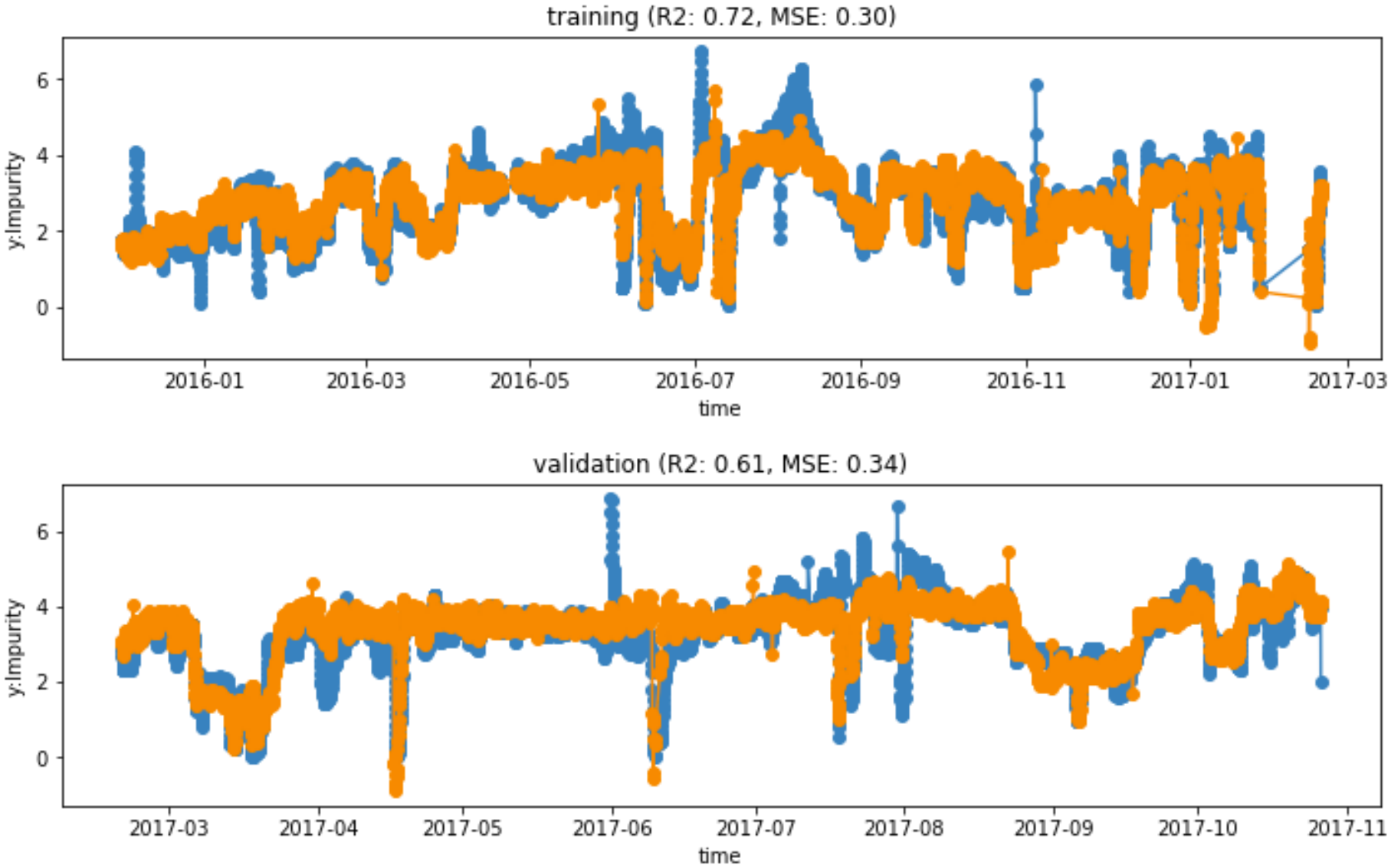
After a certain point, alpha starts impairing our model's ability to learn the relationship between **X** and **y**.

We could have used a finer alpha spacing if we wanted to get an even better model



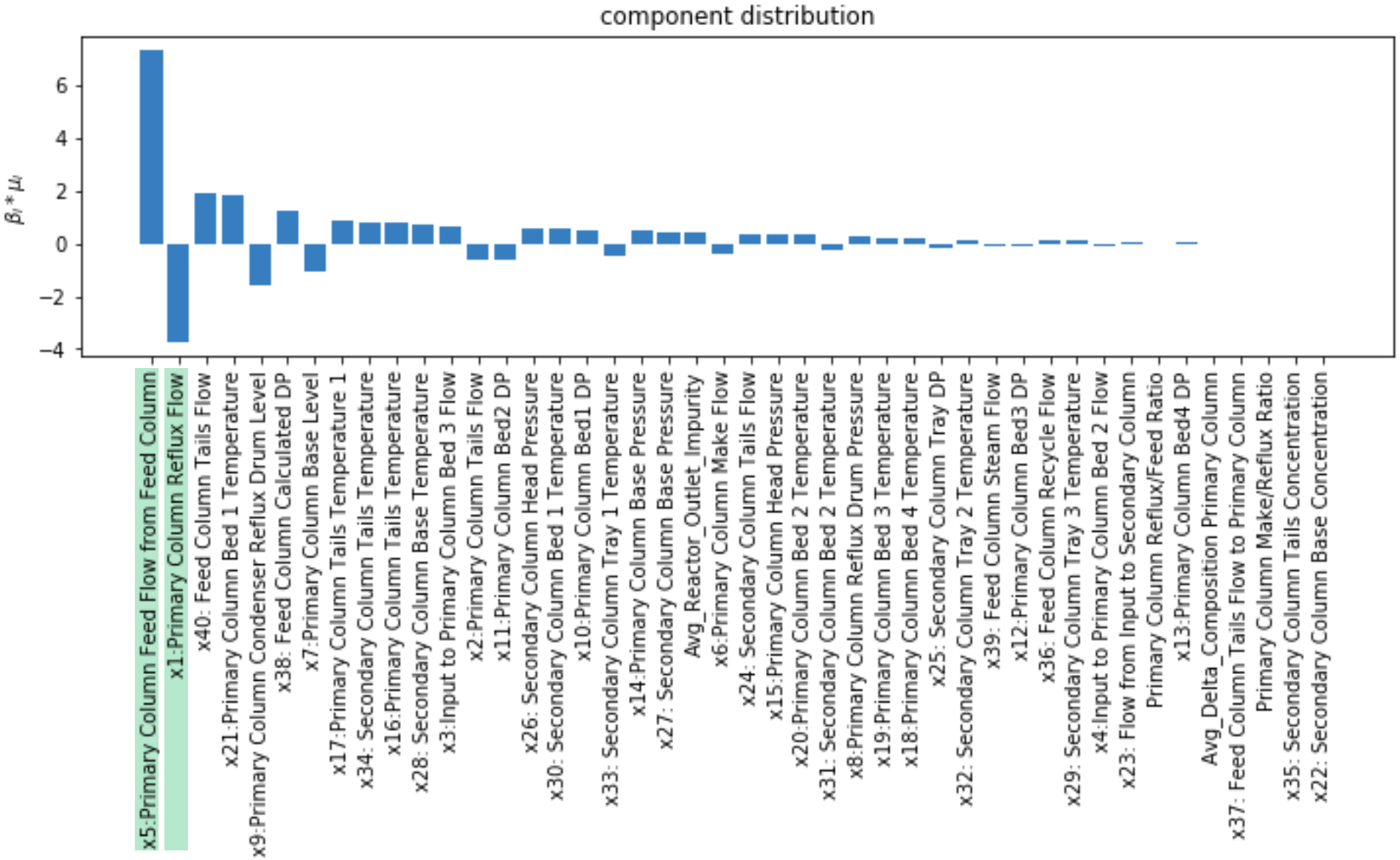
*corresponding R2 plot has identical behavior

Dow Dataset – Ridge Regression



The behavior on the validation dataset is MUCH better than for OLS.

Dow Dataset – Ridge Regression



Identifies the same two important variables as Lasso.

Dow Dataset – PLS Regression

With PLS we have a "hyperparameter" `n_components` that we need to choose. How do we do this?

We will use our dedicated validation data:

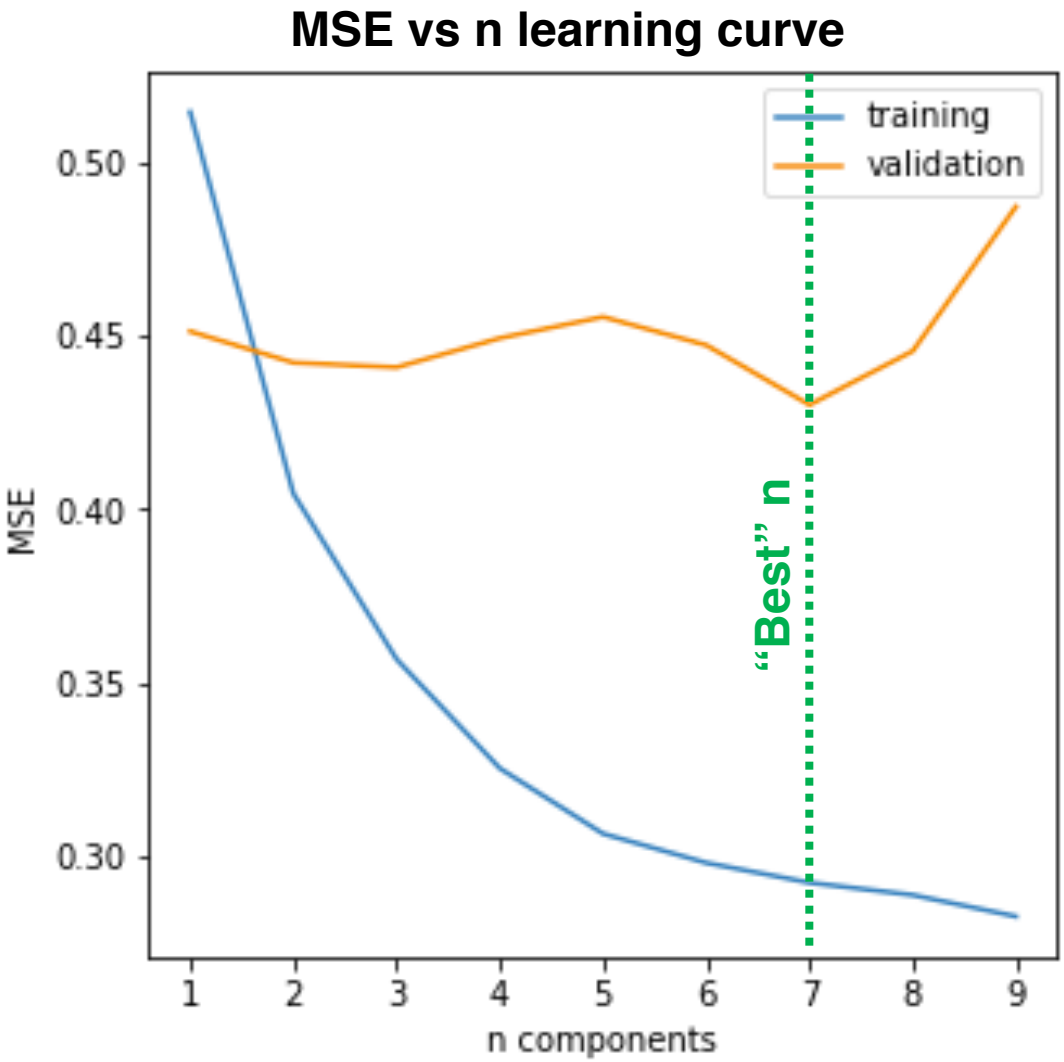
```
# Use validation data to select the best n_components hyperparameter
mse_train = []
mse_valid = []
r2_train  = []
r2_valid  = []
n_comp = range(1,10)
for i in n_comp:
  lin = PLSRegression(n_components=i) # call the constructor
  lin.fit(train_X,train_Y) # train the model
  mse_train += [mean_squared_error(train.iloc[:,-3],lin.predict(train_X))]
  mse_valid += [mean_squared_error(val.iloc[:,-3],lin.predict(val_X))]
  r2_train += [r2_score(train.iloc[:,-3],lin.predict(train_X))]
  r2_valid += [r2_score(val.iloc[:,-3],lin.predict(val_X))]

# Train on the best n_components
n_best = n_comp[np.argmin(mse_valid)]
print("best n: {}".format(n_best))
lin = PLSRegression(n_components=n_best) # call the constructor
lin.fit(train_X,train_Y) # train the model
```

As we move to more complex models, hyperparameter optimization becomes onerous, but linear regression models are very cheap to train.

Dow Dataset – PLS Regression

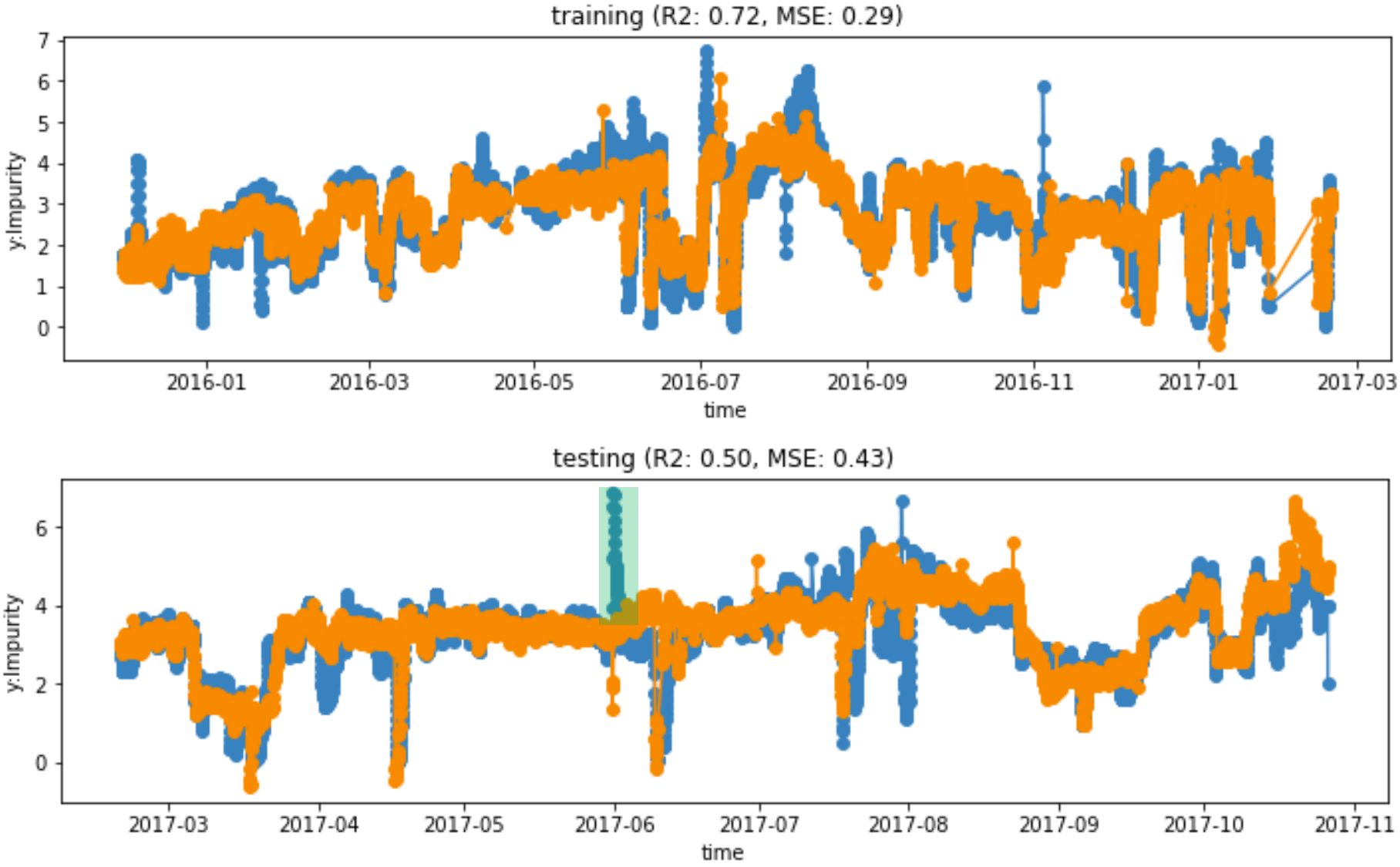
Recall that PLS learning curves are reversed relative to Lasso/Ridge.

The minimum of the validation curve is typically less clearly defined, and experimentation while consulting the components is common.



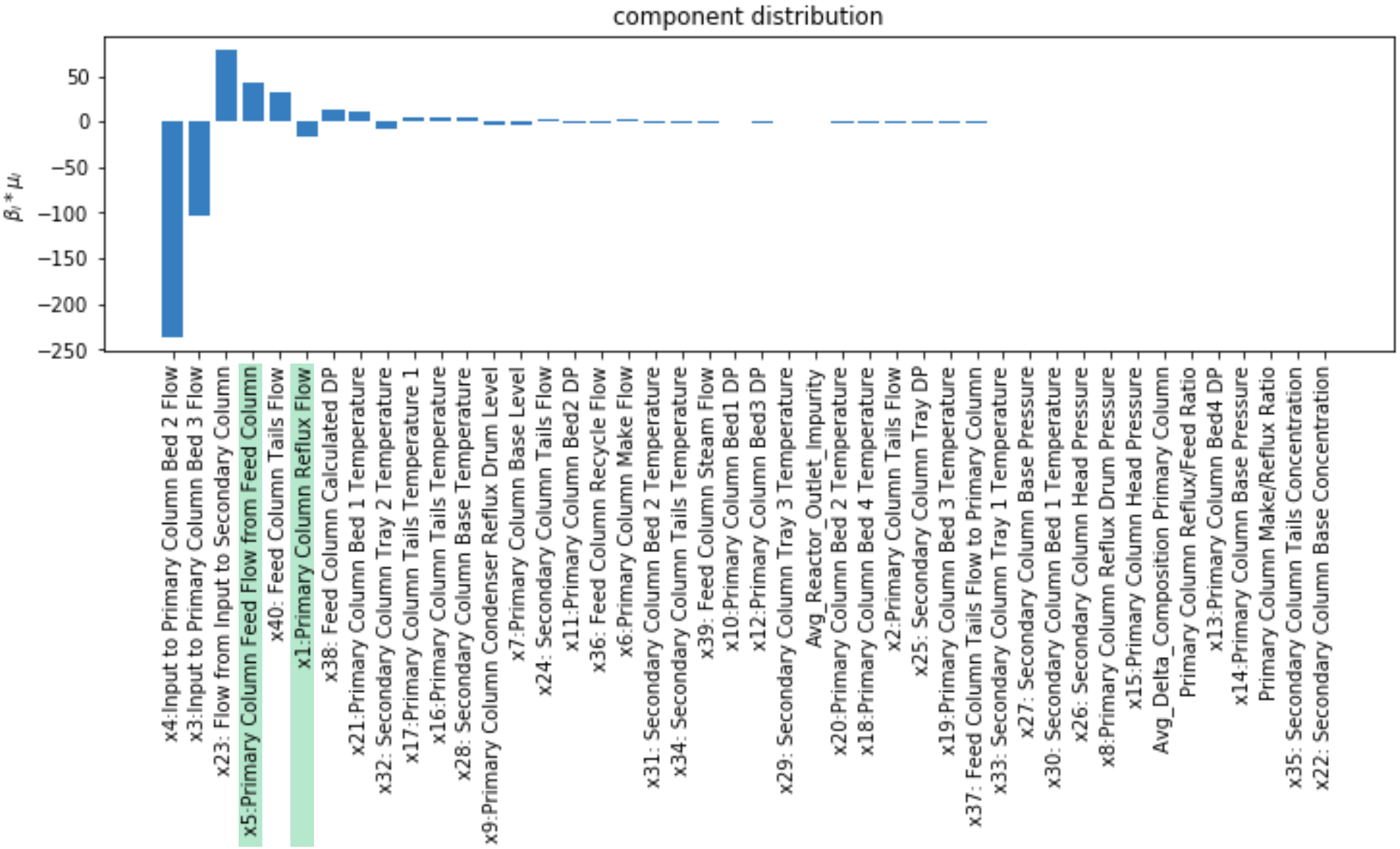
*corresponding R2 plot has identical behavior

Dow Dataset – PLS Regression



None of the models have been able to reproduce this peak. Evidence that there is some phenomenon in the validation set that isn't in the training set (linearly).

Dow Dataset – PLS Regression



Flows are overall identified as the most important variables.