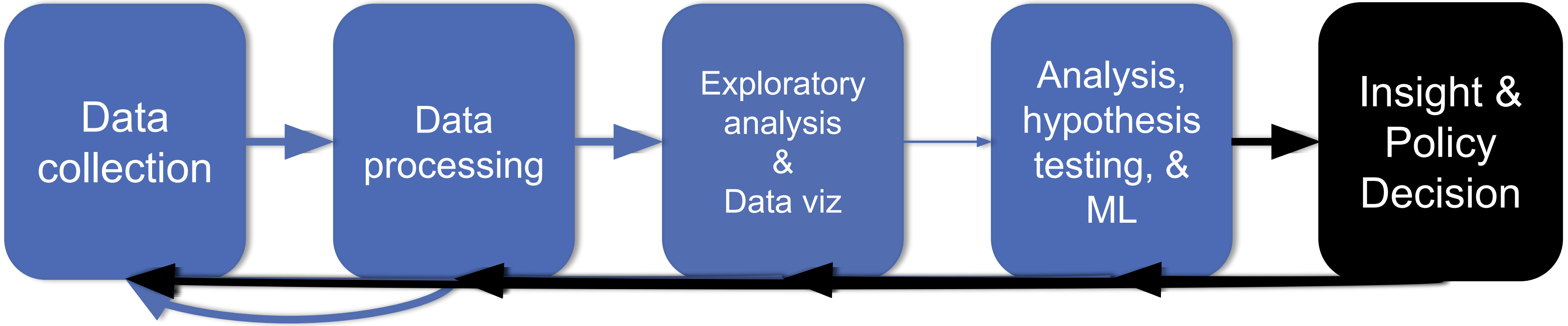
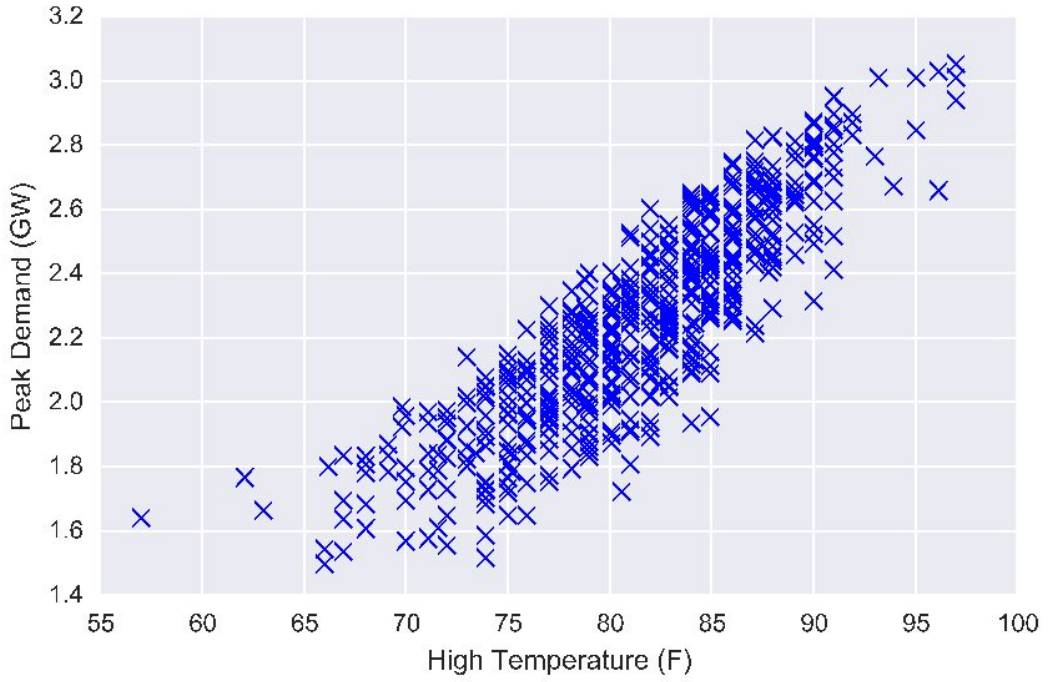
TODAY'S LECTURE


Data collection
Data processing
Exploratory analysis & Data viz
Analysis, hypothesis testing, & ML
Insight & Policy Decision

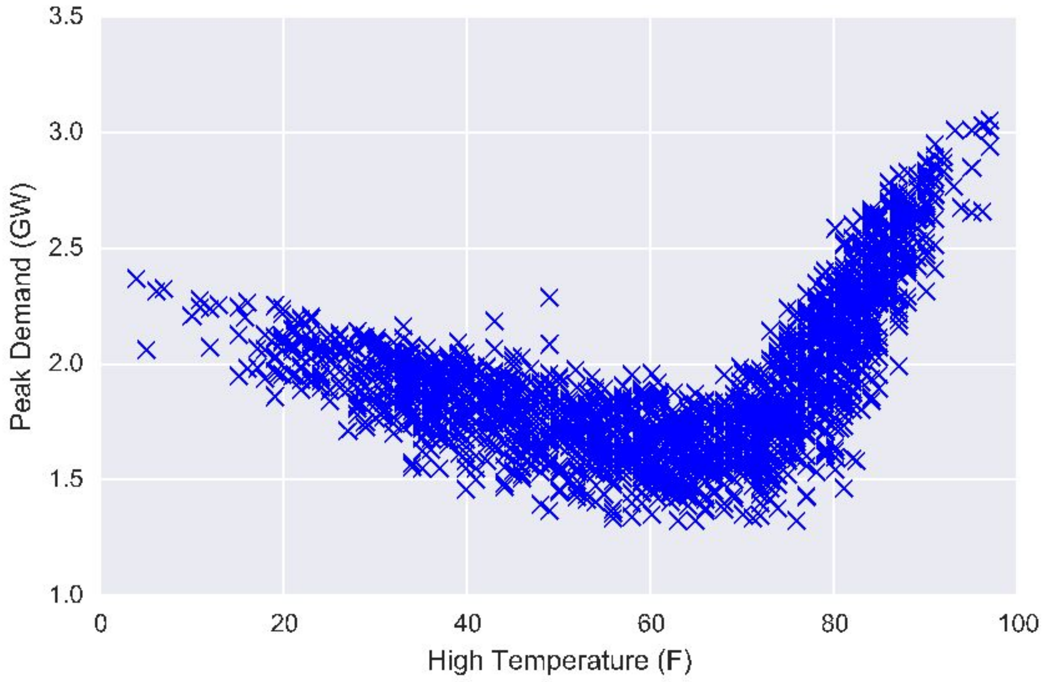
FILLING IN THE GAPS: NONLINEAR REGRESSION & REGULARIZATION

Thanks: Zico Kolter

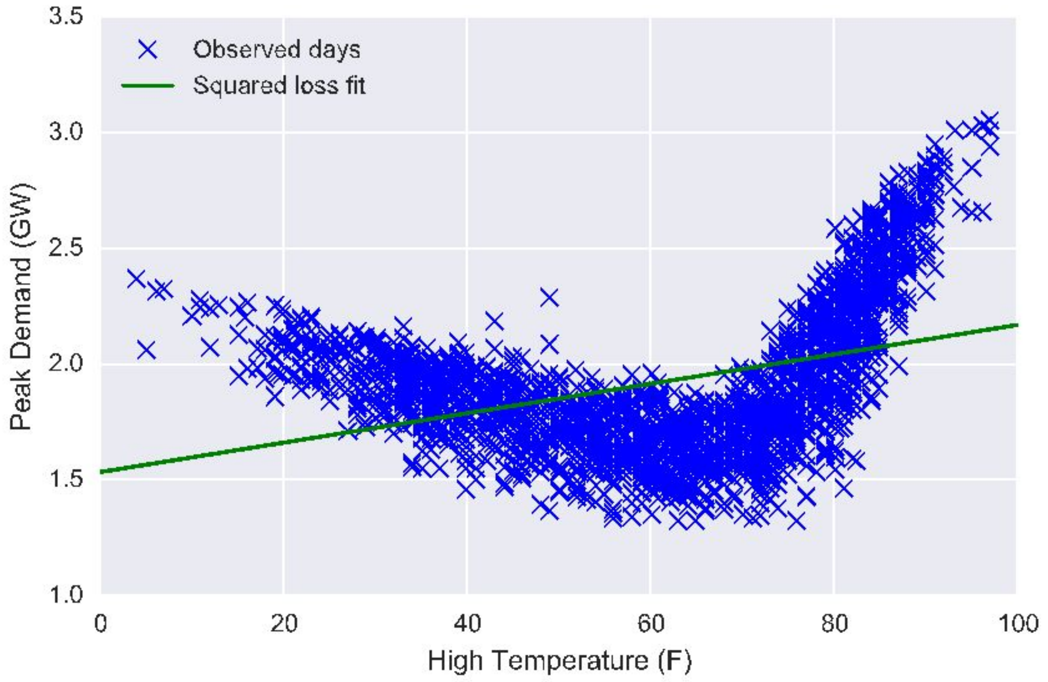
Peak demand vs. temperature (summer months)



Peak demand vs. temperature (all months)



Linear regression fit



“Non-linear” regression

Thus far, we have illustrated linear regression as “drawing a line through through the data”, but this was really a function of our input features

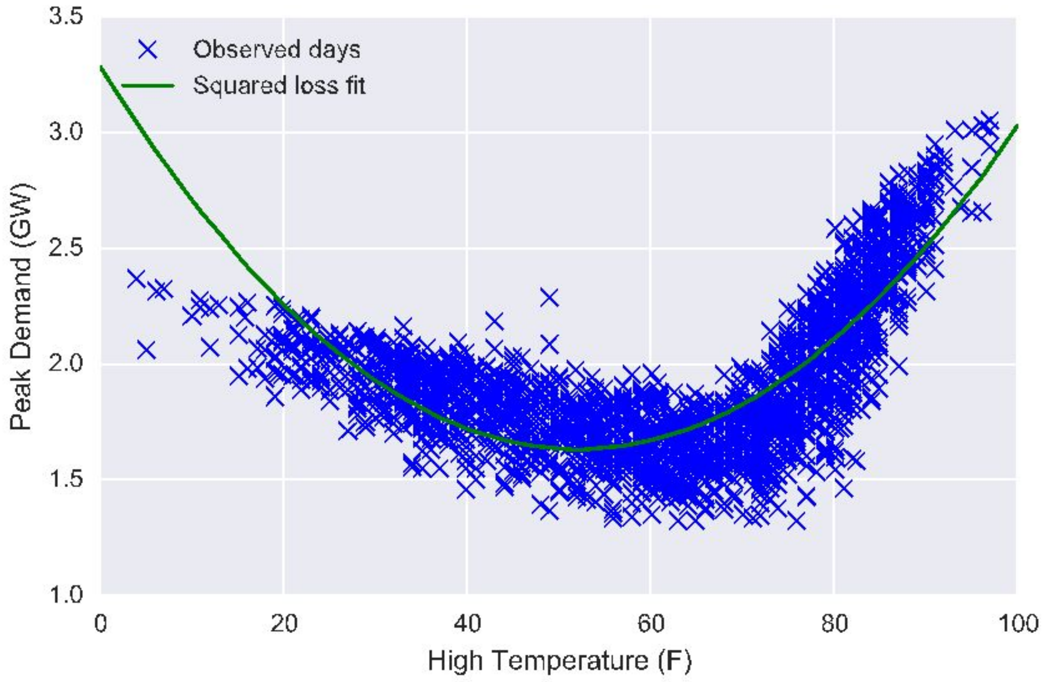
Though it may seem limited, linear regression algorithms are quite powerful when applied to *non-linear features* of the input data, e.g.

$$x^{(i)} = \begin{bmatrix} (\text{High-Temperature}^{(i)})^2 \\ \text{High-Temperature}^{(i)} \\ 1 \end{bmatrix}$$

Same hypothesis class as before $h_\theta(x) = \theta^T x$, but now prediction will be a non-linear function of base input (e.g. a quadratic function)

Same least-squares solution $\theta = (X^T X)^{-1} X^T y$

Polynomial features of degree 2



Code for fitting polynomial

The only element we need to add to write this non-linear regression is the creation of the non-linear features

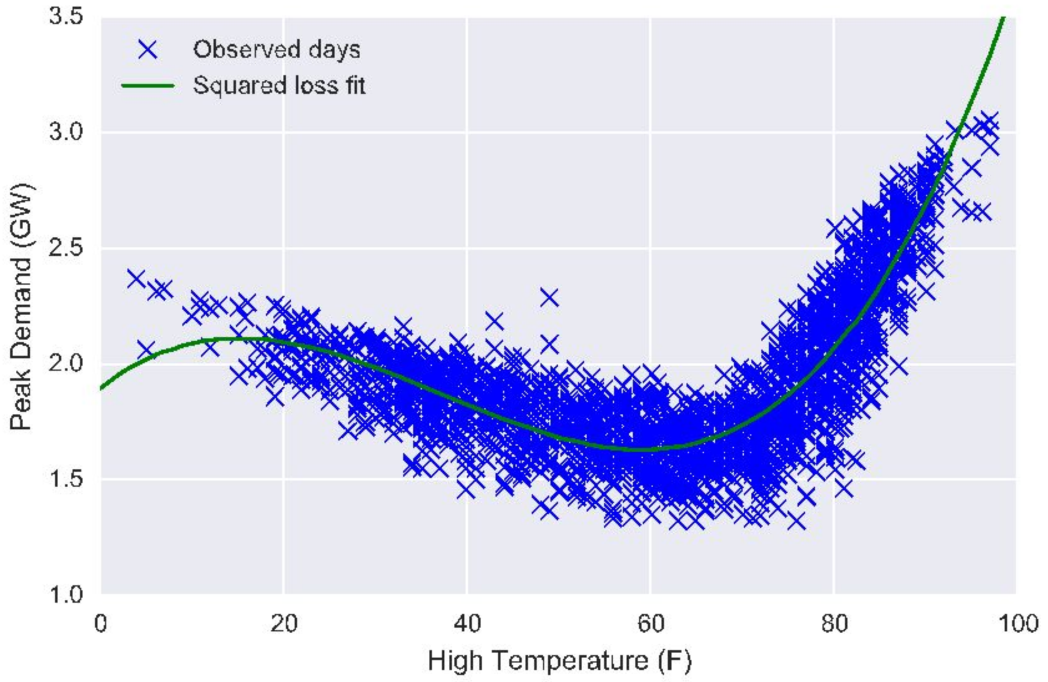
```
x = df_daily.loc[:,"Temperature"]
min_x, rng_x = (np.min(x), np.max(x) - np.min(x))
x = 2*(x - min_x)/rng_x - 1.0
y = df_daily.loc[:,"Load"]

X = np.vstack([x**i for i in range(poly_degree,-1,-1)]).T
theta = np.linalg.solve(X.T.dot(X), X.T.dot(y))
```

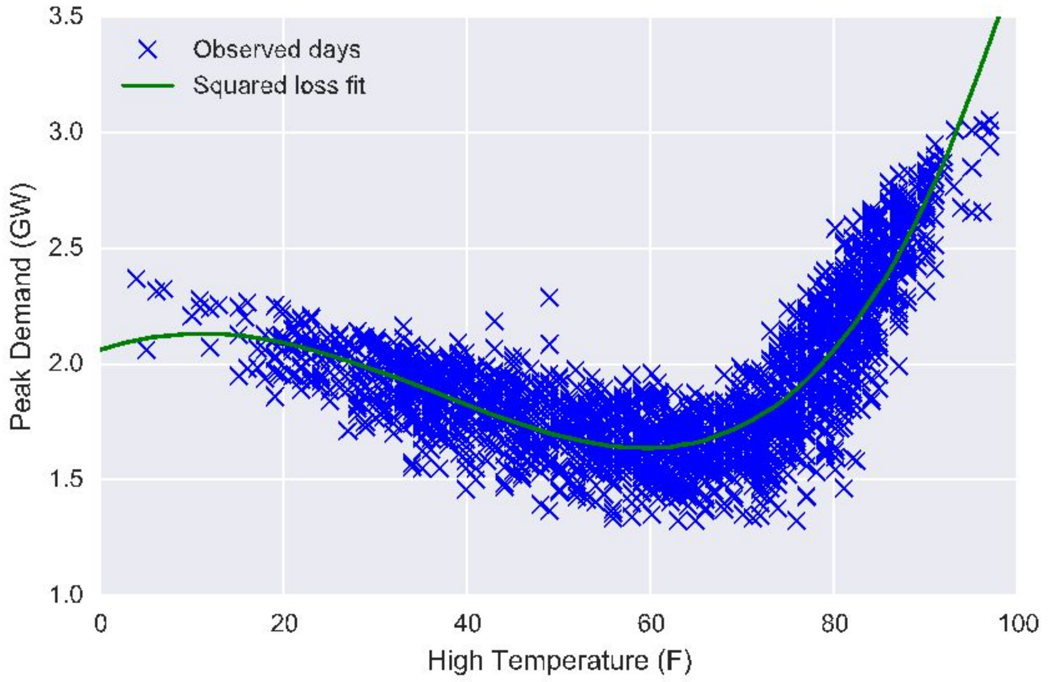
Output learned function:

```
x0 = 2*(np.linspace(xlim[0], xlim[1],1000) - min_x)/rng_x - 1.0
X0 = np.vstack([x0**i for i in range(poly_degree,-1,-1)]).T
y0 = X0.dot(theta)
```

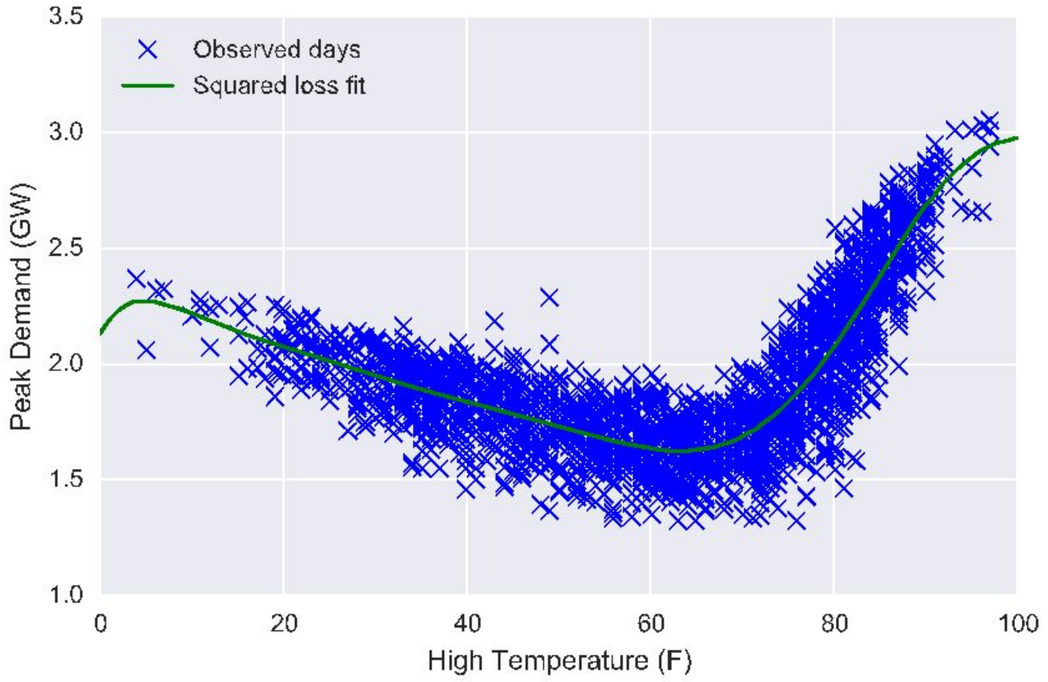
Polynomial features of degree 3



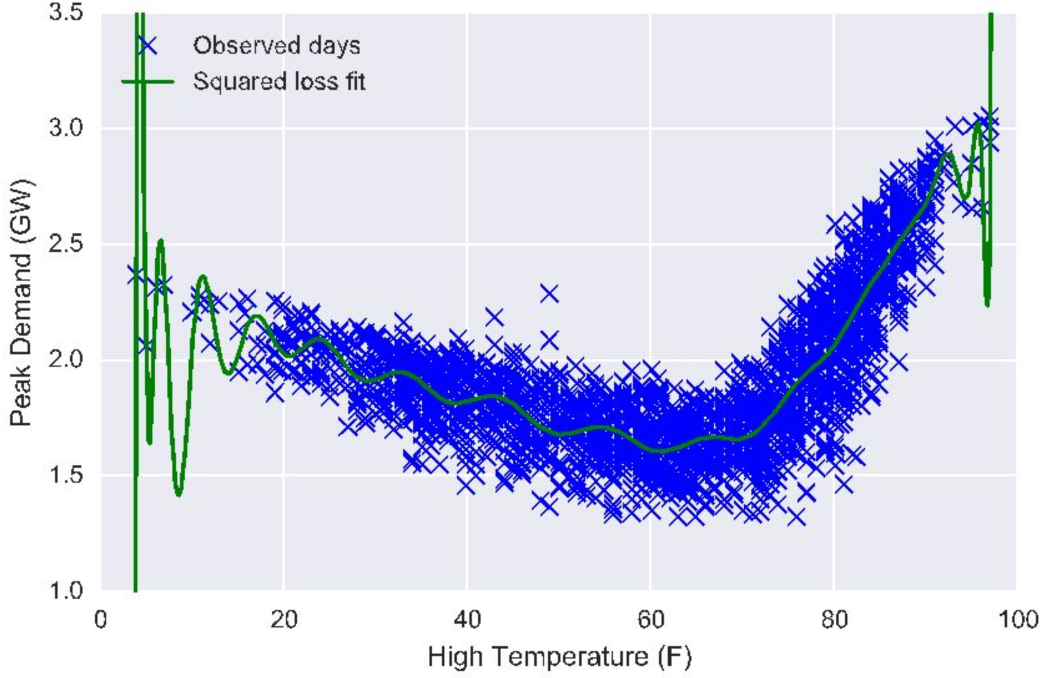
Polynomial features of degree 4



Polynomial features of degree 10



Polynomial features of degree 50



Generalization error

The problem we the canonical machine learning problem is that we don't *really* care about minimizing this objective on the given data set

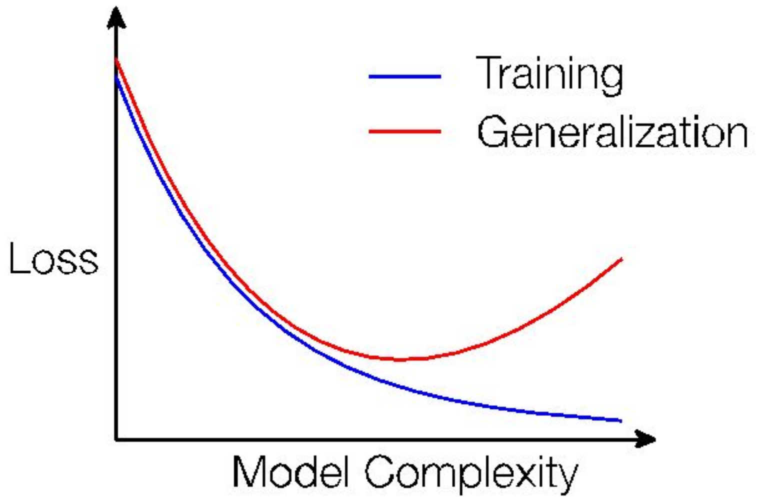
$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

What we really care about is how well our function will generalize to *new examples* that we *didn't* use to train the system (but which are drawn from the "same distribution" as the examples we used for training)

The higher degree polynomials exhibited *overfitting*: they actually have very *low* loss on the training data, but create functions we don't expect to generalize well

Cartoon version of overfitting

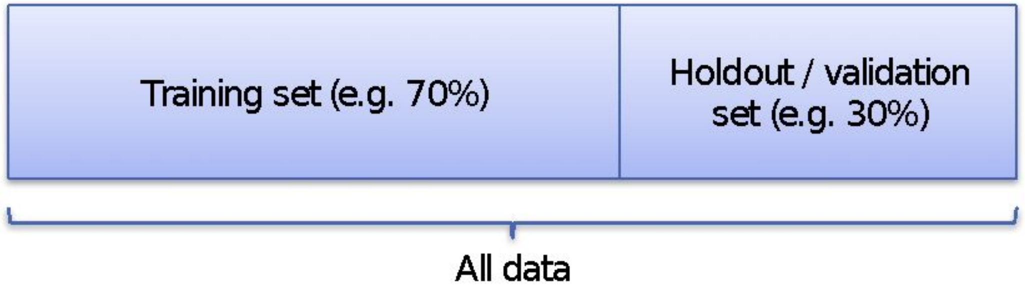
As model becomes more complex, training loss always decreases; generalization loss decreases to a point, then starts to increase



Cross-validation

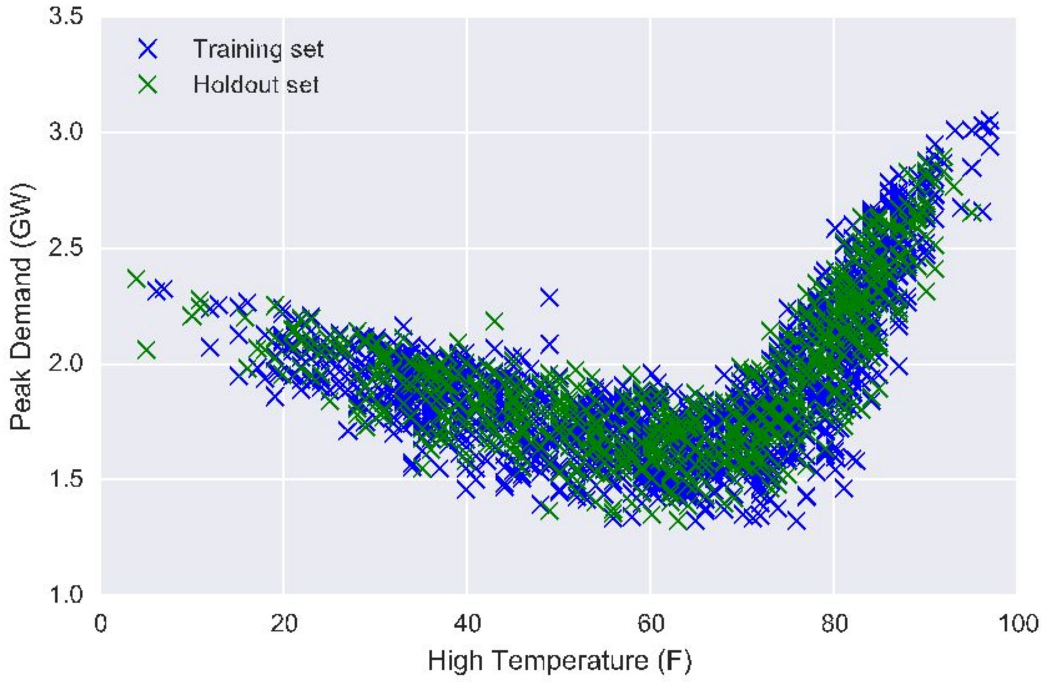
Although it is difficult to quantify the true generalization error (i.e., the error of these algorithms over the *complete* distribution of possible examples), we can approximate it by ***holdout cross-validation***

Basic idea is to split the data set into a training set and a holdout set

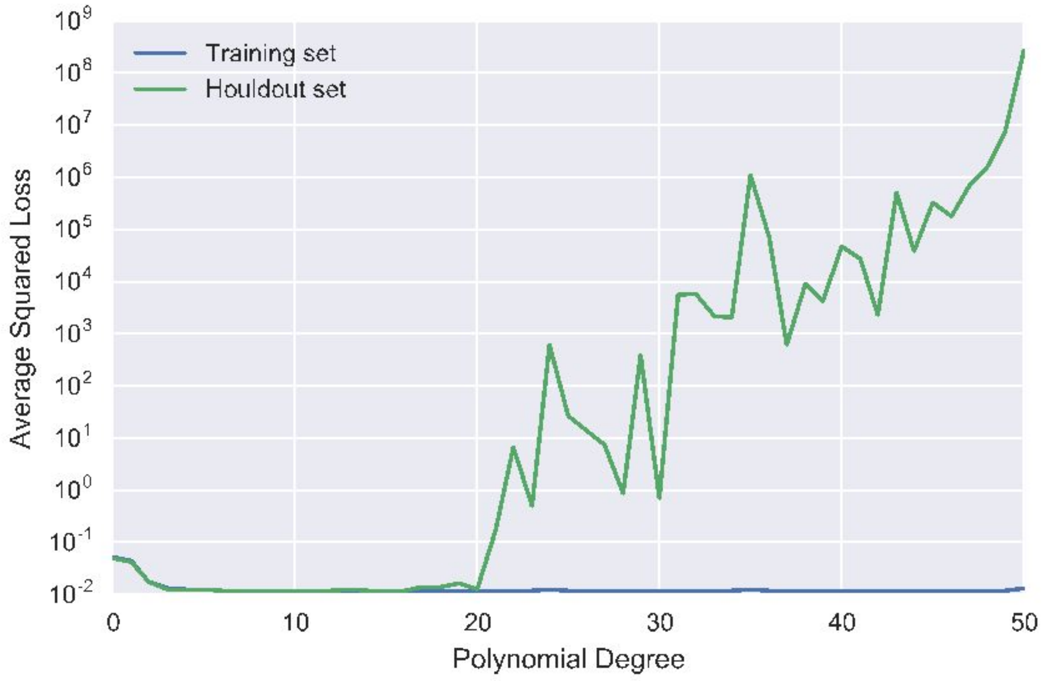


Train the algorithm on the training set and evaluate on the holdout set

Illustrating cross-validation



Training and cross-validation loss by degree



Regularization

We have seen that the degree of the polynomial acts as a natural measure of the "complexity" of the model, higher degree polynomials are more complex (taken to the limit, we fit any finite data set exactly)

But fitting these models also requires extremely *large* coefficients on these polynomials

For 50 degree polynomial, the first few coefficients are

$$\theta = -3.88 \times 10^6, 7.60 \times 10^6, 3.94 \times 10^6, -2.60 \times 10^7, \ldots$$

This suggests an alternative way to control model complexity: keep the *weights small* (**regularization**)

Regularized loss minimization

This leads us back to the regularized loss minimization problem we saw before, but with a bit more context now:

$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{2}\|\theta\|_2^2$$

This formulation trades off loss on the *training* set with a penalty on high values of the parameters

By varying λ from zero (no regularization) to infinity (infinite regularization, meaning parameters will all be zero), we can sweep out different sets of model complexity

Regularized least squares

For least squares, there is a simple solution to the regularized loss minimization problem

$$\text{minimize}_\theta \ \frac{1}{2}\|X\theta - y\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2$$

Taking gradients by the same rules as before gives:

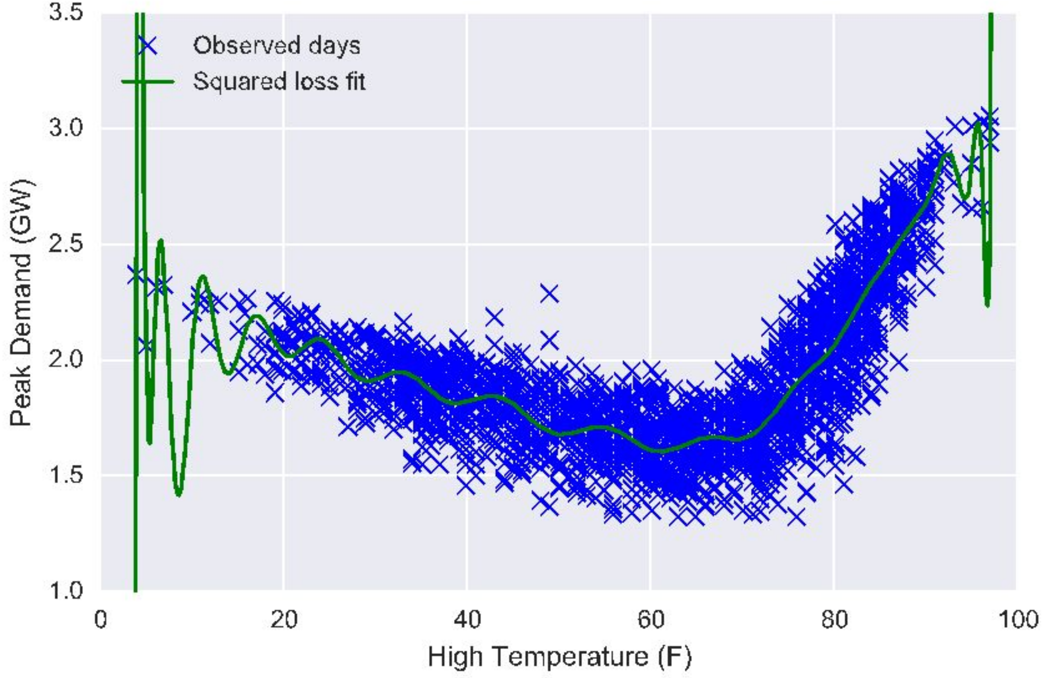
$$\nabla_\theta \left(\frac{1}{2}\|X\theta - y\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2\right) = X^T(X\theta - y) + \lambda\theta$$

Setting gradient equal to zero leads to the solution

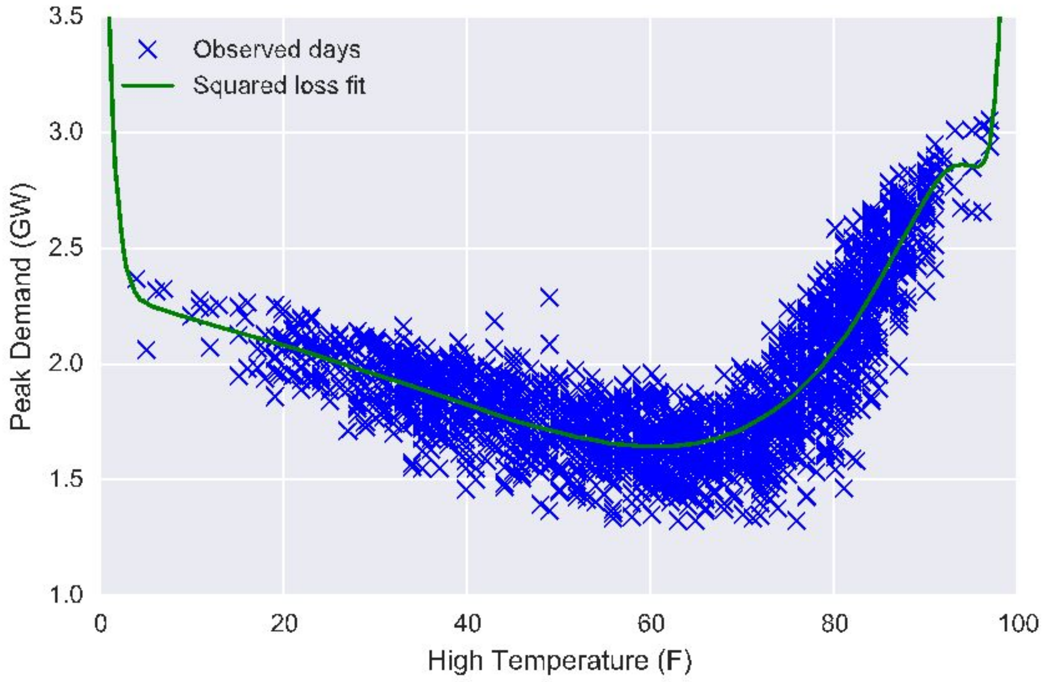
$$X^T X\theta + \lambda\theta = X^T y \implies \theta = (X^T X + \lambda I)^{-1} X^T y$$

Looks just like the normal equations but with an additional λI term

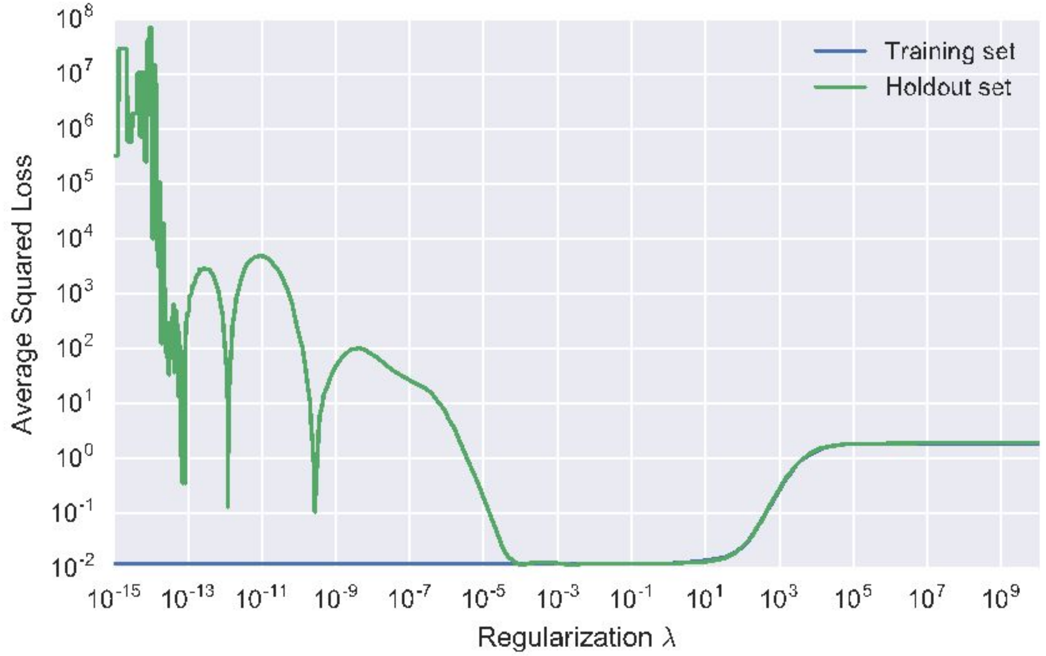
50 degree polynomial fit



50 degree polynomial fit – $\lambda = 1$



Training/cross-validation loss by regularization



Notation for more general features

We previously described polynomial features for a *single* raw input, but if our raw input is itself multi-variate, how do we define polynomial features?

Deviating a bit from past notion, for precision here we're going to use $x^{(i)} \in \mathbb{R}^k$ to denote the *raw* inputs, and $\phi^{(i)} \in \mathbb{R}^n$ to denote the input features we construct (also common to use the notation $\phi(x^{(i)})$)

We'll also drop (i) superscripts, but important to understand we're transforming *each* feature this way

E.g., for the high temperature:

$$x = [\text{High-Temperature}], \qquad \phi = \begin{bmatrix} x^2 \\ x \\ 1 \end{bmatrix}$$

Polynomial features in general

One possibility for higher degree polynomials is to just use an independent polynomial over each dimension (here of degree d)

$$x \in \mathbb{R}^k \implies \phi = \begin{bmatrix} x_1^d \\ \vdots \\ x_1 \\ \vdots \\ x_k^d \\ \vdots \\ x_k \\ 1 \end{bmatrix} \in \mathbb{R}^{kd+1}$$

But this ignores cross terms between different features, i.e., terms like $x_1 x_2^2 x_k$

Polynomial features in general

A better generalization of polynomials is to include *all* polynomial terms between raw inputs up to degree @

$$! \in \mathbb{R}^{=} \Longrightarrow > = \left\{ \prod_{"=1}^{=} !_{"}^{D_E} : \sum_{"=1}^{2} G \leq @ \right\} \in \mathbb{R}^{\binom{=+A}{=}}$$

Code to generate all polynomial features with degree exactly @

```
from itertools import combinations_with_replacement
[np.prod(a) for a in combinations_with_replacement(x, d)]
```

Code to generate all polynomial features with degree up to @

```
[np.prod(a) for i in range(d+1) for a in combinations_with_replacement(x,i)]
```

`combinations_with_replacement(p,r):`
r-length tuples, in sorted order, with replacement

Code for general polynomials

The following code efficiently (relatively) generates all polynomials up to degree @for an entire data matrix (

```
def poly(X,d):
    return np.array([reduce(operator.mul, a, np.ones(X.shape[0]))
                     for i in range(1,d+1)
                     for a in combinations_with_replacement(X.T, i)]).T
```

It is using the same logic as above, but applying it to entire columns of the data at a time, and thus only needs one call to `combinations_with_replacement`

Radial basis functions (RBFs)

For $x \in \mathbb{R}^k$, select some set of p *centers*, $\mu^{(1)}, \ldots, \mu^{(p)}$ (we'll discuss shortly how to select these), and create features

$$\phi = \left\{ \exp\left(- \frac{\|x - \mu^{(i)}\|_2^2}{2\sigma^2} \right) : i = 1, \ldots, p \right\} \bigcup \{1\} \in \mathbb{R}^{p+1}$$

Very important: need to normalize columns of X (i.e., different features), to all be the same range, or distances wont be meaningful

(Hyper)parameters of the features include the choice of the p centers, and the choice of the *bandwidth* σ

Choose centers, i.e., to be a uniform grid over input space, can choose σ e.g. using cross validation (don't do this, though, more on this shortly)

Example radial basis function

Example:

$$x = [\mathrm{High} - \mathrm{Temperature}],$$

$$\mu^{(1)} = [20], \mu^{(2)} = [25], \ldots, \mu^{(16)} = [95], \sigma = 10$$

Leads to features:

$$\phi = \begin{bmatrix} \exp(-(\text{High-Temperature} - 20)^2/200) \\ \vdots \\ \exp(-(\text{High-Temperature} - 95)^2/200) \\ 1 \end{bmatrix}$$

Code for generating RBFs

The following code generates a complete set of RBF features for an entire data matrix ($\in \mathbb{R}^{0 \ \times =}$ and matrix of centers J $\in \mathbb{R}^{K \times =}$

```
def rbf(X,mu,sig):
    sqdist = (-2*X.dot(mu.T) +
              np.sum(X**2,axis=1)[:,None] +
              np.sum(mu**2,axis=1))
    return np.exp(-sqdist/(2*sig**2))
```

Important "trick" is to efficiently compute distances between *all* data points and all centers

Difficulties with general features

The challenge with these general non-linear features is that the number of potential features grows very quickly in the dimensionality of the raw input

Polynomials: k-dimensional raw input $\Longrightarrow \binom{k+d}{k} = O(d^k)$ total features (for fixed d)

RBFs: k-dimensional raw input, uniform grid with d centers over each dimension $\Longrightarrow d^k$ total features

These quickly become impractical for large feature raw input spaces

Practical polynomials

Don't use the full set of all polynomials, for anything but very low dimensional input data (say $k \leq 4$)

Instead, form polynomials only of features where you know that the relationship may be important:

E.g. $\mathrm{Temperature}^2 \cdot \mathrm{Weekday}$, but not $\mathrm{Temperature} \cdot \mathrm{Humidity}$

For binary raw inputs, no point in every taking powers ($x_i^2 = x_i$)

These elements do all require some insight into the problem

Practical RBFs

Don’t create RBF centers in a grid over your raw input space (your data will *never* cover an entire high-dimensional space, but will lie on a subset)

Instead, pick centers by randomly choosing p data points in the training set (a bit fancier, run k-means to find centers, which we’ll describe later)

Don’t pick σ using cross validation

Instead, choose the following (called the *median trick)*

$$\sigma = \text{median}(\{\|\mu^{(i)} - \mu^{(j)}\|_2, i, j = 1, \ldots, p\})$$

K-Folds Cross Validation

Cross Validation is good!
Let’s do more of it!

1. Split the data set into ‘k’ folks (5-10)
2. Fit your model using k-1 of the folds
3. Validate the model using the kth fold (remember your error)
4. Repeat steps 2 and 3, changing with one of the folds being used for validation
5. Average of the errors

K-Folds Cross Validation

Things to consider:

1. **What we have described is "leave on out" (LOO) CV**
   a. There is also "Leave P Out"
   b. LOO lets you train the model better but have fewer tests. LPO better tests but if its a steep learning curve, you might overestimate the error
2. **Random sampling can be bad (what about class labels)**
   a. StratifiedKFold helps you ensure that each fold as a similar proportion of various class labels

Nonlinear classification

Just like linear regression, the nice thing about using nonlinear features for classification is that our algorithms remain exactly the same as before

I.e., for an SVM, we just solve (using gradient descent)

$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\} + \frac{\lambda}{2}\|\theta\|_2^2$$

Only difference is that $x^{(i)}$ now contains non-linear functions of the input data

Support Vector Machines (SVM)

Linear Regressions are great but sometimes we don’t want a ‘predicative’ line. Sometimes we want a dividing line.

Idea is to find the hyperplane that best divides the dataset

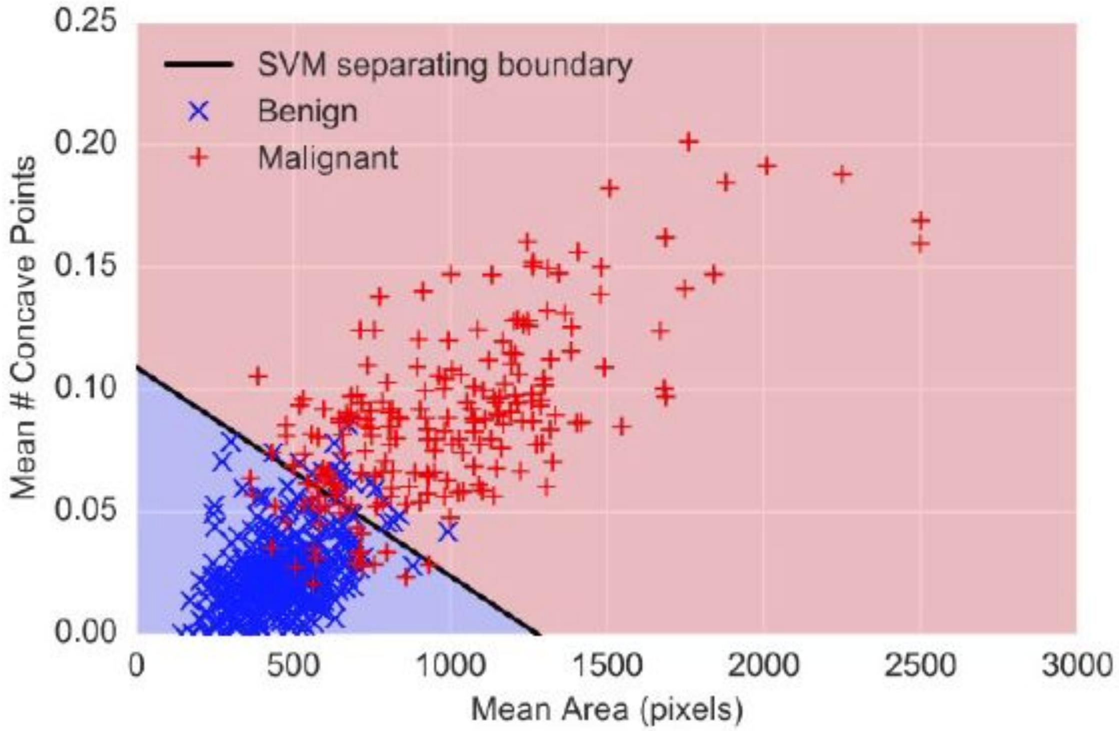
If linear regressions can be thought of a statistical, SVMs are geometric

Good for ‘medium’ sized datasets (low 10’s of thousands)

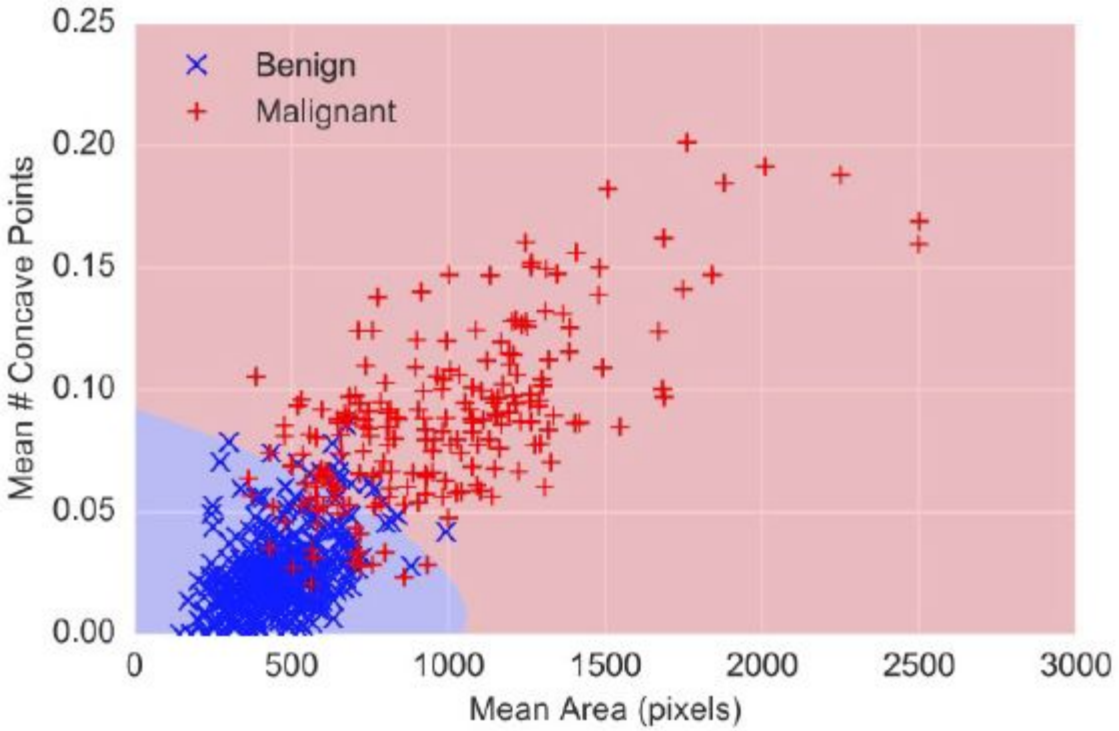
Has classification (SVC) and regression (SVR) variants

SKLearn has implementations!

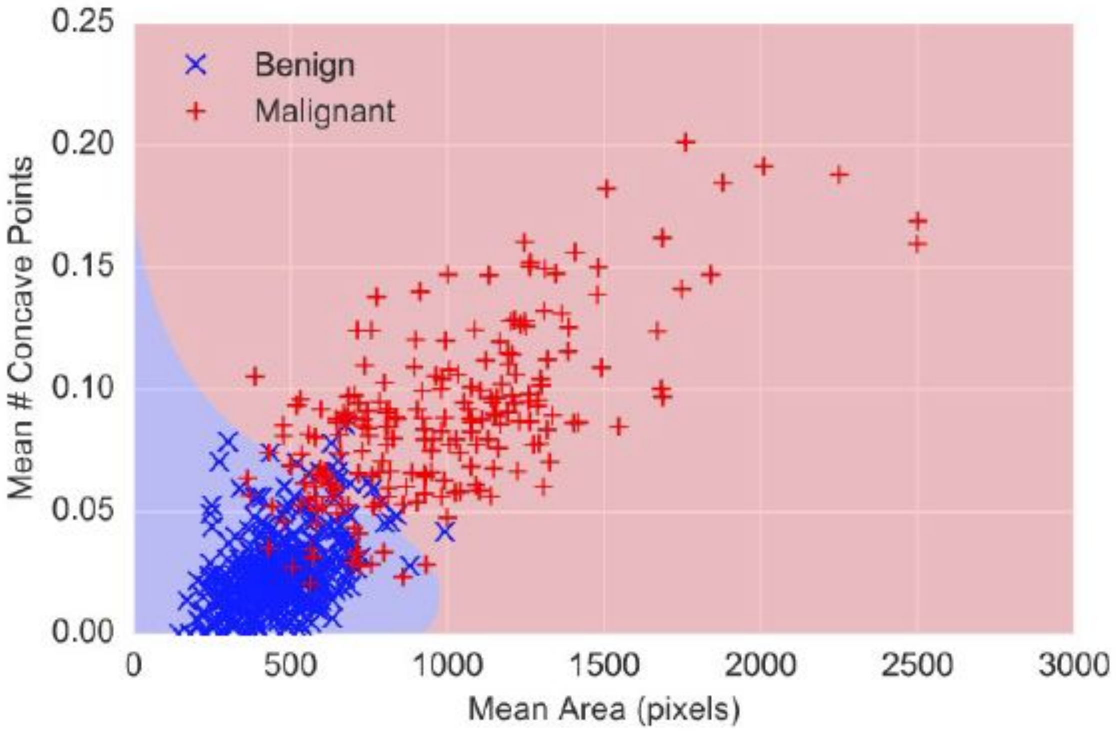
Linear SVM on cancer data set



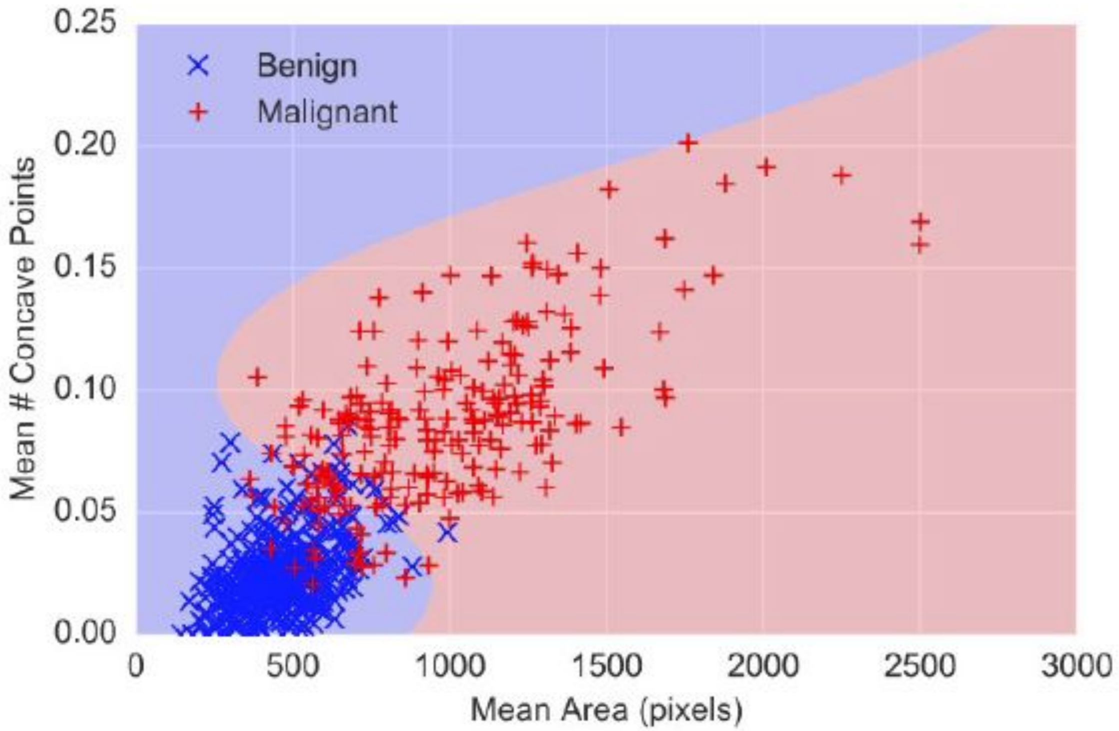
Polynomial features $d = 2$



Polynomial features $d = 3$



Polynomial features $d = 10$


Benign
Malignant
Mean # Concave Points
Mean Area (pixels)