15-388/688 - Practical Data Science: Nonlinear modeling, cross-validation, regularization, and evaluation

J. Zico Kolter
Carnegie Mellon University
Fall 2016

Outline

Example: return to peak demand prediction

Overfitting, generalization, and cross validation

Regularization

General nonlinear features

Nonlinear classification

Evaluating machine learning algorithms

Other (classification) metrics

Announcements, 10/10

Tutorial example to be released this evening (apologies, again, for delay)

Class project assignment to be released tonight

We’re going to take a Piazza poll regarding the final project presentation during final exam time

On Piazza, try to post follow-ups as much as possible instead of starting new posts (counts the same towards class participation)

Announcements, 10/12

HW 3 due tonight (taking late days will let you submit up until Saturday)

GIS tutorial and final project description posted to web page

Important dates:

- 10/24: Final project proposals
- 11/11: Final project midterm report
- **12/9:** Final project reports
- **12/14:** Final project presentations (videos)

Outline

Example: return to peak demand prediction

Overfitting, generalization, and cross validation

Regularization

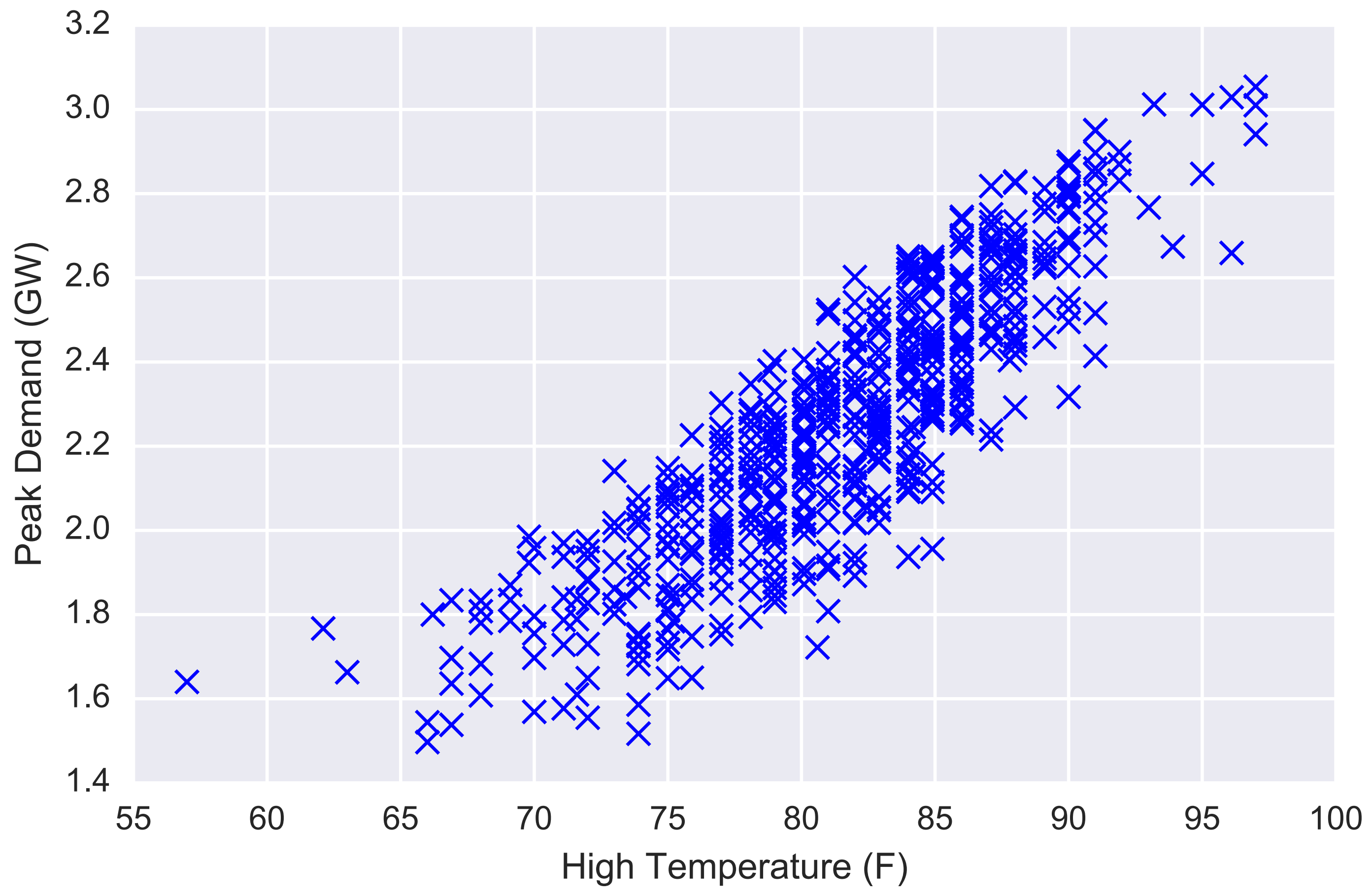
General nonlinear features

Nonlinear classification

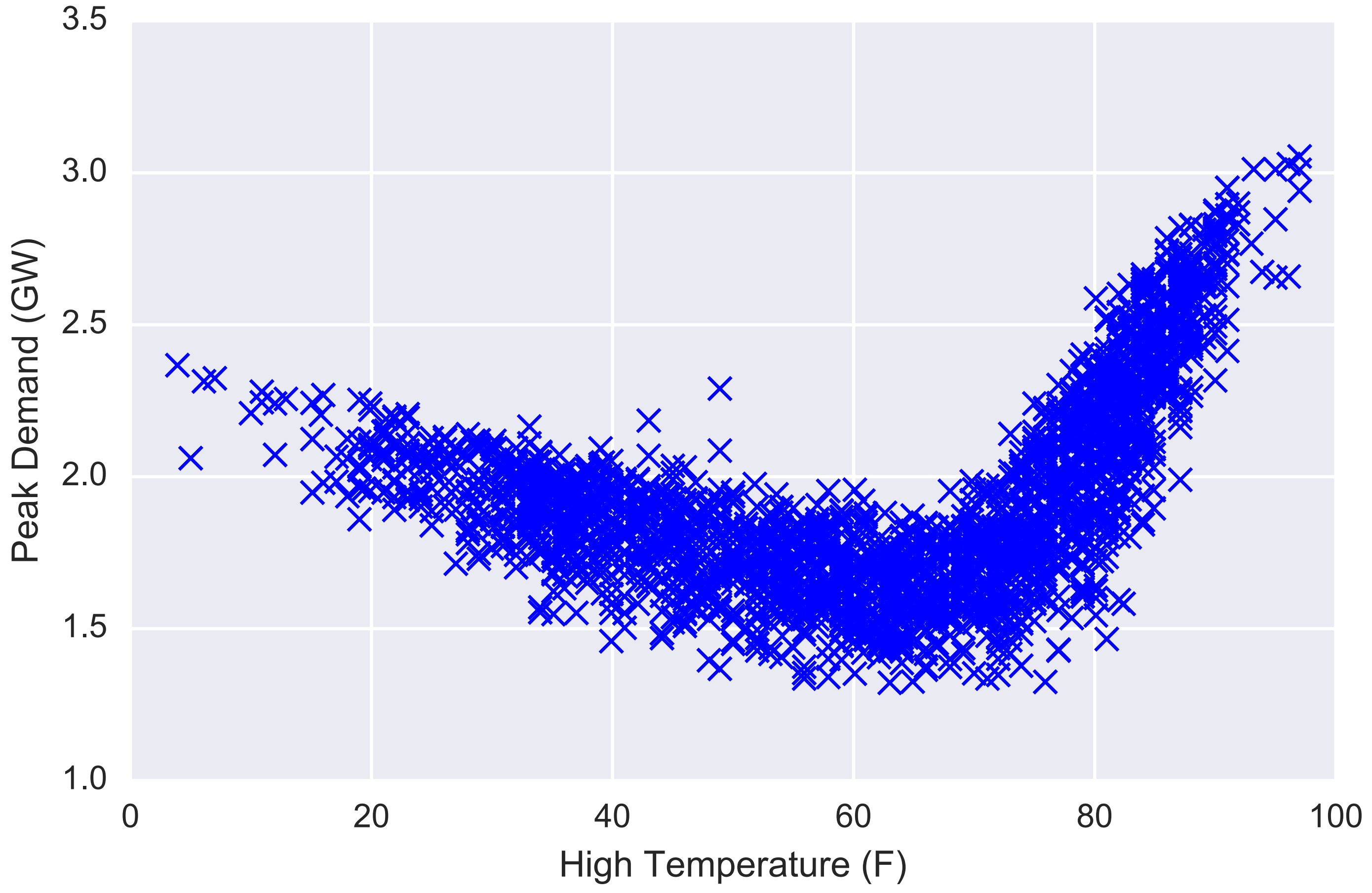
Evaluating machine learning algorithms

Other (classification) metrics

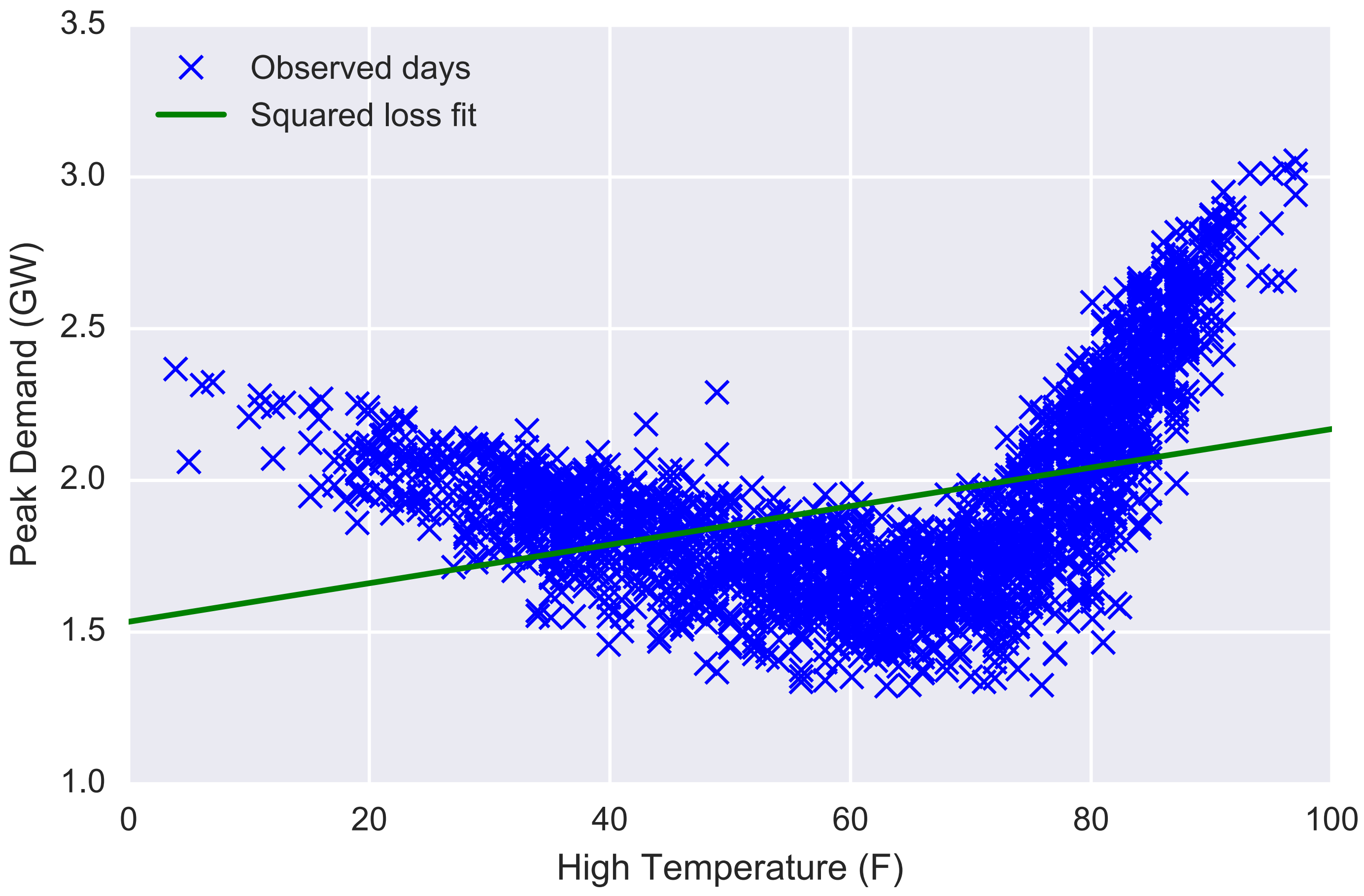
Peak demand vs. temperature (summer months)



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

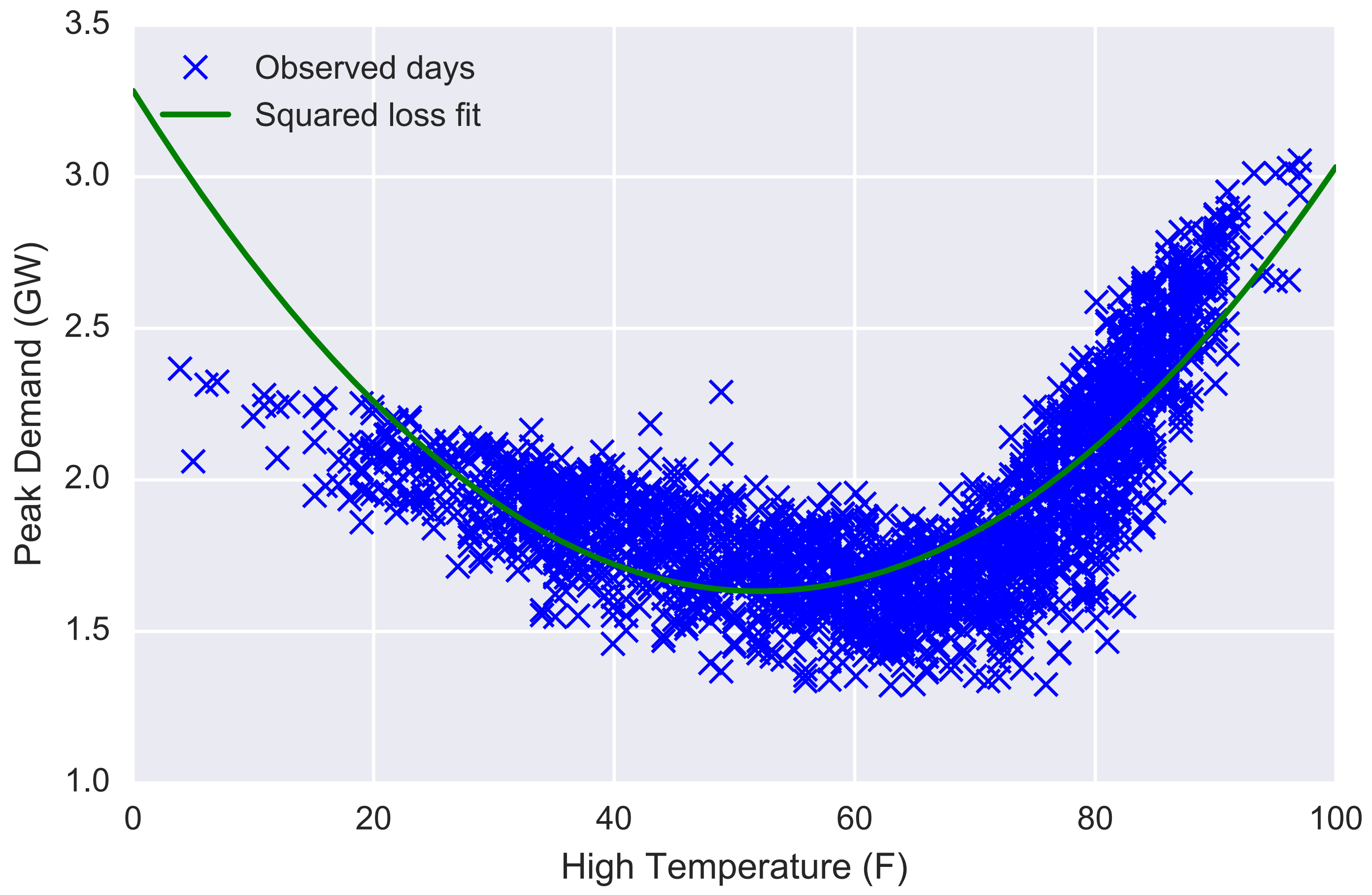
Though it may seem limited, linear regression algorithms are quite powerful when applied to *non-linear features* of the input data, e.g.

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

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

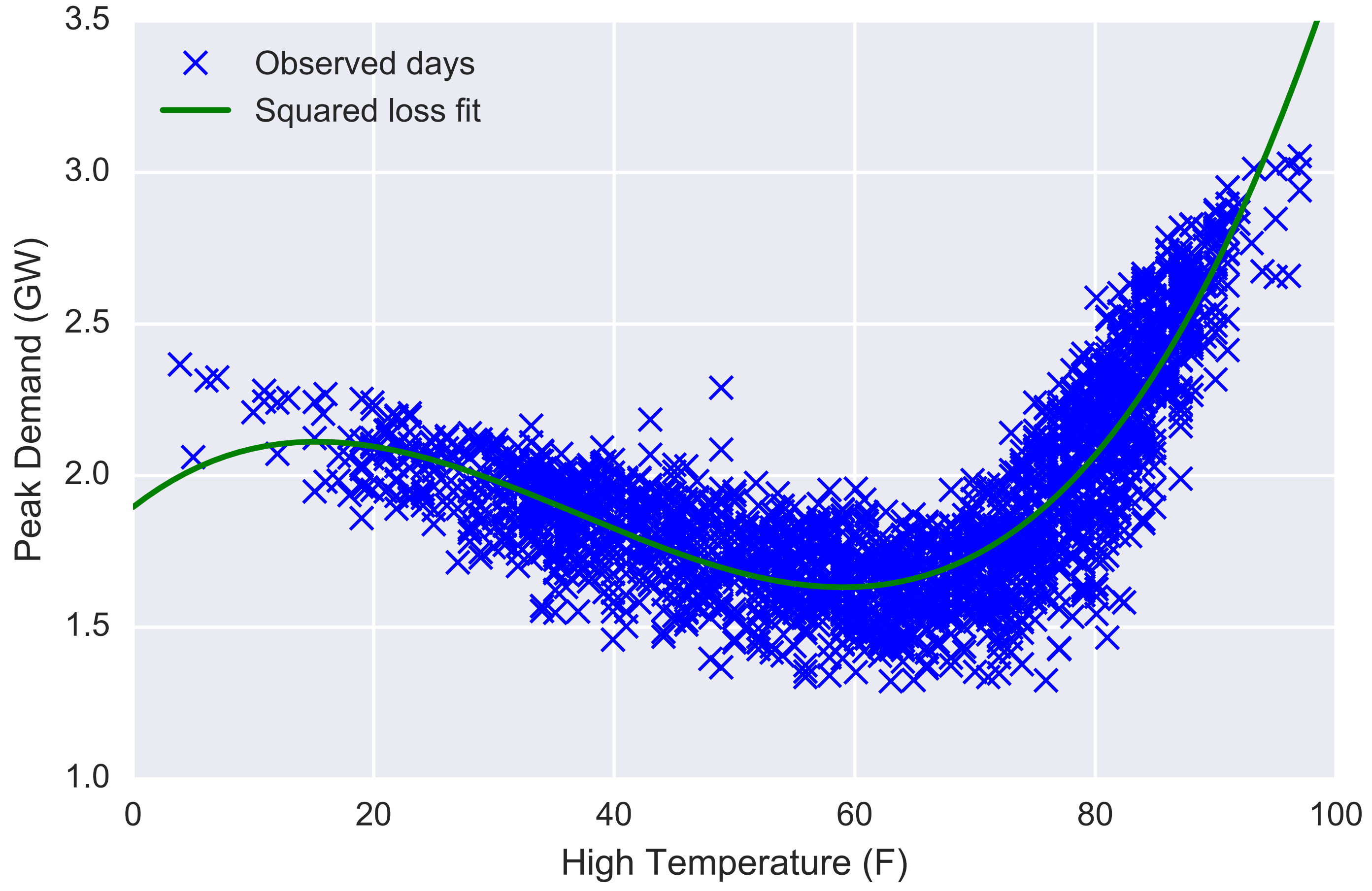
```
x = df_daily.loc[:,"Temperature"]
min_x, rng_x = (np.min(x), np.max(x) - np.min(x))
x = 2*(x - min_x)/rng_x - 1.0
y = df_daily.loc[:,"Load"]

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))
```

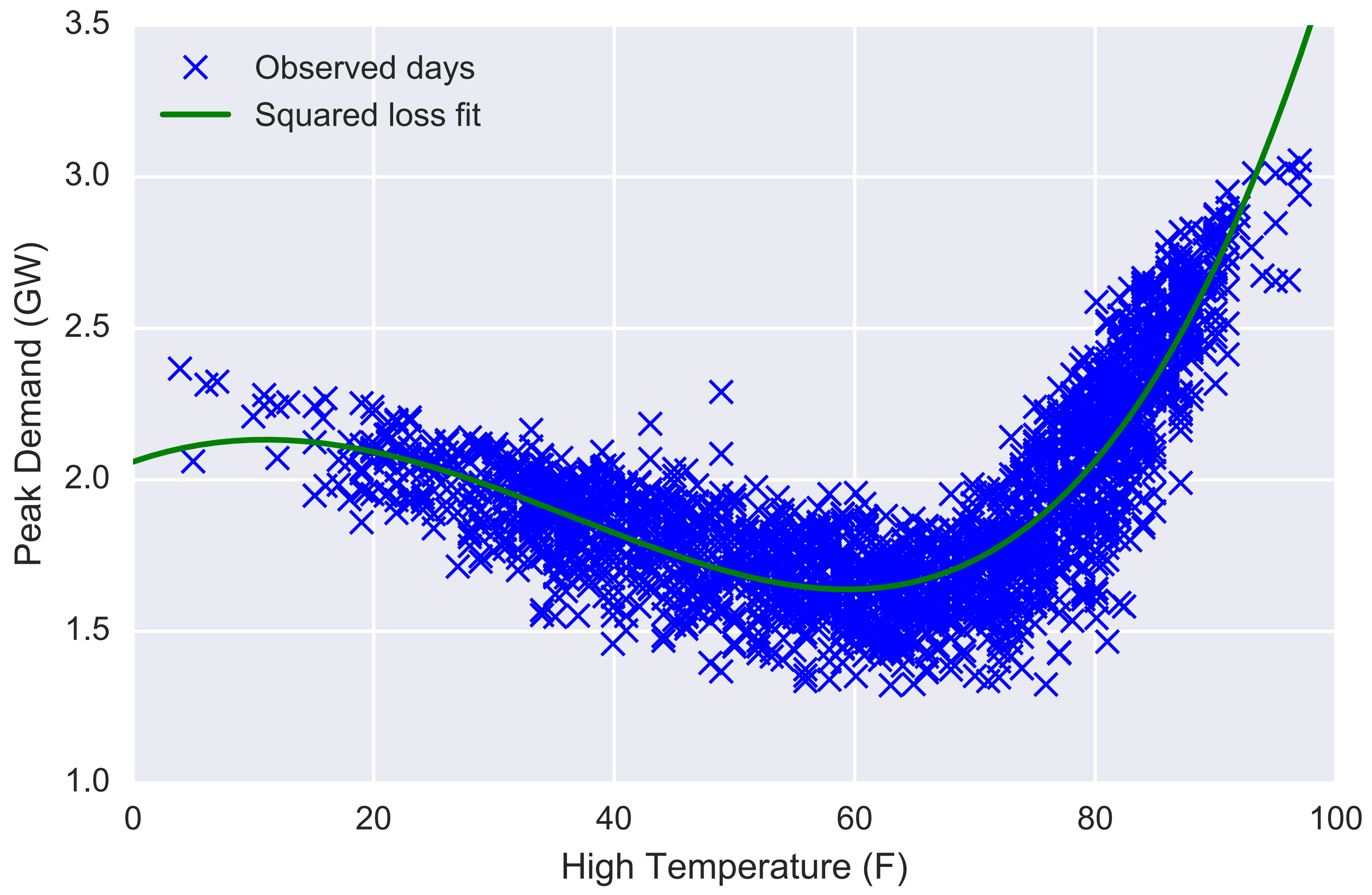
Output learned function:

```
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)
```

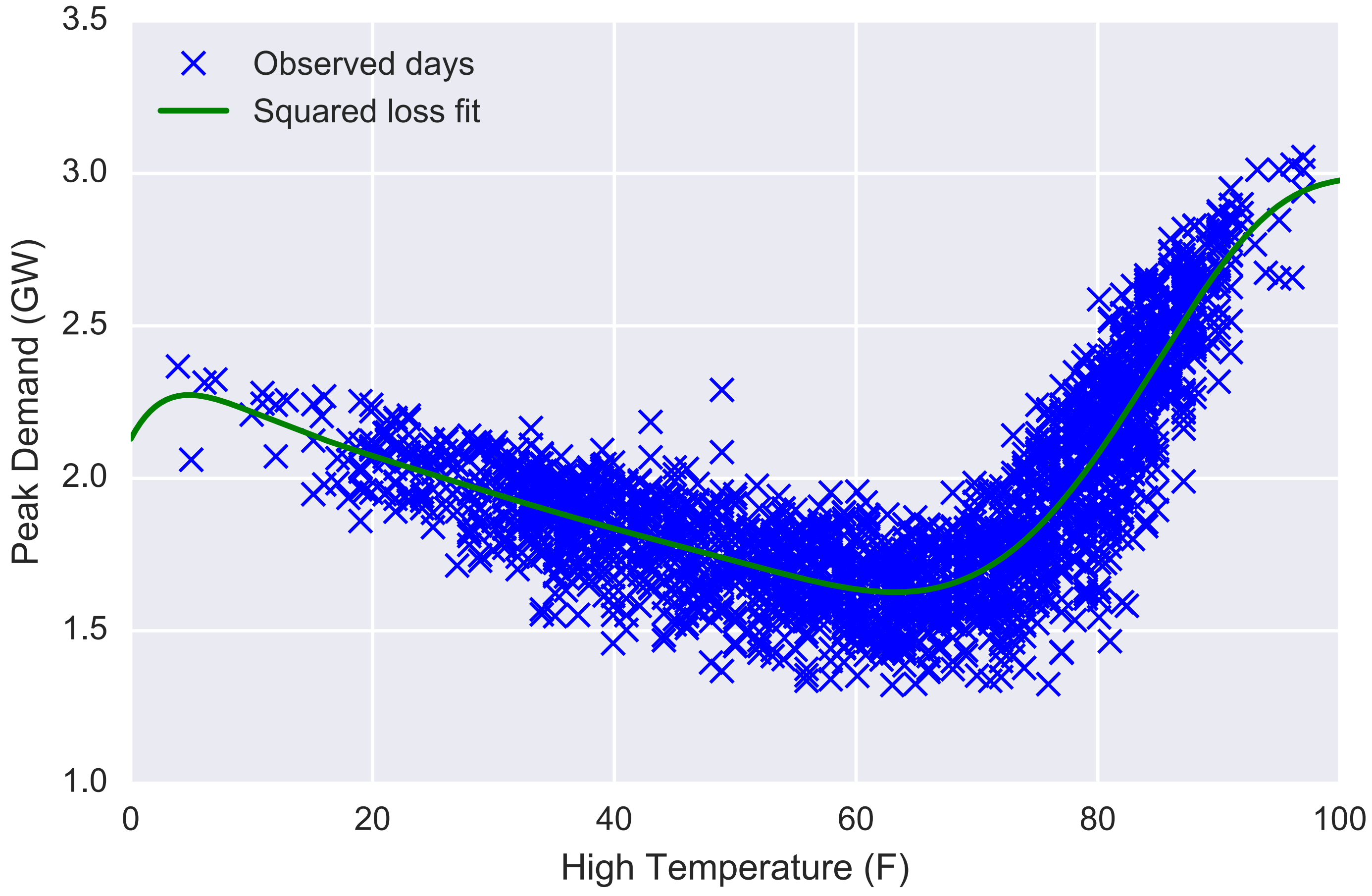
Polynomial features of degree 3



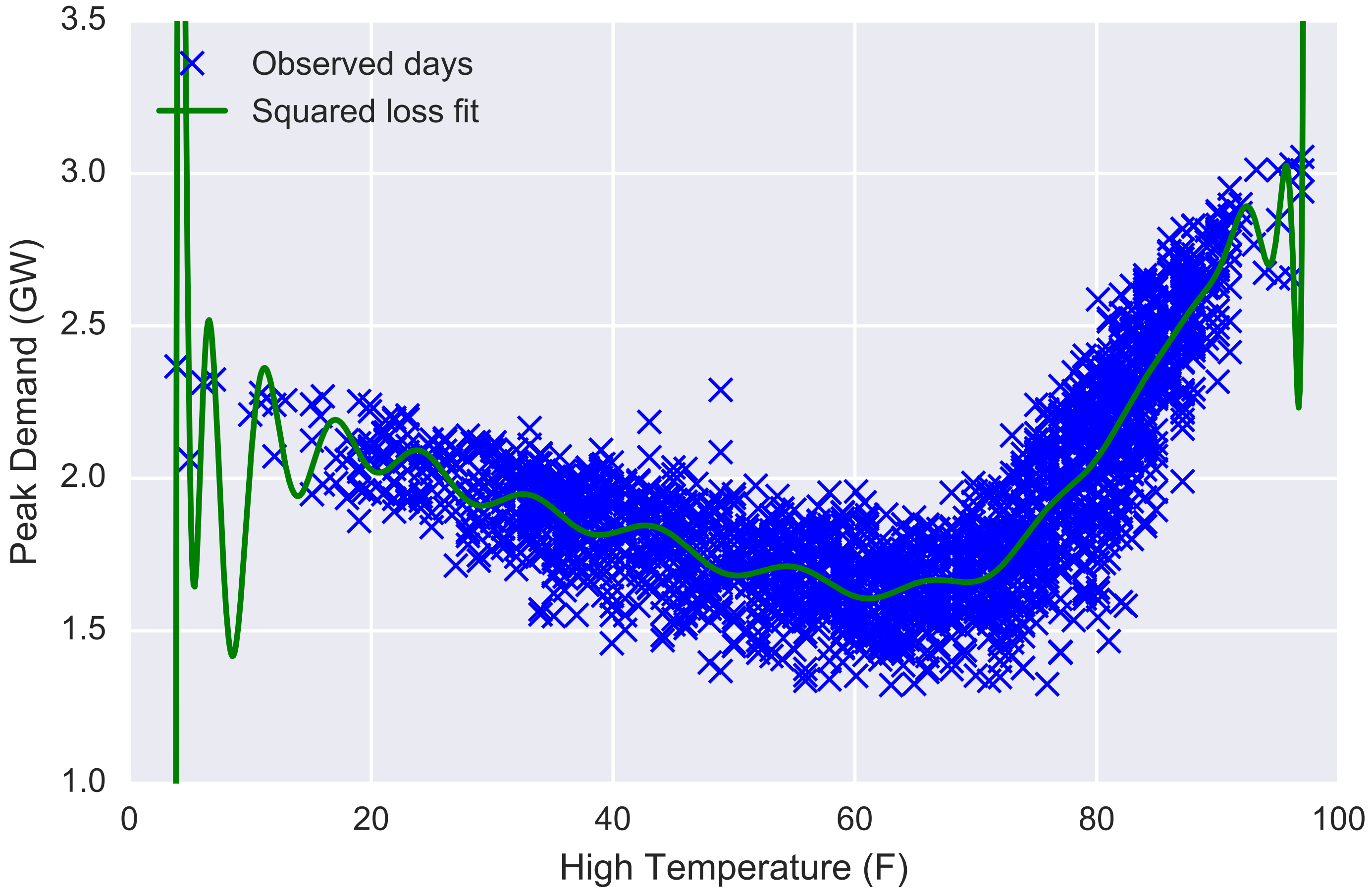
Polynomial features of degree 4



Polynomial features of degree 10



Polynomial features of degree 50



Linear regression with many features

Suppose we have m examples in our data set and $n = m$ features (plus assumption that features are linearly independent, though we'll always assume this)

Then $X \in \mathbb{R}^{m \times n}$ is a square matrix, and least squares solution is:

$$\theta = (X^T X)^{-1} X^T Y = X^{-1} X^{-T} X^T y = X^{-1} y$$

and we therefore have $X\theta = y$ (i.e., we fit data exactly)

Note that we can *only* perform the above operations when X is square, though if we have *more* features than examples, we can still get an exact fit by simply discarding features

Outline

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

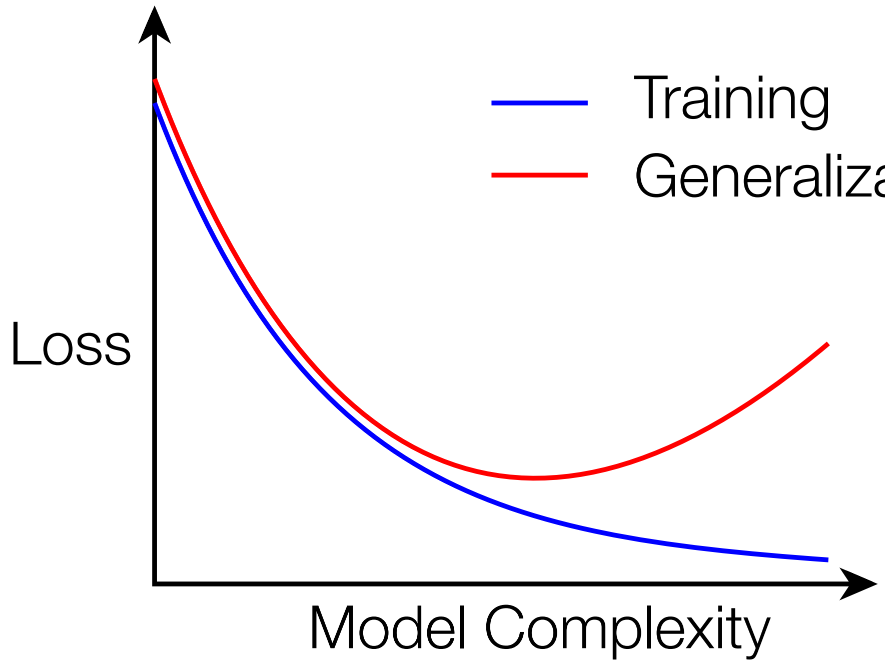
$$\mathrm{minimize}_\theta \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

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

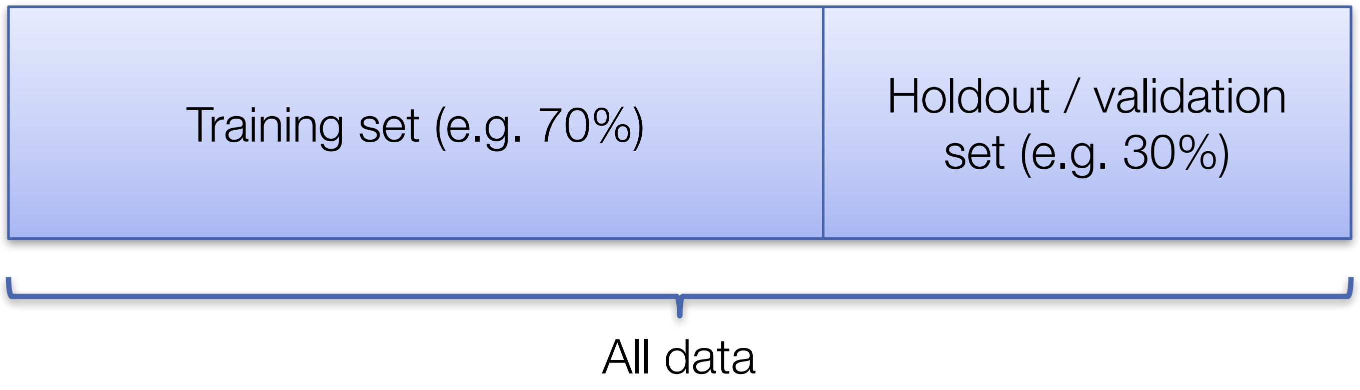
As model becomes more complex, training loss always decreases; generalization loss decreases to a point, then starts to increase



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

Cross-validation in code

A simple example of holdout cross-validation:

```
# compute a random split of the data
np.random.seed(0)
perm = np.random.permutation(len(df_daily))
idx_train = perm[:int(len(perm)*0.7)]
idx_cv = perm[int(len(perm)*0.7):]

# scale features for each split based upon training
xt = df_daily.iloc[idx_train,0]
min_xt, rng_xt = (np.min(xt), np.max(xt) - np.min(xt))
xt = 2*(xt - min_xt)/rng_xt - 1.0
xcv = 2*(df_daily.iloc[idx_cv,0] - min_xt)/rng_xt -1
yt = df_daily.iloc[idx_train,1]
ycv = df_daily.iloc[idx_cv,1]

# compute least squares solution and error on holdout and training
X = np.vstack([xt**i for i in range(poly_degree,-1,-1)]).T
theta = np.linalg.solve(X.T.dot(X), X.T.dot(yt))
err_train = 0.5*np.linalg.norm(X.dot(theta) - yt)**2/len(idx_train)
err_cv = 0.5*np.linalg.norm(Xcv.dot(theta) - ycv)**2/len(idx_cv)
```

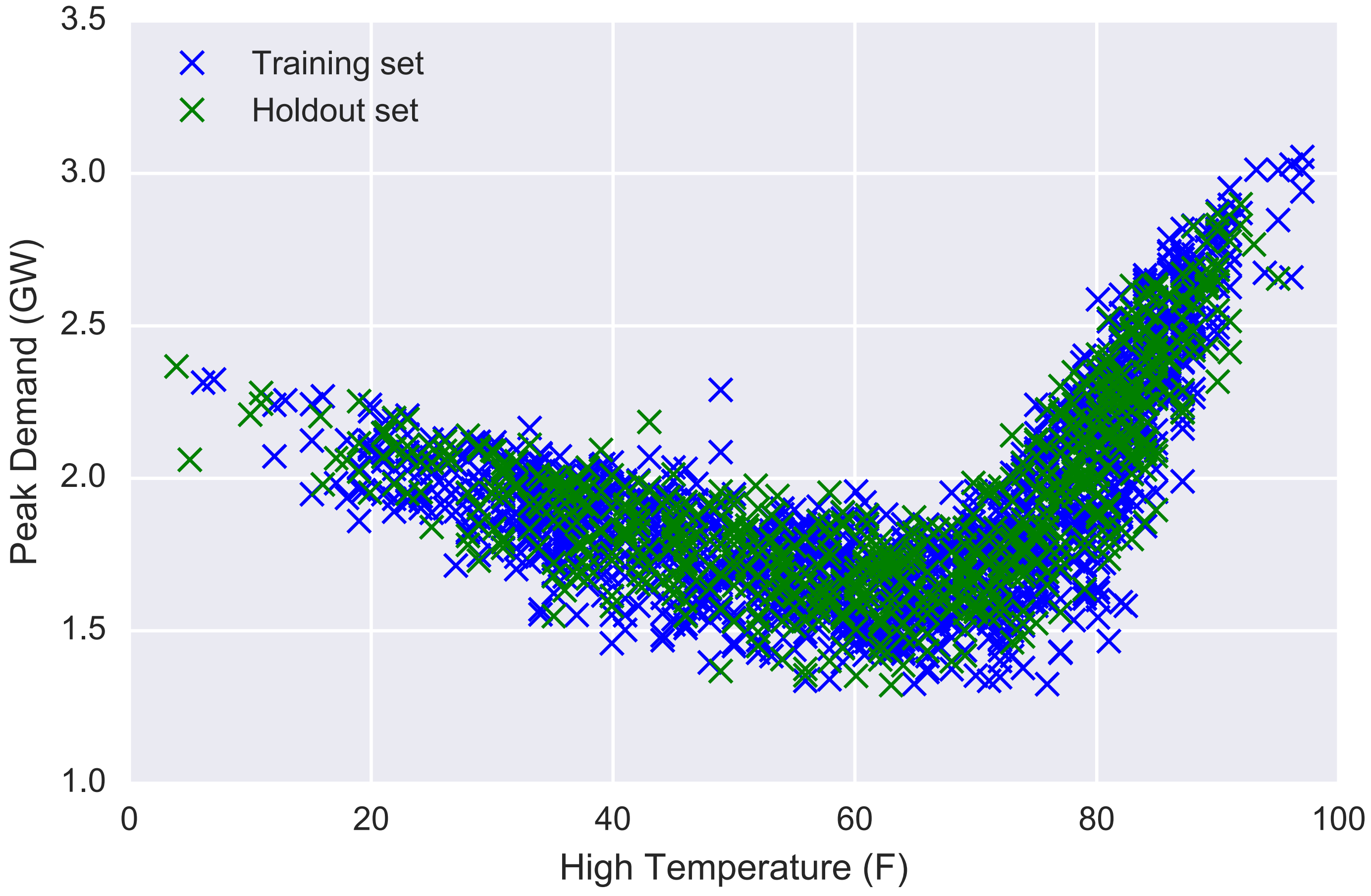
Parameters and hyperparameters

We refer to the θ variables as the *parameters* of the machine learning algorithm

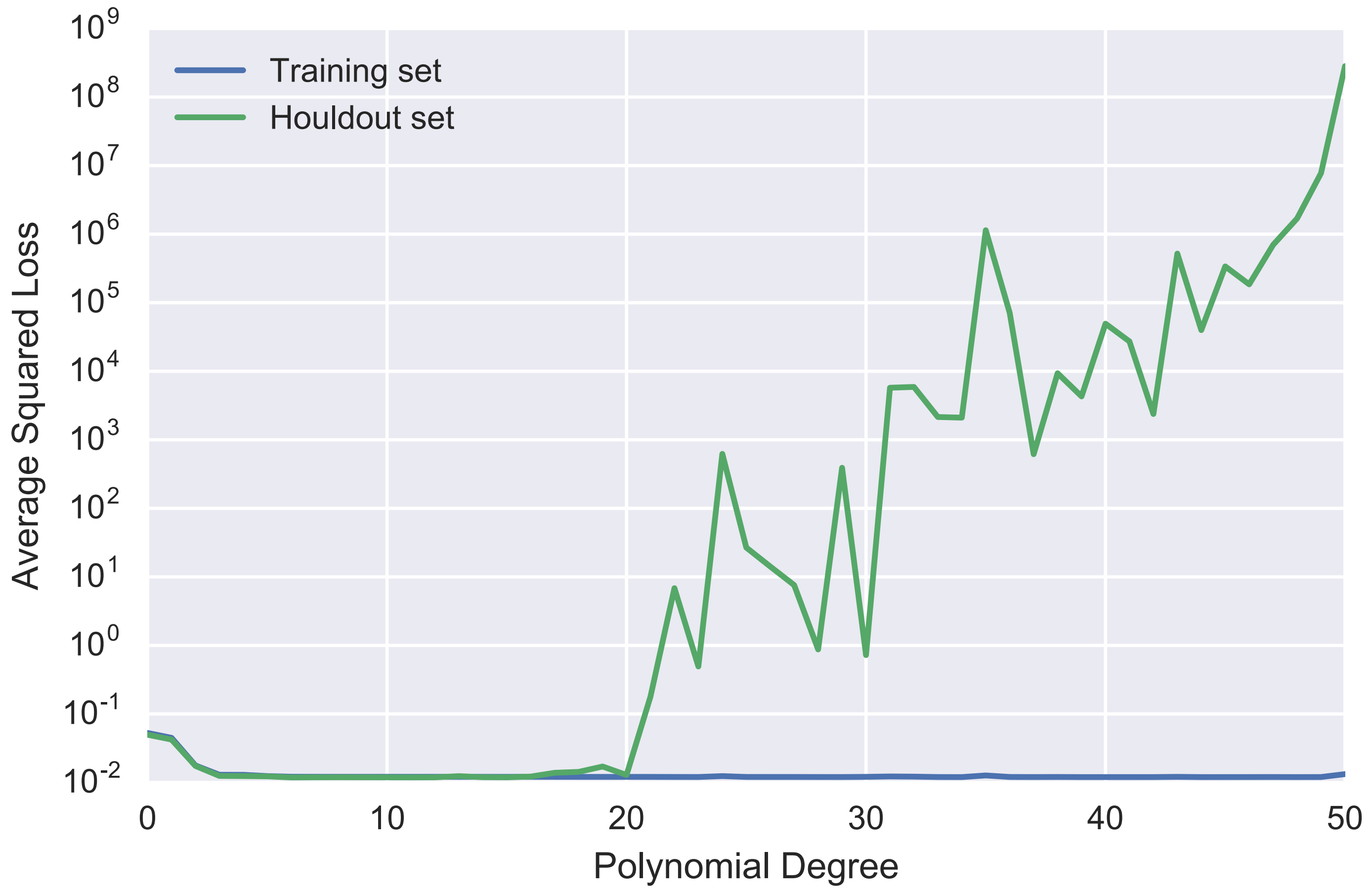
But there are other quantities that also affect the classifier: degree of polynomial, amount of regularization, etc; these are collectively referred to as the *hyperparameters* of the algorithm

Basic idea of cross-validation: use training set to determine the parameters, use holdout set to determine the hyperparameters

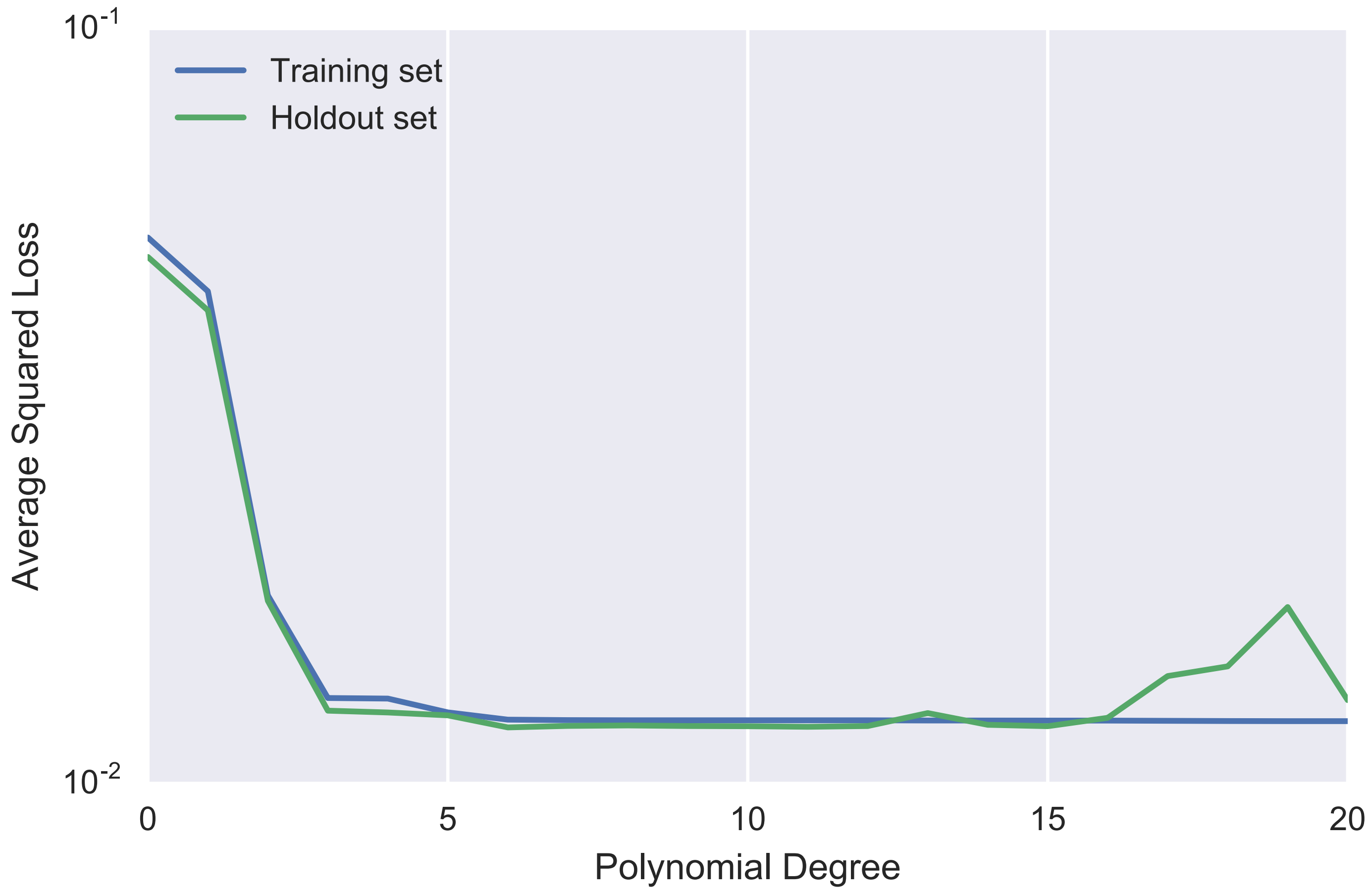
Illustrating cross-validation



Training and cross-validation loss by degree



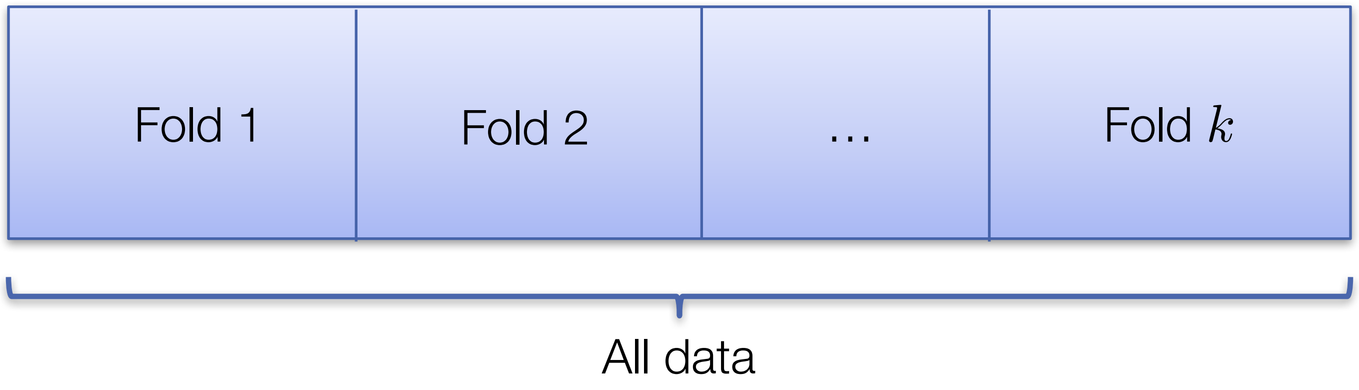
Training and cross-validation loss by degree



K-fold cross-validation

A more involved (but actually slightly more common) version of cross validation

Split data set into k disjoint subsets (folds); train on $k-1$ and evaluate on remaining fold; repeat k times, holding out each fold once



Report average error over all held out folds

Variants

Leave-one-out cross-validation: the limit of k-fold cross-validation, where each fold is only a single example (so we are training on all other examples, testing on that one example)

> [Somewhat surprisingly, for least squares this can be computed *more* efficiently than k-fold cross validation, same complexity solving for the optimal θ using matrix equation]

Stratified cross-validation: keep an approximately equal percentage of positive/negative examples (or any other feature), in each fold

Warning: k-fold cross validation is *not* always better (e.g., in time series prediction, you would want to have holdout set all occur after training set)

Outline

Example: return to peak demand prediction

Overfitting, generalization, and cross validation

Regularization

General nonlinear features

Nonlinear classification

Evaluating machine learning algorithms

Other (classification) metrics

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:

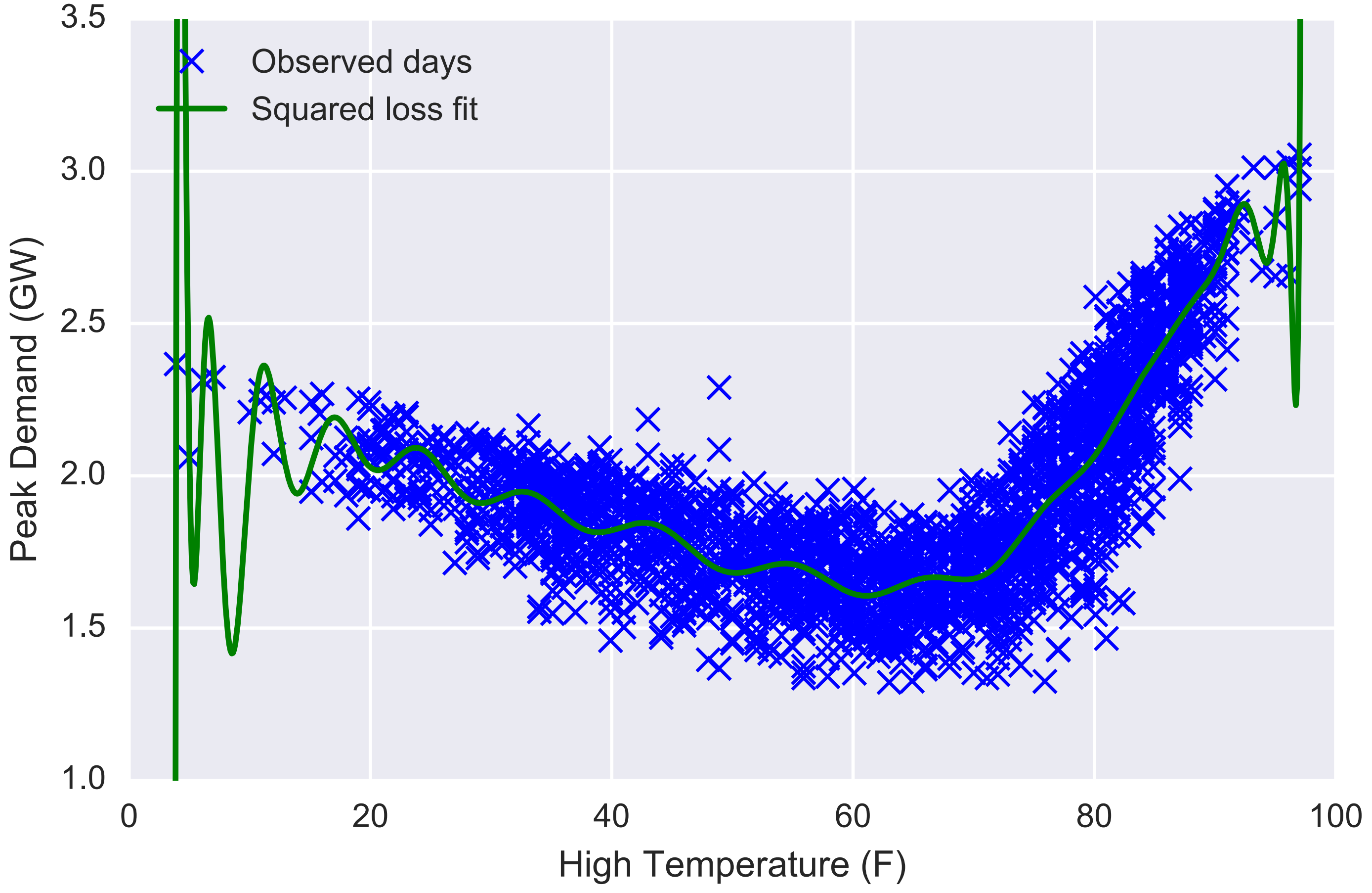
$$\nabla_\theta \left(\frac{1}{2}\|X\theta - y\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2\right) = X^T(X\theta - y) + \lambda\theta$$

Setting gradient equal to zero leads to the solution

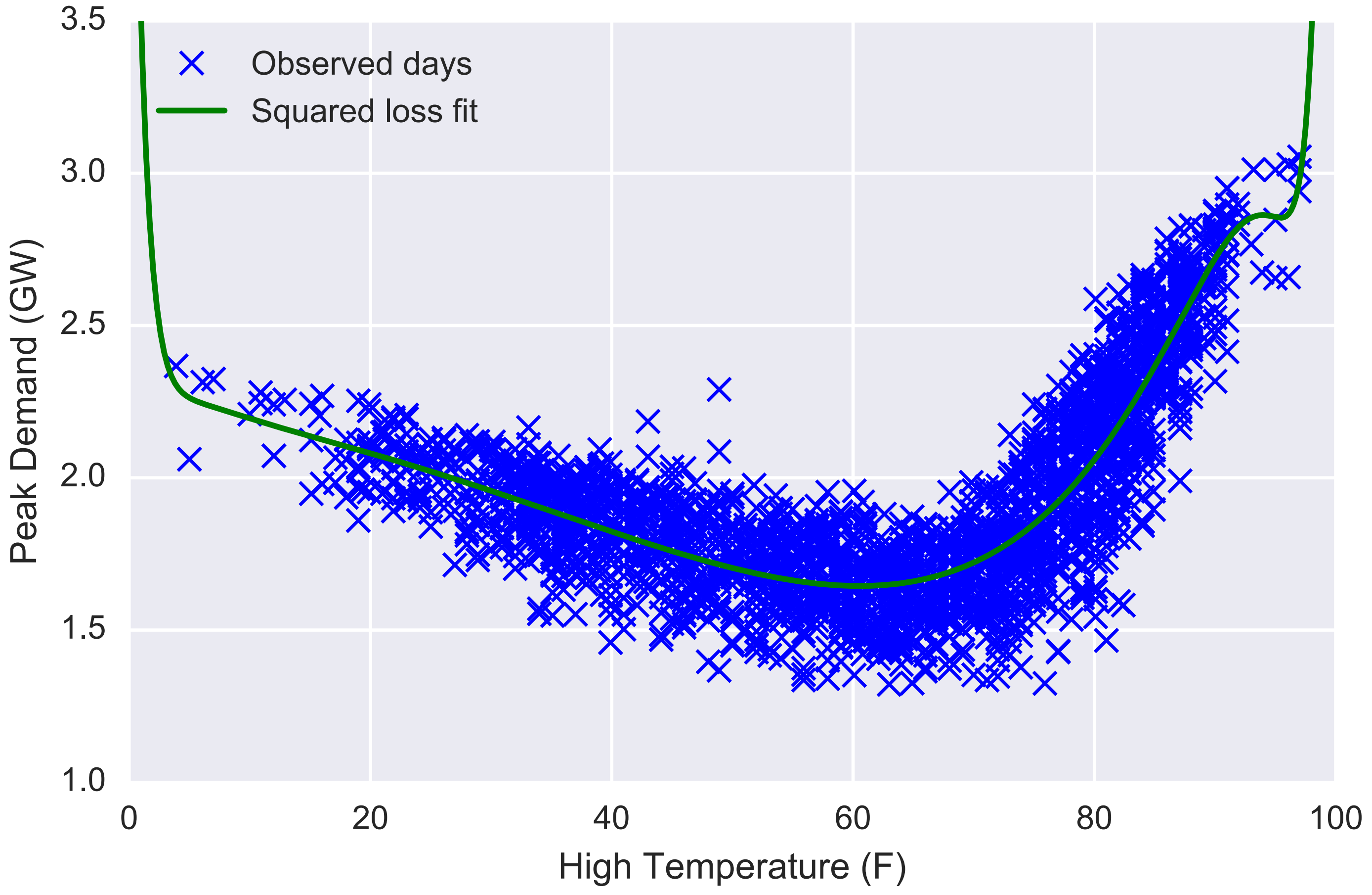
$$X^T X\theta + \lambda\theta = X^T y \implies \theta = (X^T X + \lambda I)^{-1} X^T y$$

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

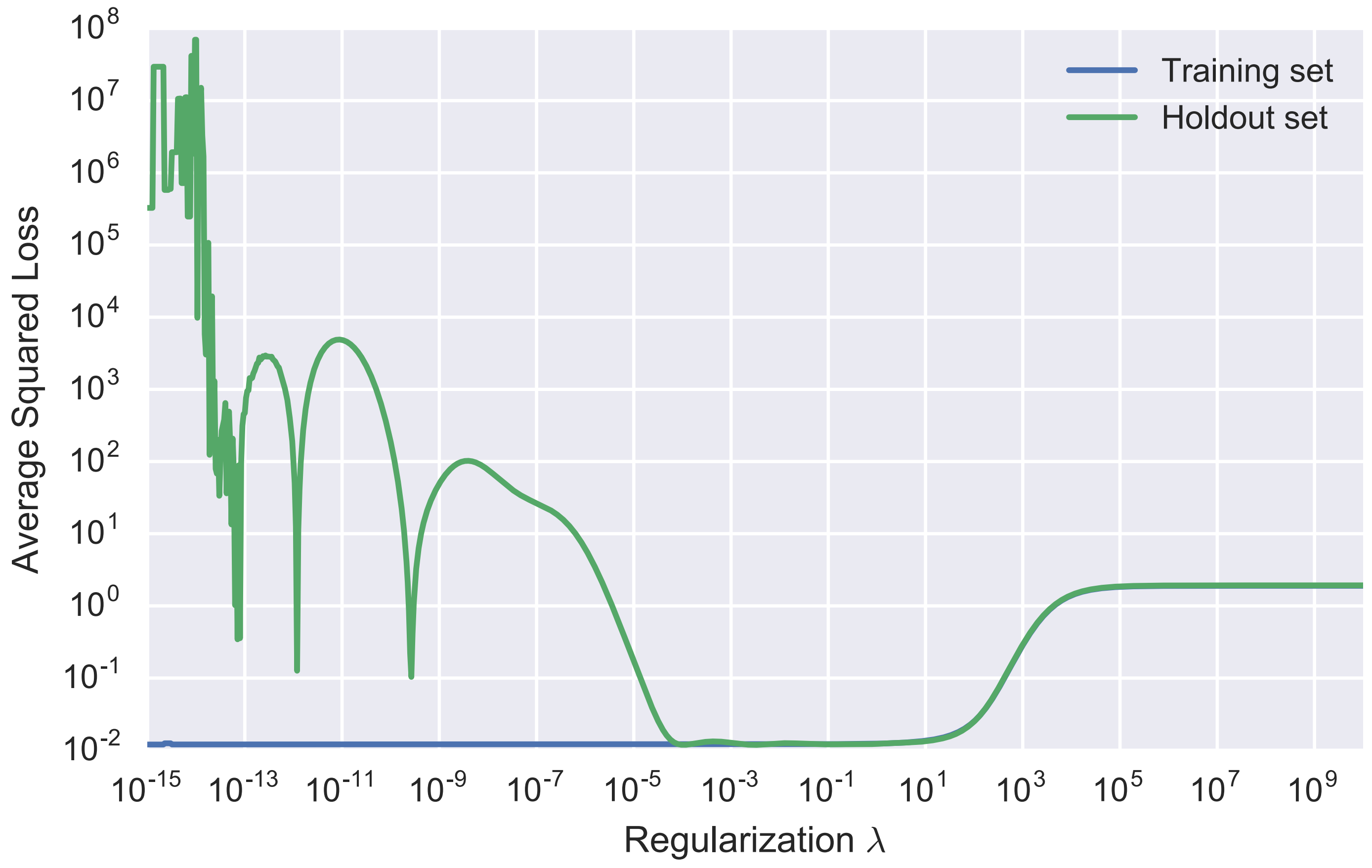
50 degree polynomial fit



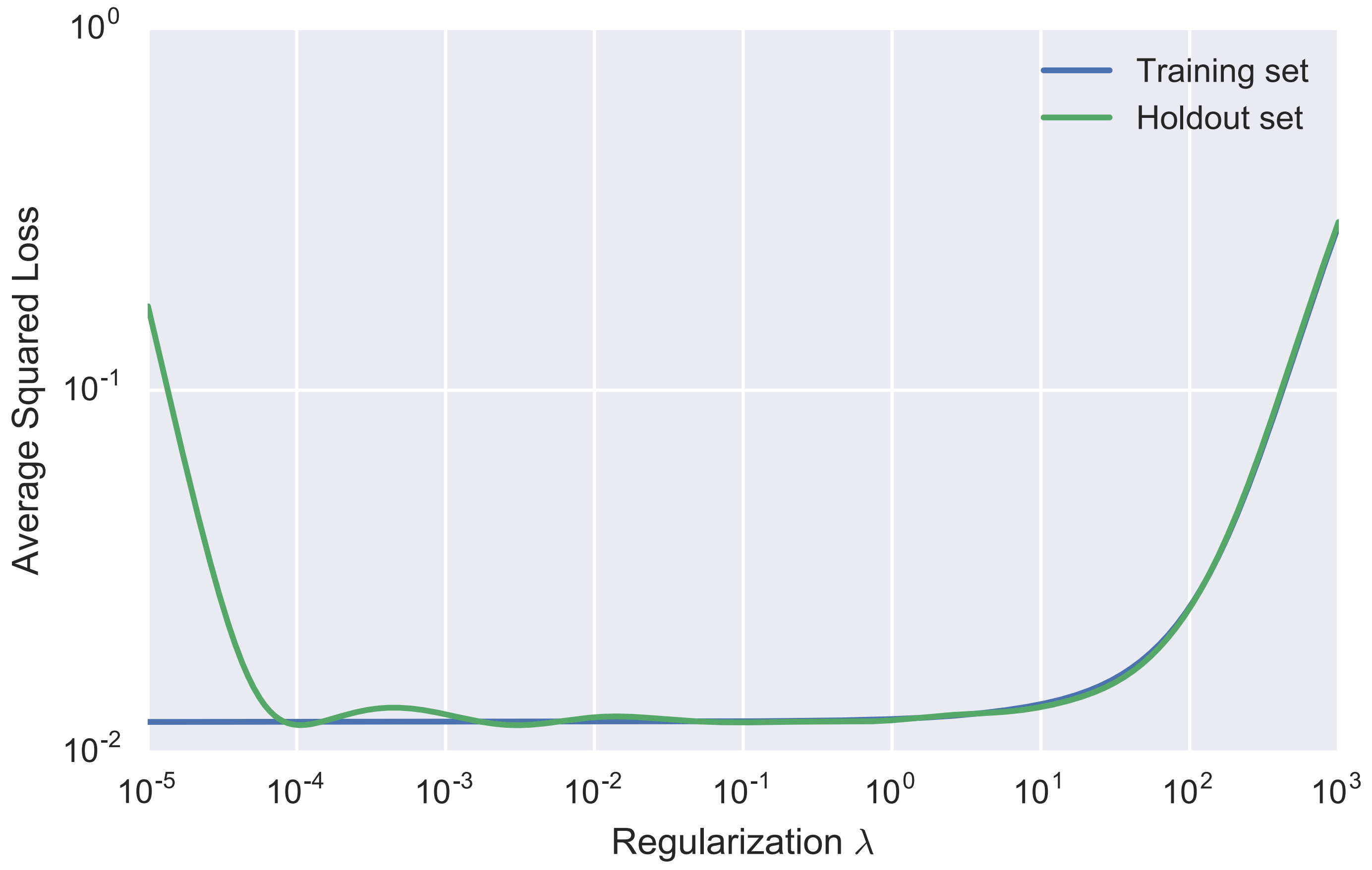
50 degree polynomial fit – $\lambda = 1$



Training/cross-validation loss by regularization



Training/cross-validation loss by regularization



Outline

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)

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

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 d

$$x \in \mathbb{R}^k \Longrightarrow \phi = \left\{ \prod_{i=1}^{k} x_i^{b_i} : \sum_{i=1}^{n} b_i \leq d \right\} \in \mathbb{R}^{\binom{k+d}{k}}$$

Code to generate all polynomial features with degree exactly d:

```
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 d

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

Code for general polynomials

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

```
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

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

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 $X \in \mathbb{R}^{m \times k}$ and matrix of centers $\mu \in \mathbb{R}^{p \times k}$

```
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)*

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

Outline

Nonlinear classification

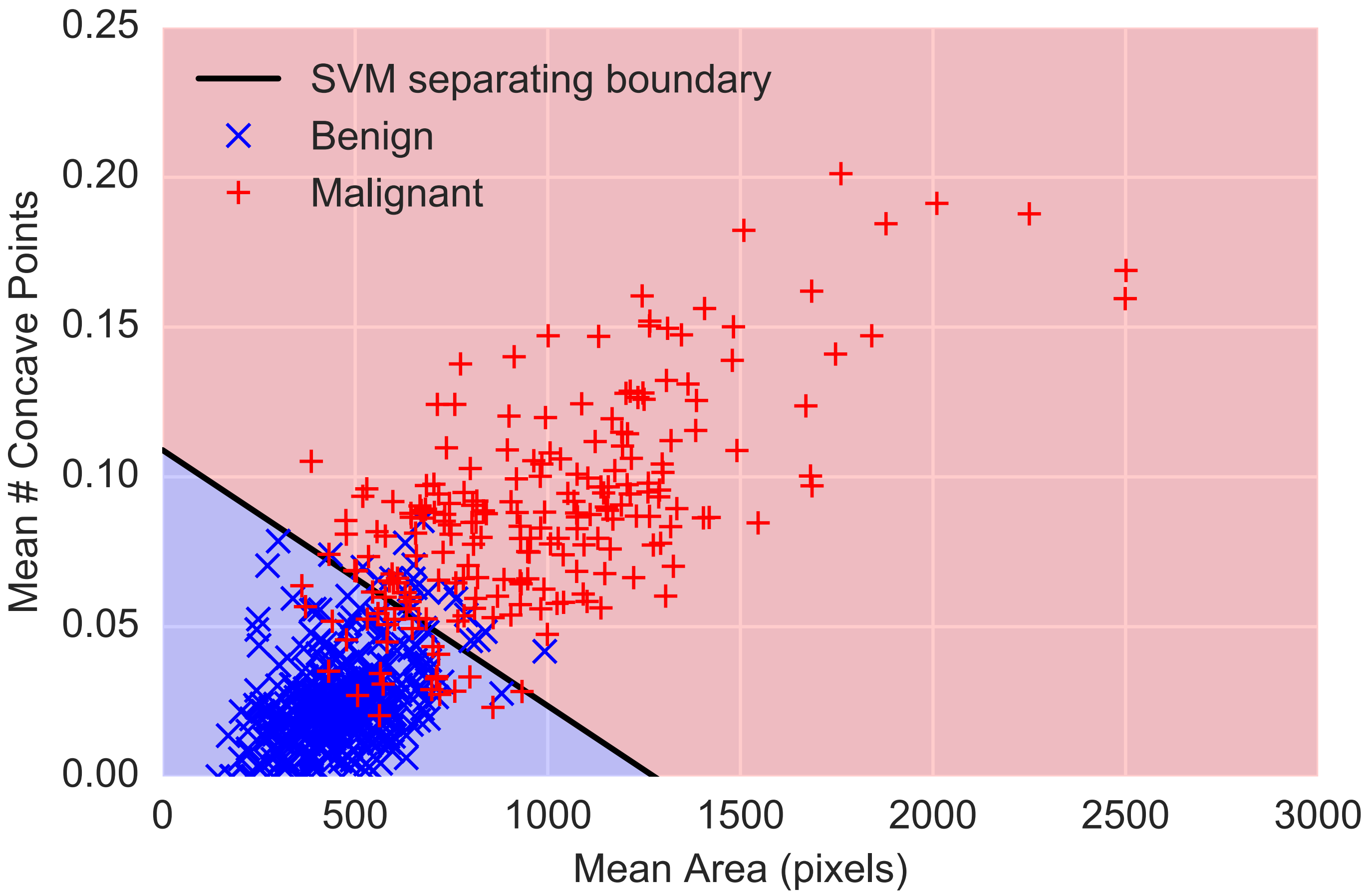
Just like linear regression, the nice thing about using nonlinear features for classification is that our algorithms remain exactly the same as before

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

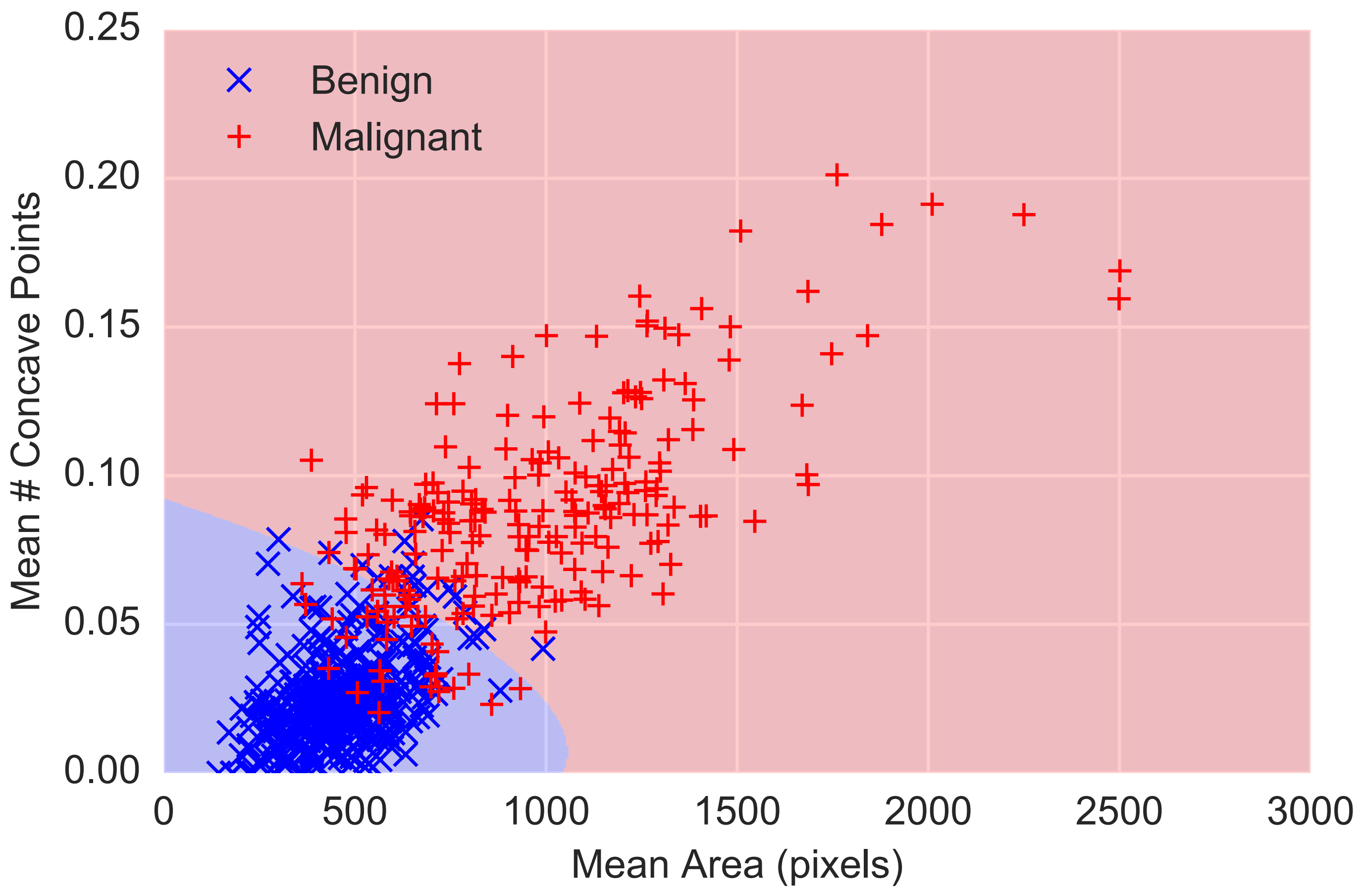
$$\mathrm{minimize}_\theta \quad \sum_{i=1}^{m} \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\} + \frac{\lambda}{2}\|\theta\|_2^2$$

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

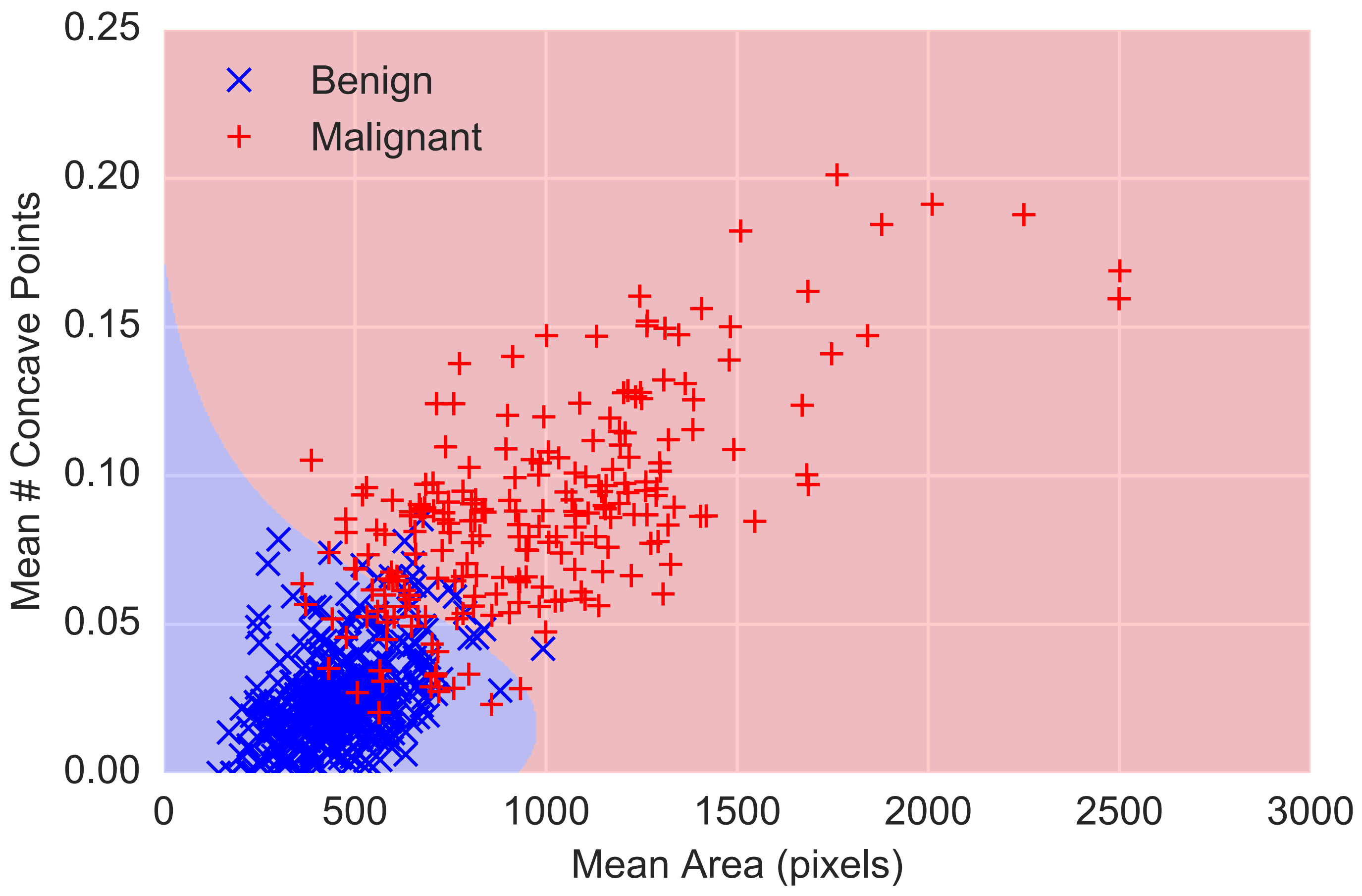
Linear SVM on cancer data set



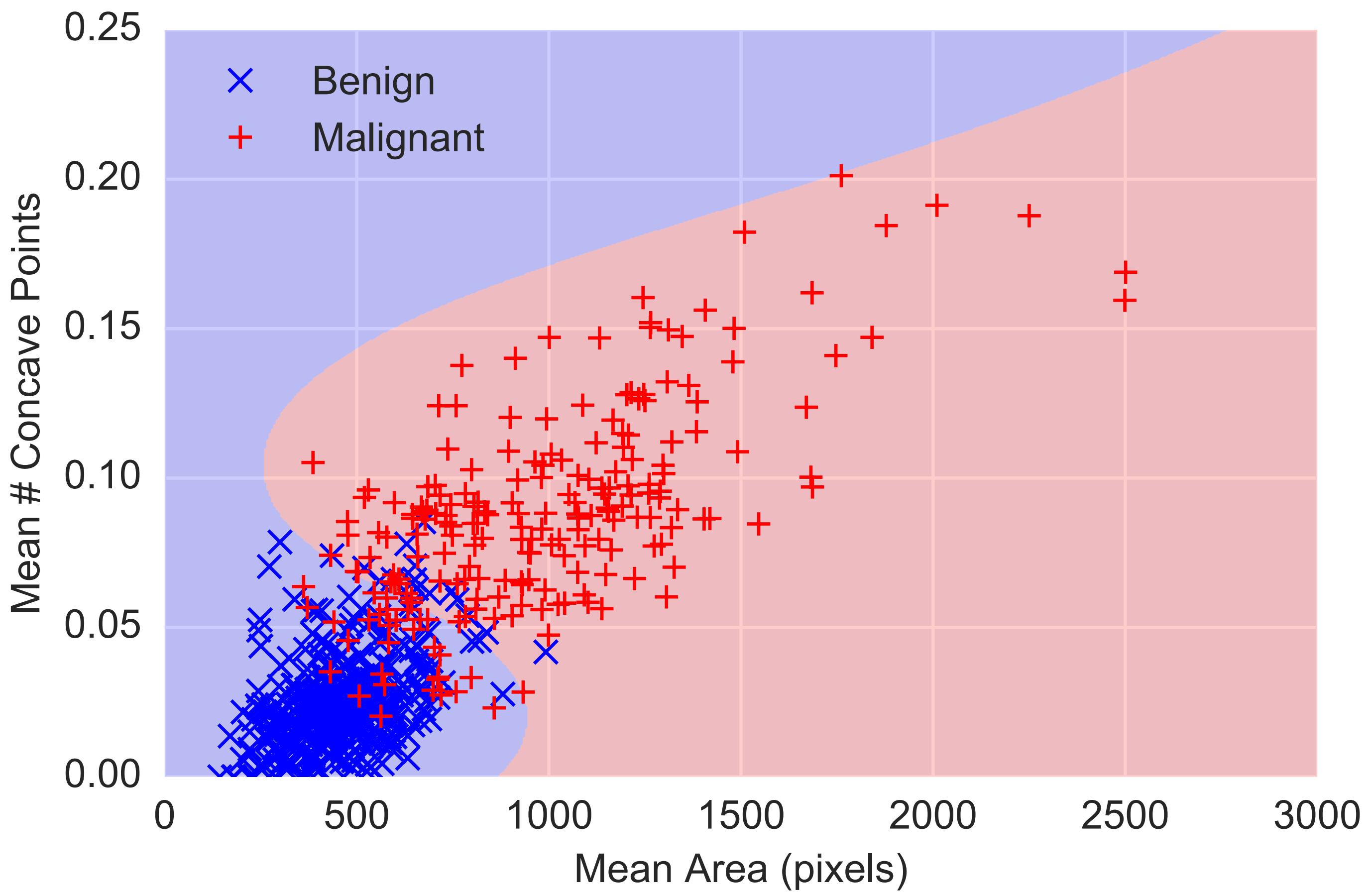
Polynomial features $d = 2$



Polynomial features $d = 3$



Polynomial features $d = 10$



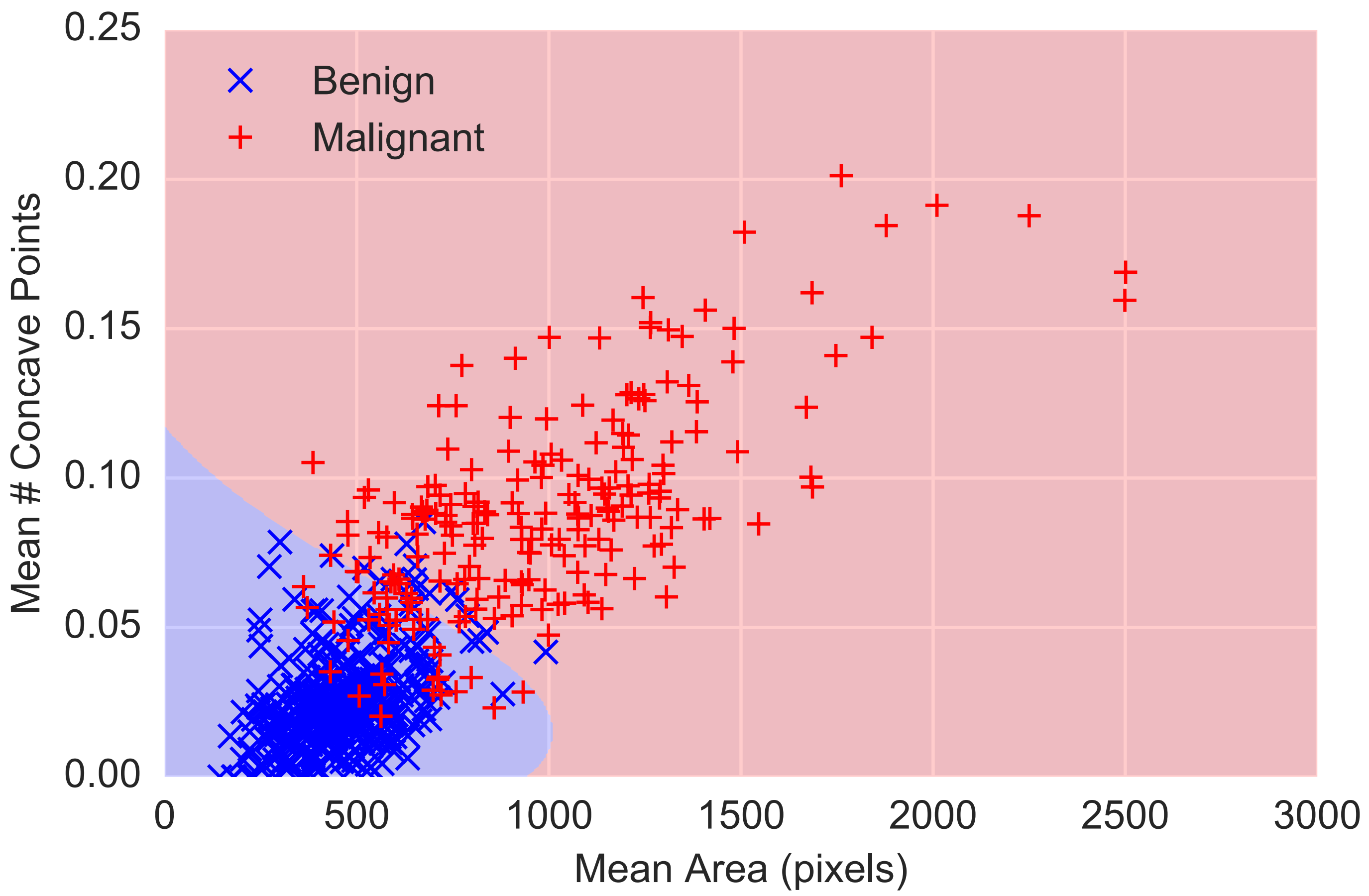
RBF features

Below, we assume that X has been normalized so that each feature lies between $[-1, +1]$ (same as we did for polynomial features)

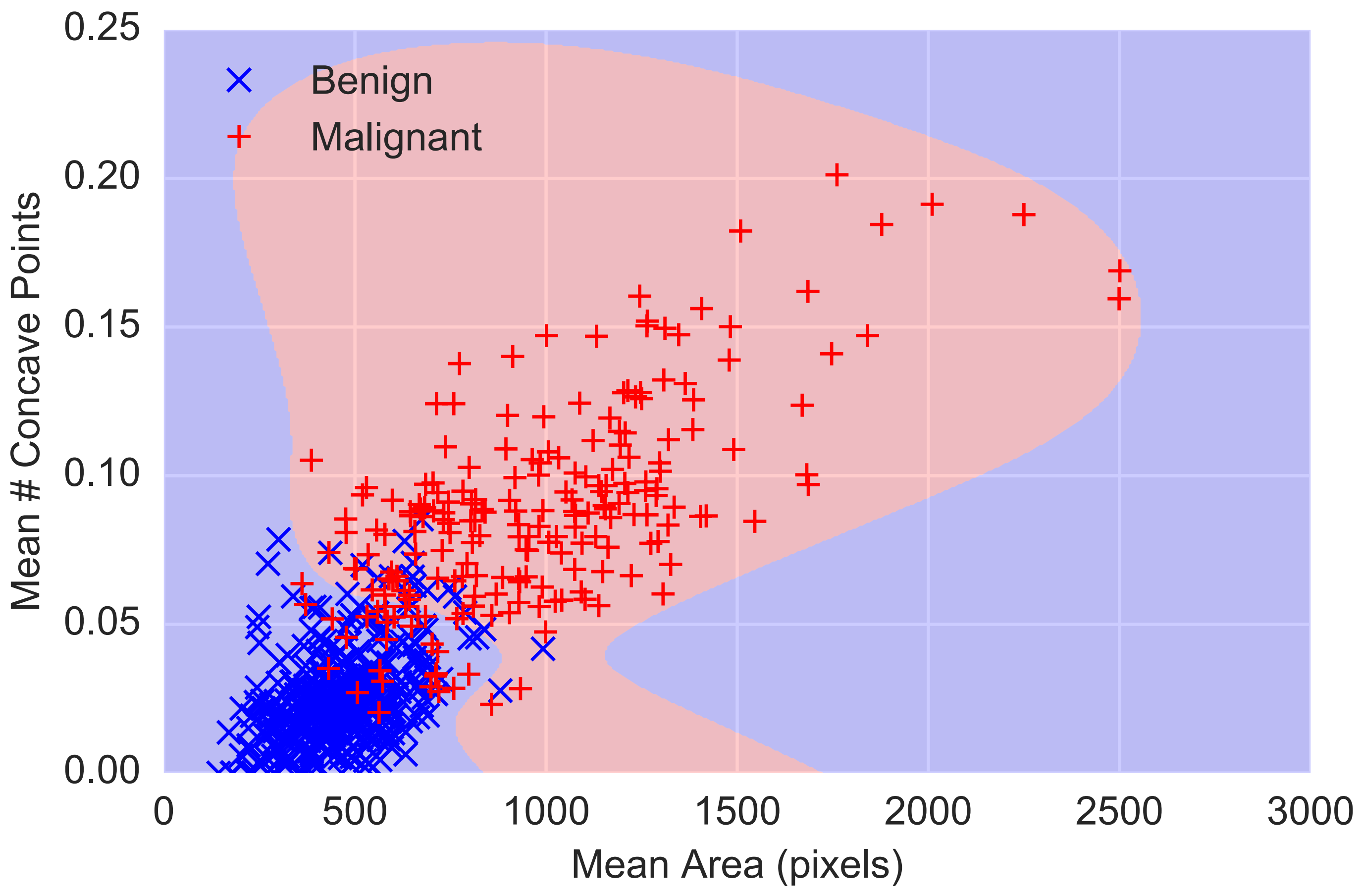
We're consider to observe how the classifier changes as we change different parameters of the RBFs

p will refer to total number of centers, d will refer the number of centers along each dimensions, assuming centers form a regular grid (so since we have two raw inputs, $p = d^2$)

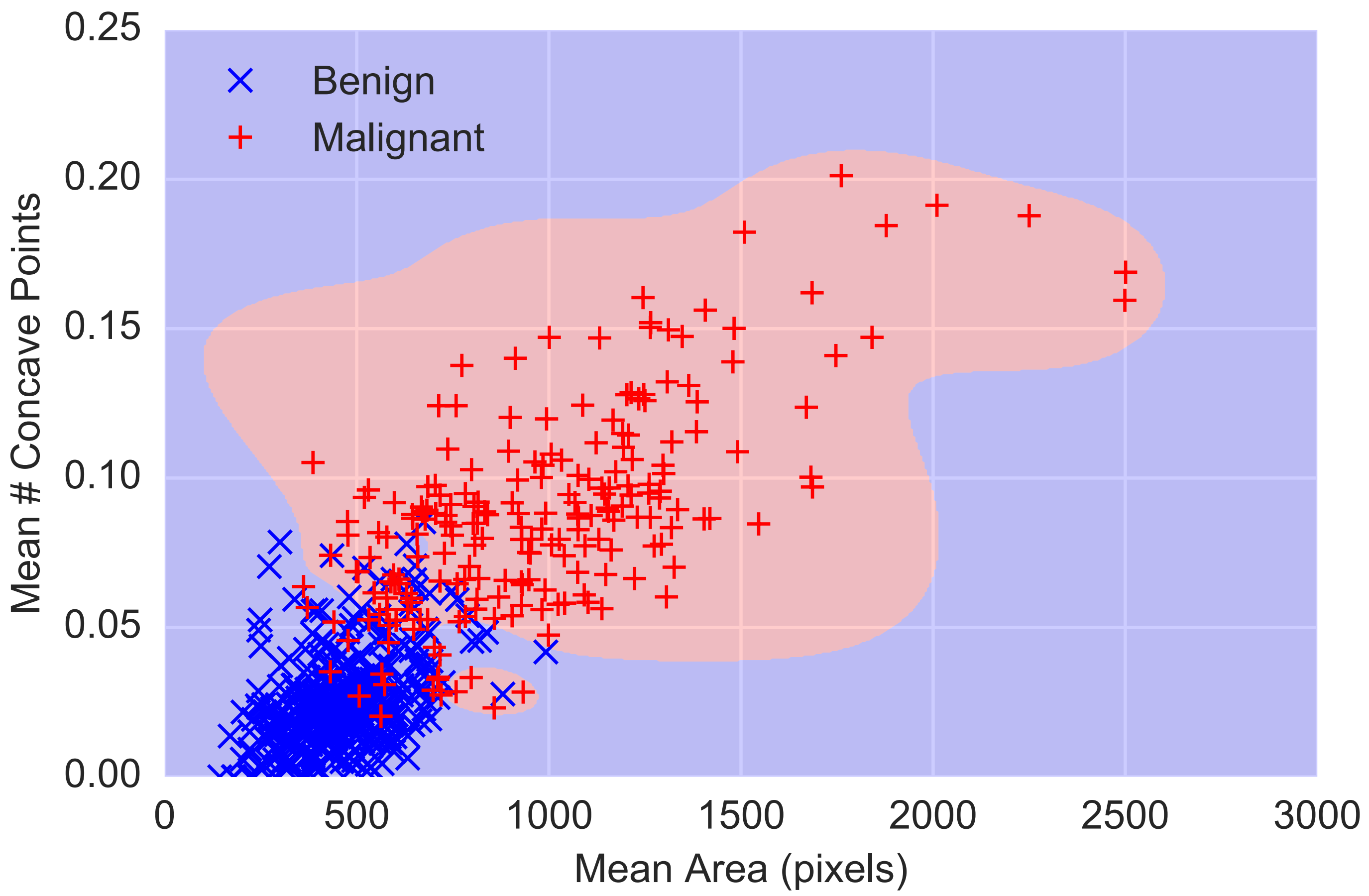
RBF features, $d = 3, \sigma = 2/d$



RBF features, $d = 10, \sigma = 2/d$



RBF features, $d = 20, \sigma = 2/d$

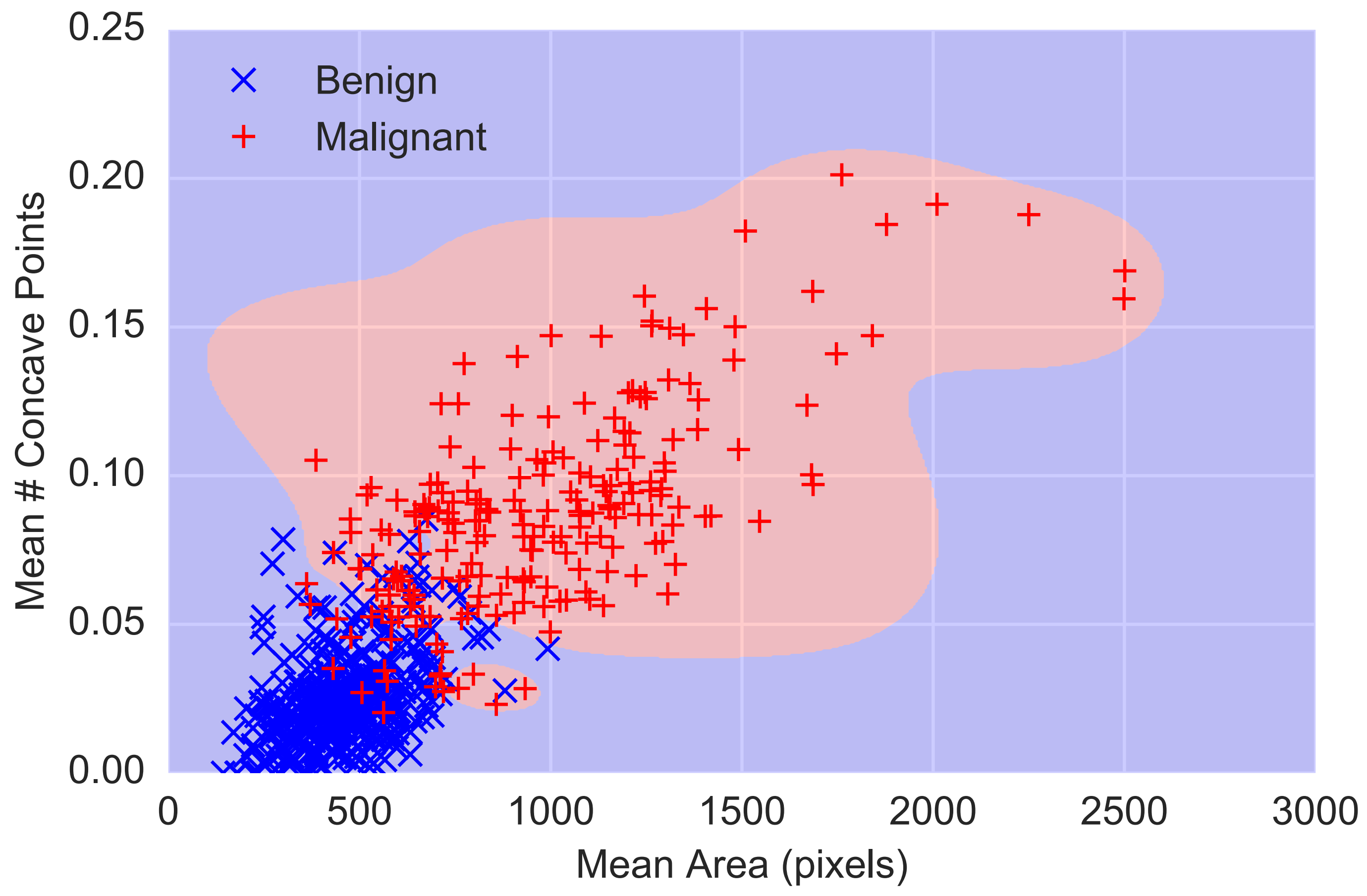


Model complexity and bandwidth

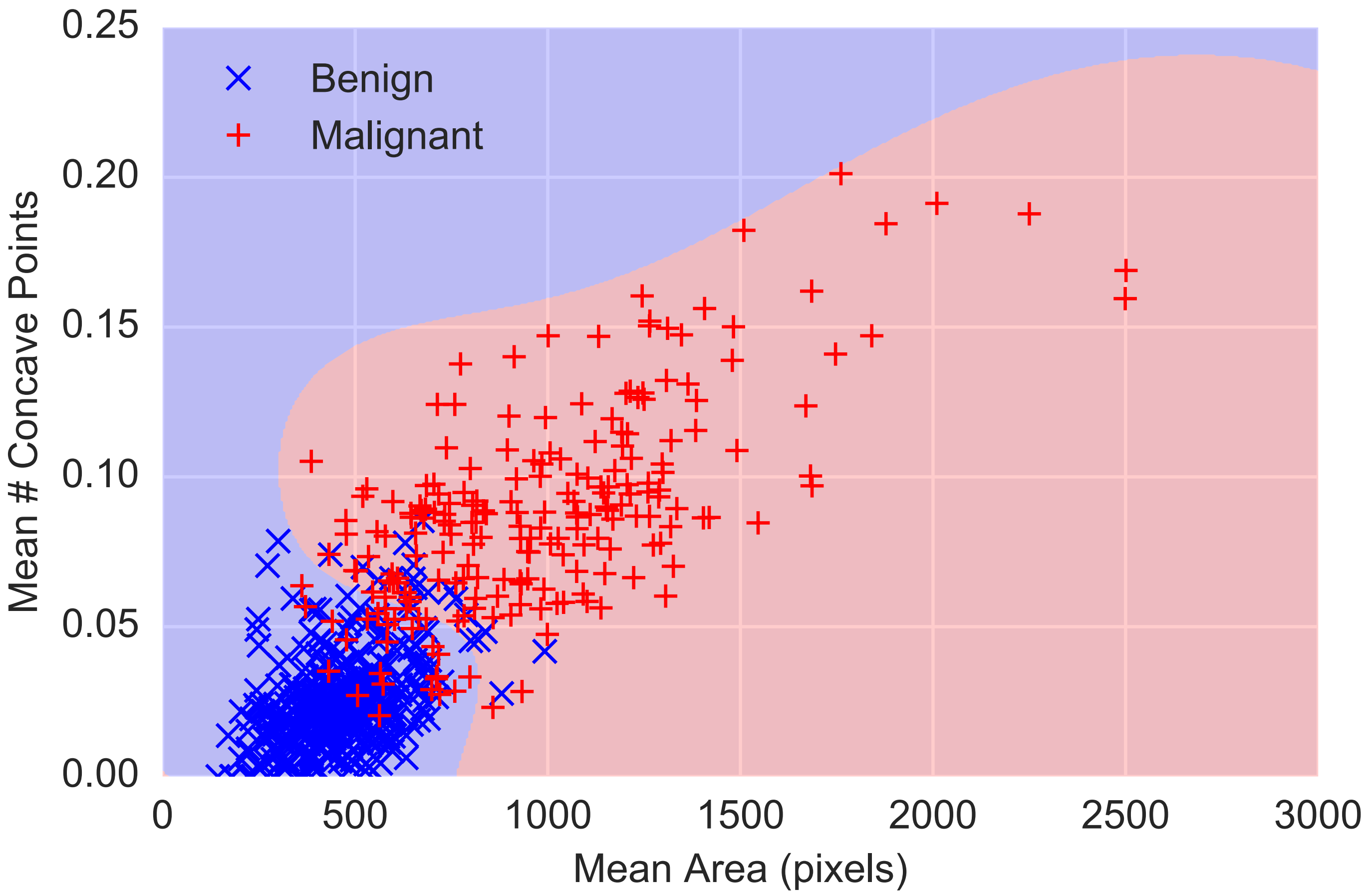
We can control model complexity with RBFs in three ways: two of which we have already seen

1. Choose number of RBF centers
2. Increase/decrease regularization parameter
3. Increase/decrease bandwidth

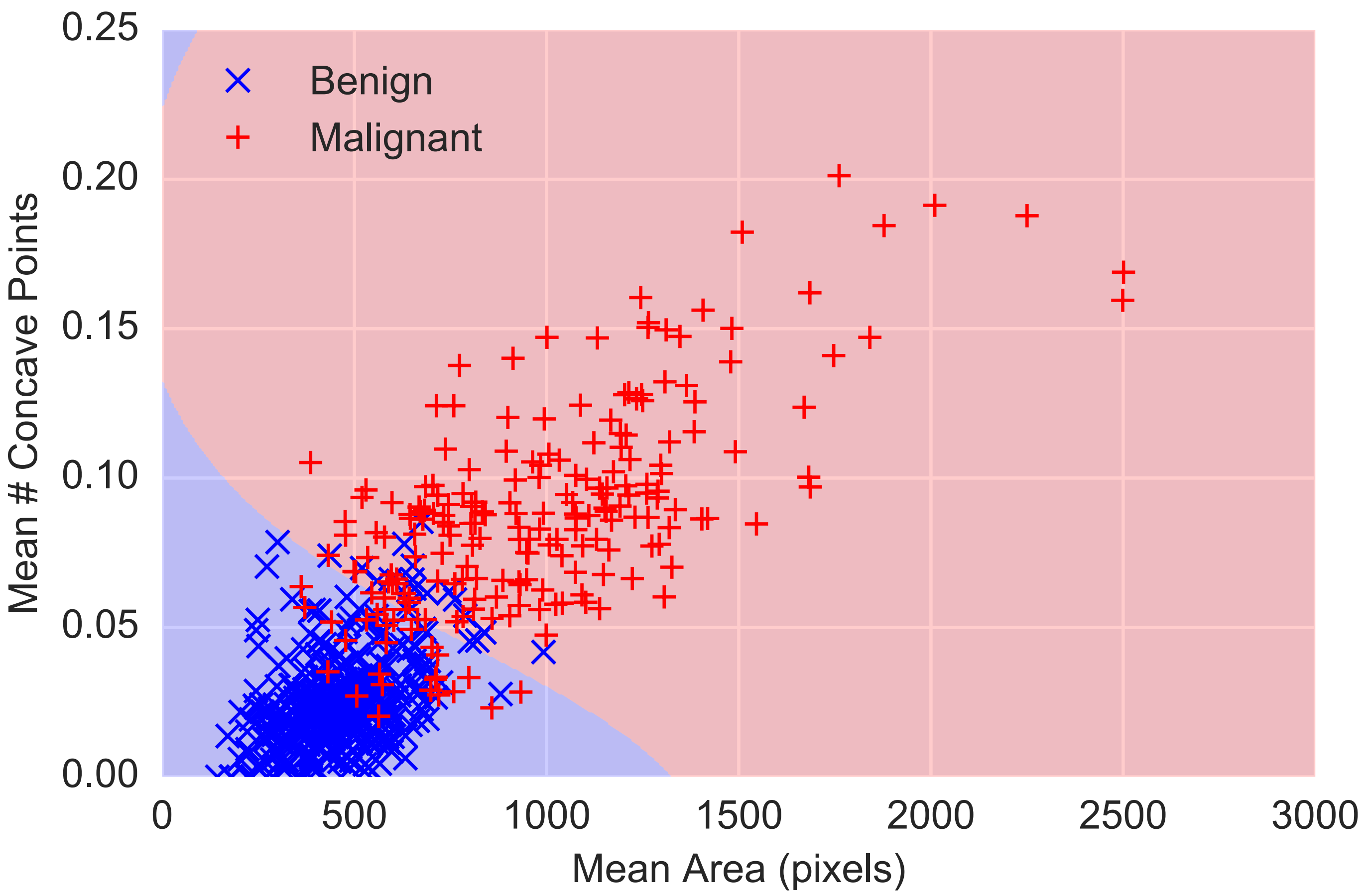
RBF features, $d = 20, \sigma = 0.1$



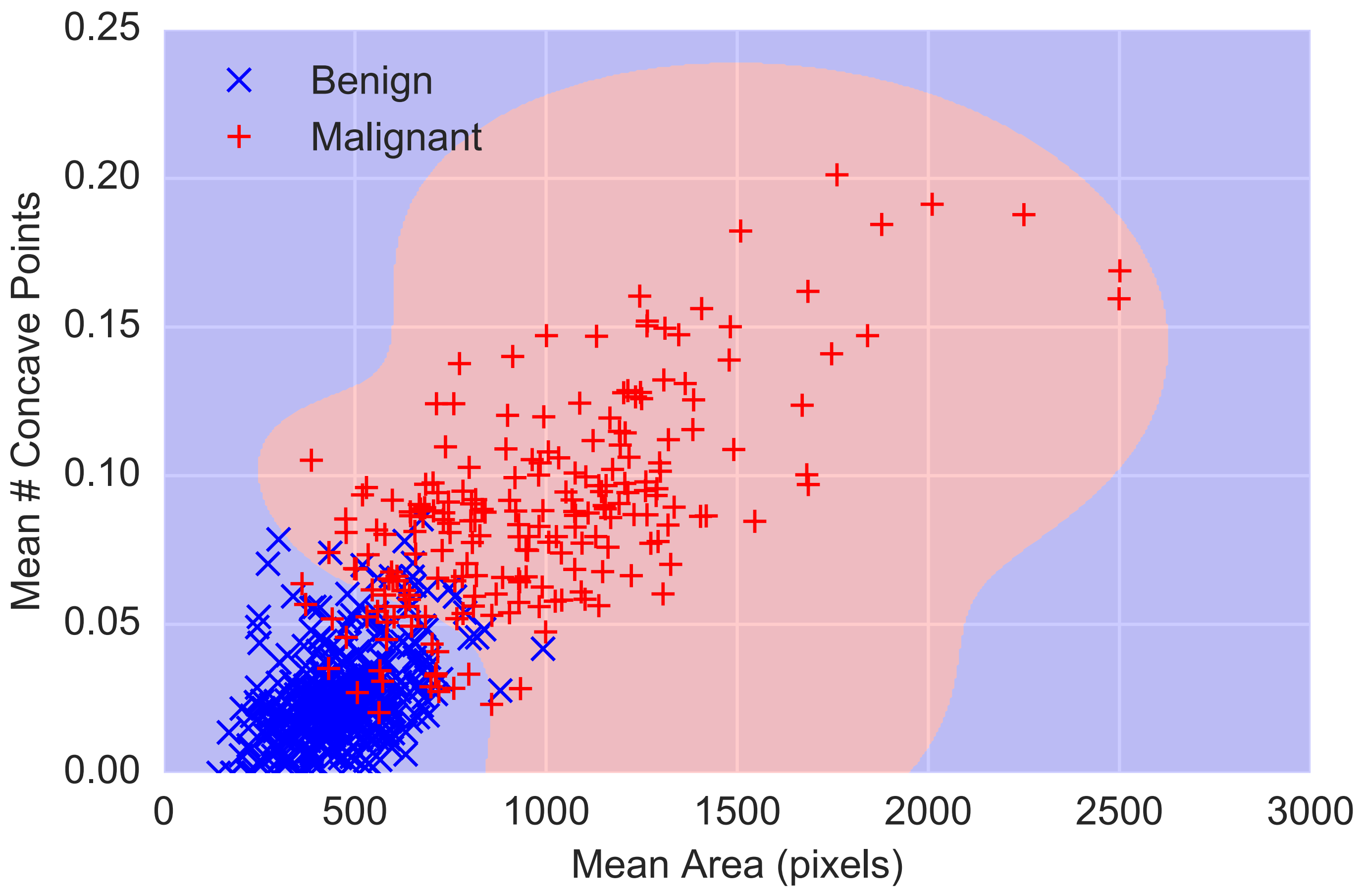
RBF features, $d = 20, \sigma = 0.5$



RBF features, $d = 20, \sigma = 1.07$ (median trick)



RBFs from data, $p = 50, \sigma = \text{median} - \text{trick}$



Outline

Example: return to peak demand prediction

Overfitting, generalization, and cross validation

Regularization

General nonlinear features

Nonlinear classification

Evaluating machine learning algorithms

Other (classification) metrics

A common strategy for evaluating algorithms

1. Divide data set into training and holdout sets

2. Train different algorithms (or a single algorithm with different hyperparameter settings) using the training set

3. Evaluate performance of all the algorithms on the holdout set, and report the best performance (e.g., lowest holdout error)

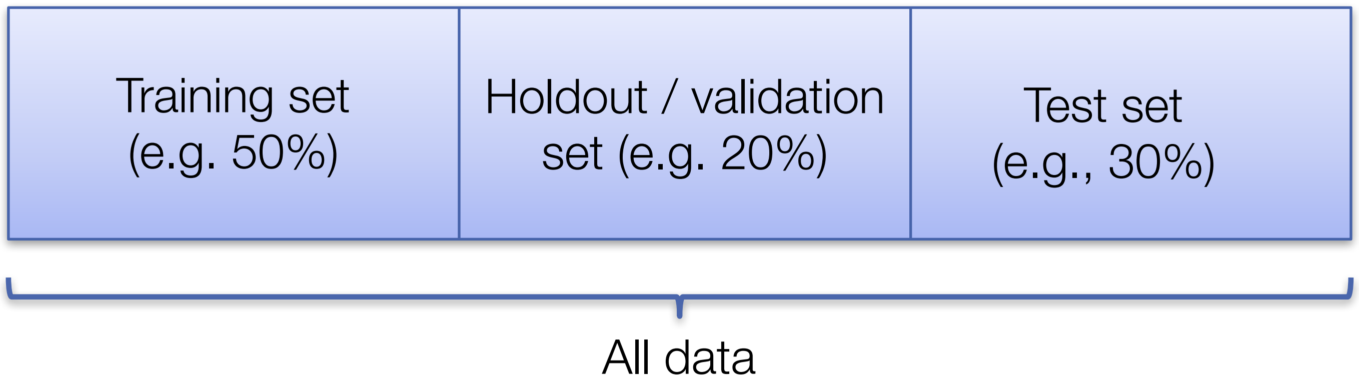
What is wrong with this?

Issues with the previous evaluation

Even though we used a training/holdout split to fit the *parameters*, we are still effectively fitting the *hyperparameters* to the holdout set

Imagine an algorithm that ignores the training set and makes random predictions; given a large enough hyperparameter search (e.g., over random seed), we could get perfect holdout performance

What to do instead



1. Divide data into training set, holdout set, and test set

2. Train algorithm on training set (i.e., to learn parameters), use holdout set to select hyperparameters

3. (Optional) retrain system on training + holdout

4. Evaluate performance on test set

In practice…

“Leakage” of test set performance into algorithm design decisions in almost always a reality when dealing with any fixed data set (in theory, as soon as you look at test set performance once, you have corrupted that data as a valid set set)

This is true in research as well as in data science practice

The best solutions: evaluate your system “in the wild” (where it will see truly novel examples) as often a possible; recollect data if you suspect overfitting to test set; look at test set performance sparingly

An interesting and very active area of research: adaptive data analysis (differential privacy to theoretically guarantee no overfitting)

Outline

Example: return to peak demand prediction

Overfitting, generalization, and cross validation

Regularization

General nonlinear features

Nonlinear classification

Evaluating machine learning algorithms

Other (classification) metrics

Classification metrics

So far, we have considered accuracy (0/1 loss) as the primary method for evaluating classifiers

However, sometimes the benefits for correctly classifying positive and negative examples are different, as are the costs for predictive a positive example to be negative, and vice versa

In cancer dataset, it is a very different thing (in terms of real-world effects) to predict that an actually malignant tumor is benign, versus predicting a benign tumor is malignant

Confusion matrix

A *confusion matrix* explicitly lists the number of examples for each actual class and each prediction

	Predicted Positive	Predicted Negative
Actual Positive	True Positive	False Negative
Actual Negative	False Positive	True negative

Can compute these (and all associated metrics) on training / holdout / testing sets, but we'll just show examples on training sets here

```
import sklearn.metrics
sklearn.metrics.confusion_matrix(y, clf.predict(X))
```

Derived quantities

Several common metrics are associated with entries of the confusion matrix (TP = true positive, FP = false positive, TN = true negative, TP = true negative)

$$\text{TP Rate (also called Recall)} = \frac{TP}{TP + FN}$$

$$\text{FP Rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Different metrics (and man others) are standard for different domains

Changing the prediction threshold

Classifiers are implicitly trained around a “threshold” of zero (positive hypothesis means predict positive, negative means predict negative)

But there is no reason to use only this threshold when we want to make predictions (may want to “overpredict” one class or the other)

Key idea: by *sorting* the hypothesis function outputs, and adjusting the threshold at which we call something positive or negative, we can sweep out *all* possible classifications that a classifier can produce

Example thresholds

Sorted hypothesis function outputs (assume 10 total examples, 5 total positive examples):

$$\text{sorted}(h_\theta(x^{(i)})) = \begin{bmatrix} 10 \\ 9 \\ 8.5 \\ \vdots \end{bmatrix}, y = \begin{bmatrix} +1 \\ -1 \\ +1 \\ \vdots \end{bmatrix}$$

TP Rate = 0.0, FP Rate = 0.0

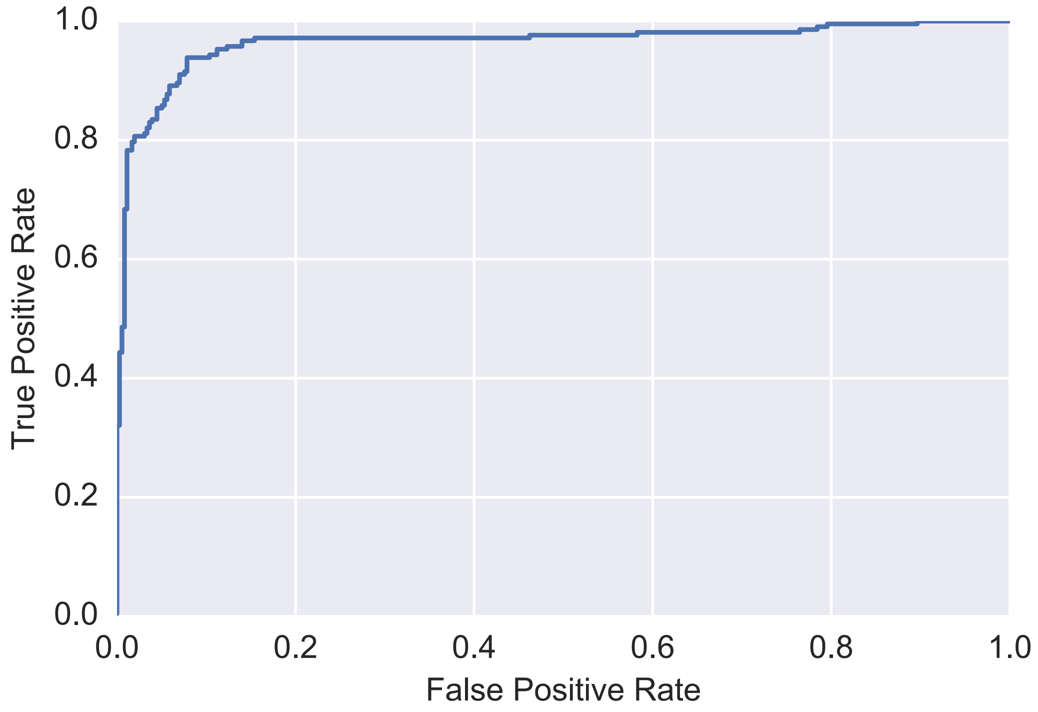
TP Rate = 0.2, FP Rate = 0.0

TP Rate = 0.2, FP Rate = 0.2

TP Rate = 0.4, FP Rate = 0.2

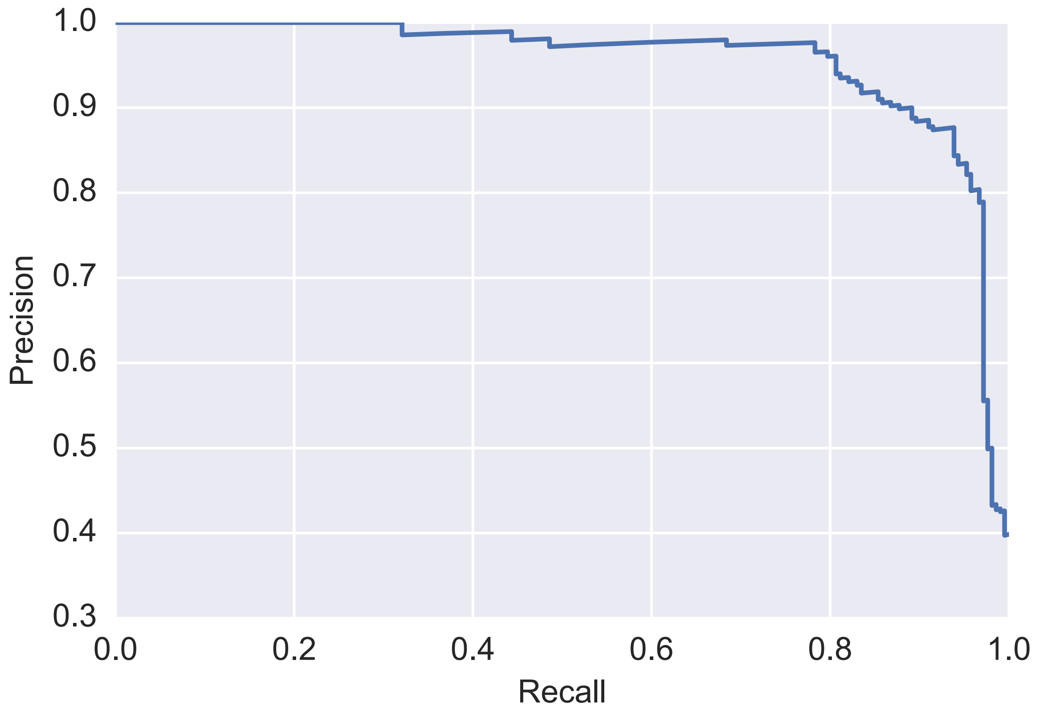
ROC Curve

If we plot the true positive rate versus the false positive rate for this procedure, we get a figure known as an ROC (receiver operating characteristic) curve



Precision recall curves

We can perform similar operations for other metrics, to for e.g. a precision-recall curve (plot of recall vs. precision as threshold varies)


Precision
Recall