

15-780 – Graduate Artificial Intelligence: Machine learning

J. Zico Kolter (this lecture) and Ariel Procaccia
Carnegie Mellon University
Spring 2018

Outline

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

Outline

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

Introduction: digit classification

The task: write a program that, given a 28x28 grayscale image of a digit, outputs the number in the image

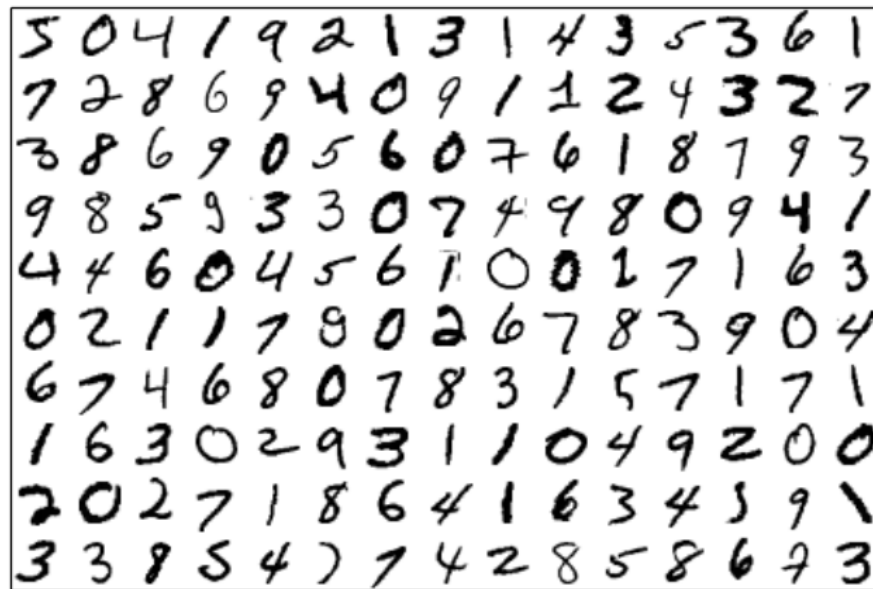




Image: digits from the MNIST data set
(<http://yann.lecun.com/exdb/mnist/>)

Approaches

Approach 1: try to write a program by hand that uses your a priori knowledge about what images look like to determine what number they are

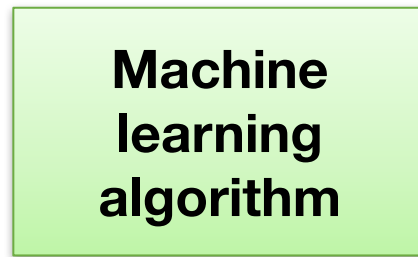
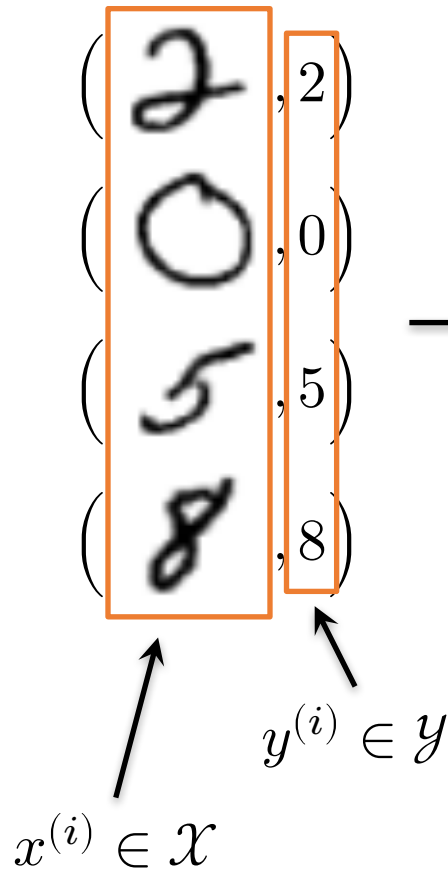
Approach 2: (the machine learning approach) collect a large volume of images and their corresponding numbers, let the “write its own program” to map from these images to their corresponding number

(More precisely, this is a subset of machine learning called supervised learning)

 → 8 → 5

Supervised learning pipeline

Training data



Hypothesis function

$$h: \mathcal{X} \rightarrow \mathcal{Y} \text{ such that } y^{(i)} \approx h(x^{(i)}), \forall i$$

(On new data $x' \in \mathcal{X}$,
make prediction
 $y' = h(x')$)

Outline

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

A simple example: predicting electricity use

What will peak power consumption be in Pittsburgh tomorrow?

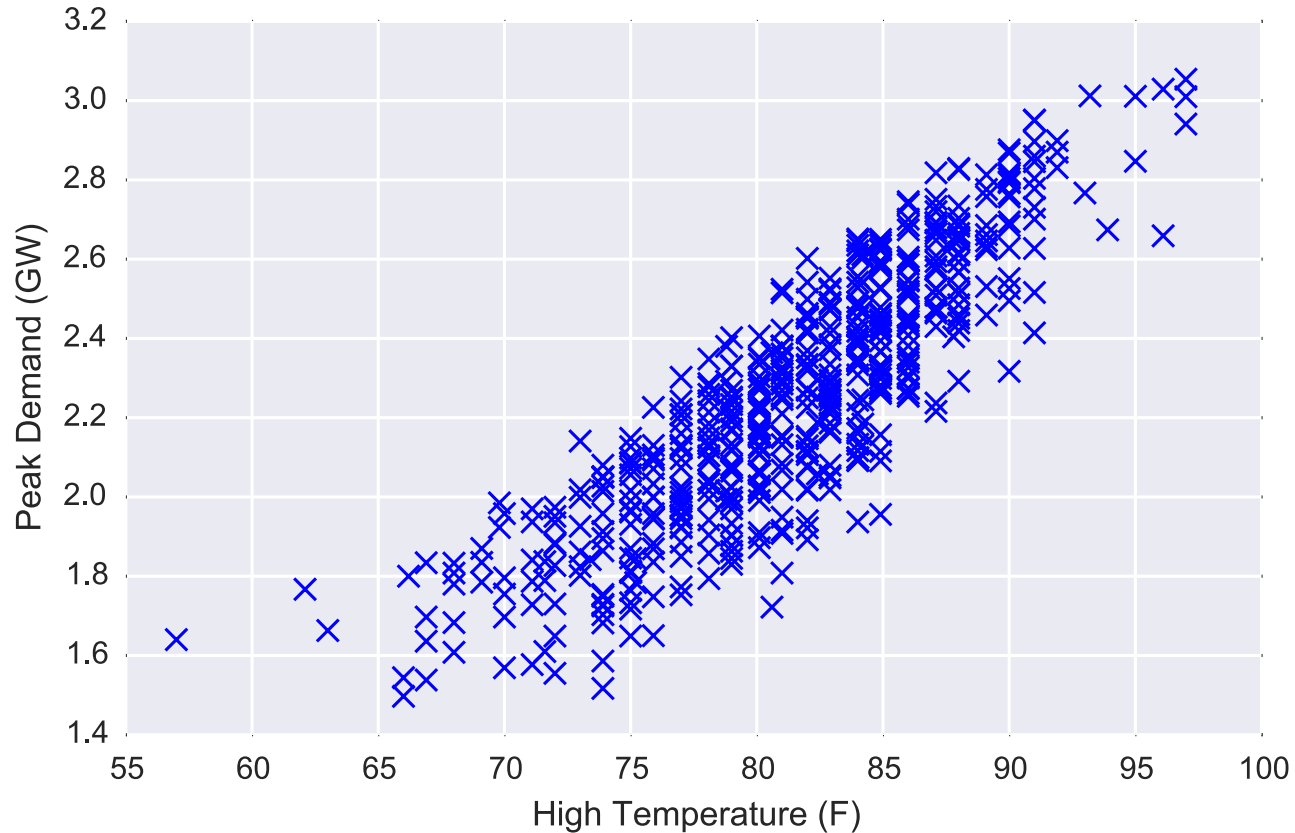
Difficult to build an “a priori” model from first principles to answer this question

But, relatively easy to record past days of consumption, plus additional features that affect consumption (i.e., weather)

Date	High Temperature (F)	Peak Demand (GW)
2011-06-01	84.0	2.651
2011-06-02	73.0	2.081
2011-06-03	75.2	1.844
2011-06-04	84.9	1.959
...

Plot of consumption vs. temperature

Plot of high temperature vs. peak demand for summer months (June – August) for past six years



Hypothesis: linear model

Let's suppose that the peak demand approximately fits a *linear model*

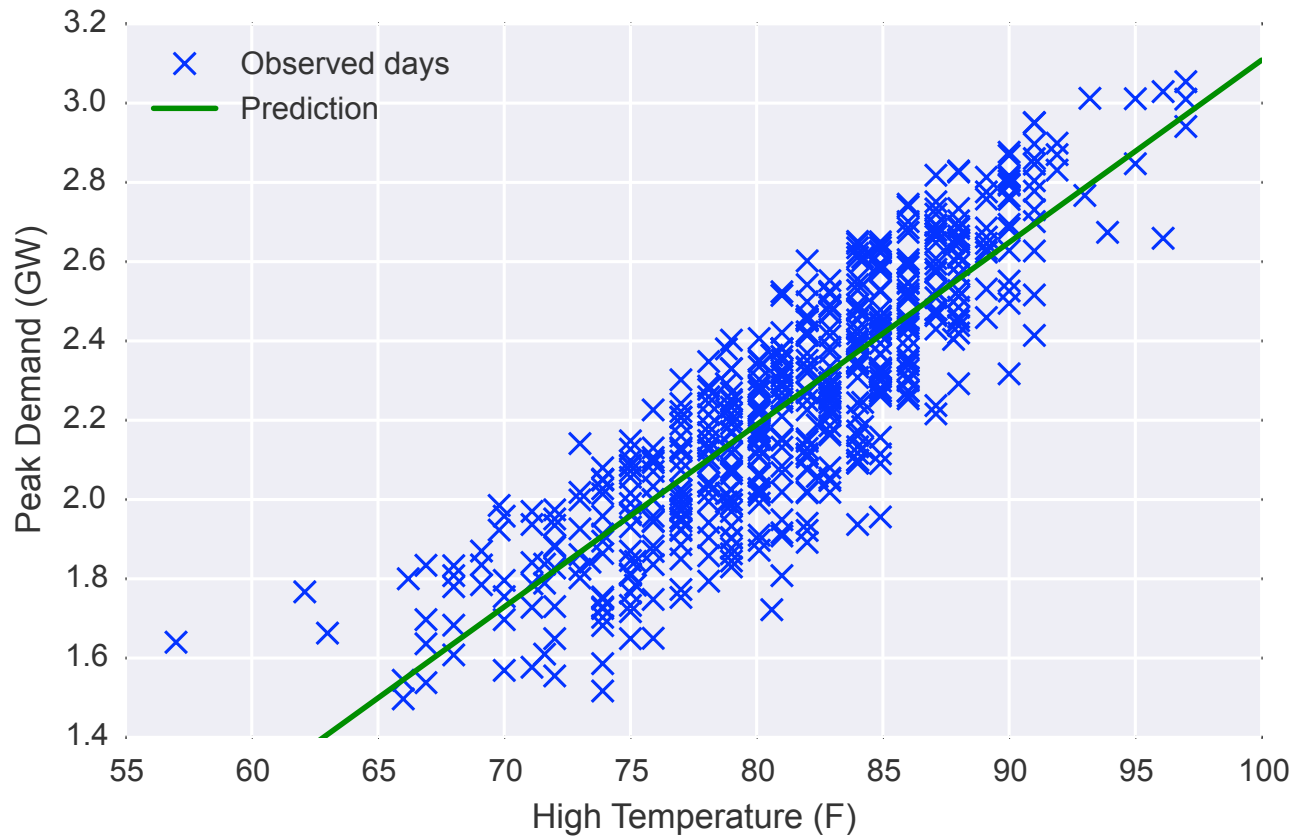
$$\text{Peak_Demand} \approx \theta_1 \cdot \text{High_Temperature} + \theta_2$$

Here θ_1 is the “slope” of the line, and θ_2 is the intercept

Now, given a forecast of tomorrow's weather (ignoring for a moment that this is also a prediction), we can predict how high the peak demand

Predictions

Predicting in this manner is equivalent to “drawing line through data”



Machine learning notation

Input features: $x^{(i)} \in \mathbb{R}^n, i = 1, \dots, m$

$$\text{E. g.: } x^{(i)} = \begin{bmatrix} \text{High_Temperature}^{(i)} \\ 1 \end{bmatrix}$$

Outputs: $y^{(i)} \in \mathbb{R}$ (regression task)

$$\text{E. g.: } y^{(i)} \in \mathbb{R} = \text{Peak_Demand}^{(i)}$$

**Training
data**

Model parameters: $\theta \in \mathbb{R}^n$

Hypothesis function: $h_{\theta}: \mathbb{R}^n \rightarrow \mathbb{R}$, predicts output given input

$$\text{E. g.: } h_{\theta}(x) = \theta^T x = \sum_{j=1}^n \theta_j \cdot x_j$$

Loss functions

How do we measure how “good” a hypothesis function is, i.e. how close is our approximation on our training data

$$y^{(i)} \approx h_{\theta}(x^{(i)})$$

Typically done by introducing a loss function

$$\ell: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}_+$$

where $\ell(h_{\theta}(x), y)$ denotes how far apart prediction is from actual output

E.g., for regression a common loss function is squared error:

$$\ell(h_{\theta}(x), y) = (h_{\theta}(x) - y)^2$$

The canonical machine learning problem

With this notation, we define the canonical machine learning problem: given a set of input features and outputs $(x^{(i)}, y^{(i)})$, $i = 1, \dots, m$, find the parameters that minimize the sum of losses

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

Virtually all machine learning algorithms have this form, we just need to specify

1. What is the hypothesis function?
2. What is the loss function?
3. How do we solve the optimization problem?

Least squares

Let's formulate our linear least squares problem in this notation

Hypothesis function: $h_{\theta}(x) = \theta^T x$

Squared loss function: $\ell(h_{\theta}(x), y) = (h_{\theta}(x) - y)^2$

Leads to the machine learning optimization problem

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^m \ell(h_{\theta}(x^{(i)}), y^{(i)}) \equiv \underset{\theta}{\text{minimize}} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2$$

A convex optimization problem in θ , so we expect global solutions

But how do we solve this optimization problem?

Solution via gradient descent

Recall the gradient descent algorithm (written now to optimize θ)

$$\text{Repeat: } \theta \rightarrow \theta - \alpha \nabla_{\theta} f(\theta)$$

What is the gradient of our objective function?

$$\begin{aligned} \nabla_{\theta} \sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2 &= \sum_{i=1}^m \nabla_{\theta} (\theta^T x^{(i)} - y^{(i)})^2 \\ &= 2 \sum_{i=1}^m x^{(i)} (\theta^T x^{(i)} - y^{(i)}) \end{aligned}$$

(using chain rule and the fact that $\nabla_{\theta} \theta^T x^{(i)} = x^{(i)}$), gives update:

$$\text{Repeat: } \theta \rightarrow \theta - \alpha \sum_{i=1}^m x^{(i)} (\theta^T x^{(i)} - y^{(i)})$$

Linear algebra notation

Summation notation gets cumbersome, so convenient to introduce a more compact notation:

$$X = \begin{bmatrix} (x^{(1)})^T \\ (x^{(2)})^T \\ \vdots \\ (x^{(m)})^T \end{bmatrix} \in \mathbb{R}^{m \times n}, \quad y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^m$$

Least squares objective can now be written

$$\sum_{i=1}^m (\theta^T x^{(i)} - y^{(i)})^2 = \|X\theta - y\|_2^2$$

and gradient given by $\nabla_{\theta} \|X\theta - y\|_2^2 = 2X^T (X\theta - y)$

An alternative solution method

In order for θ^* to minimize some (unconstrained, differentiable), function f , necessary and sufficient that $\nabla_{\theta} f(\theta^*) = 0$

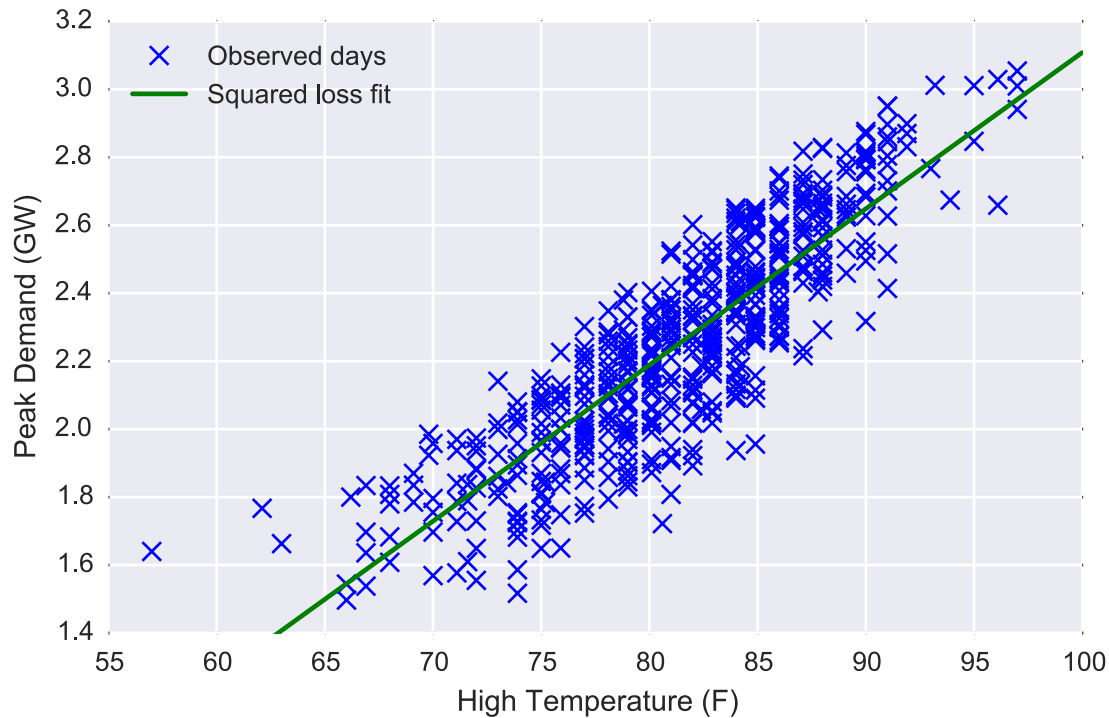
Previously, we attained this point iteratively through gradient descent, but for squared error loss, we can also find it analytically

$$\begin{aligned}\nabla_{\theta} \|X\theta^* - y\|_2^2 &= 0 \\ \implies 2X^T(X\theta^* - y) &= 0 \\ \implies X^T X\theta^* &= X^T y \\ \implies \theta^* &= (X^T X)^{-1} X^T y\end{aligned}$$

These are called the *normal equations*, a closed form solution for minimization of sum of squared losses

Least squares solution

Solving normal equations (or running gradient descent), gives coefficients θ_1 and θ_2 corresponding to the following fit



Poll: least squares when $m < n$

What happens you run a least-squares solver, built using the simple normal equations in Python, when $m < n$?

1. Python will return an error, because the true minimum least-squares cost is infinite
2. Python will return an error, even though the true minimum least-squares cost is zero
3. Python will correctly compute the optimal solution, with strictly positive cost
4. Python will correctly compute the optimal solution, with zero cost

Alternative loss functions

Why did we pick the squared loss $\ell(h_\theta(x), y) = (h_\theta(x) - y)^2$?

Why not use an alternative like absolute loss $\ell(h_\theta(x), y) = |h_\theta(x) - y|$?

We could write this optimization problem as

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^m \ell(h_\theta(x^{(i)}), y^{(i)}) \equiv \underset{\theta}{\text{minimize}} \|X\theta - y\|_1$$

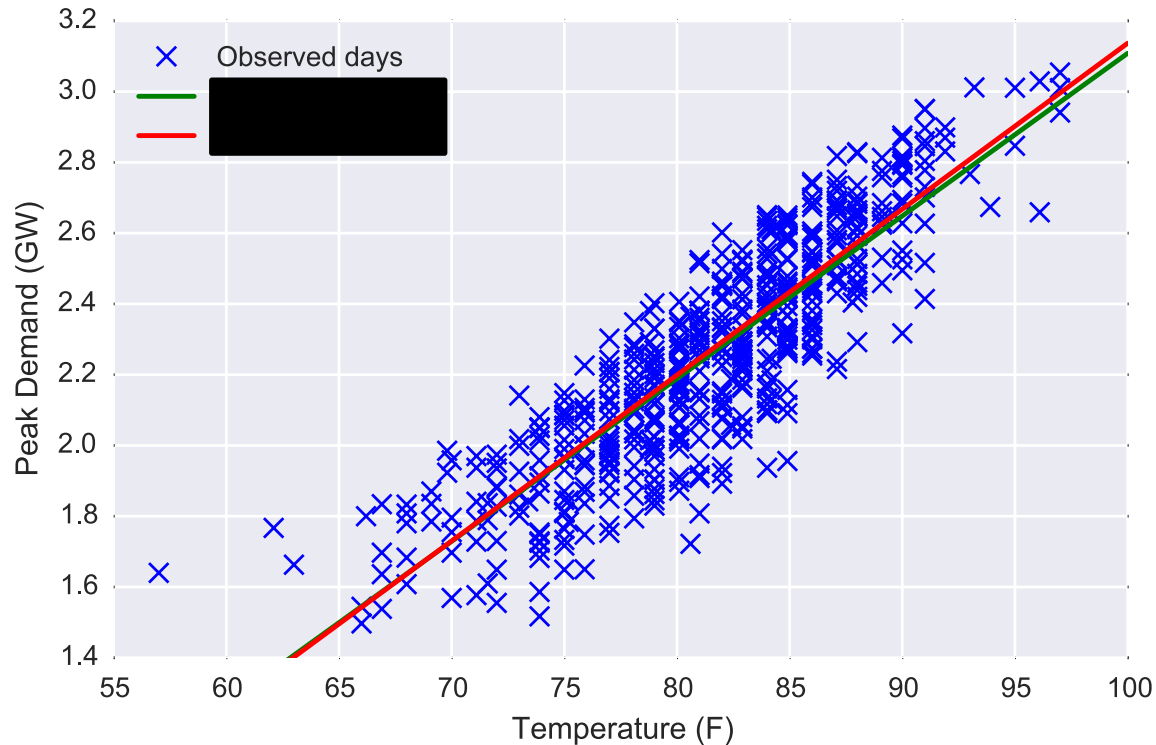
where $\|z\|_1 = \sum_i |z_i|$ is called the ℓ_1 norm

No closed-form solution, but (sub)gradient is given by

$$\nabla_\theta \|X\theta - y\|_1 = X^T \text{sign}(X\theta - y)$$

Poll: alternative loss solutions

Solutions for minimizing squared error and absolute error



Poll: which solution is which?

1. Green is squared loss, red is absolute
2. Red is squared loss, green is absolute
3. Those lines look identical to me

Outline

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

Classification tasks

Regression tasks: predicting real-valued quantity $y \in \mathbb{R}$

Classification tasks: predicting *discrete-valued* quantity y

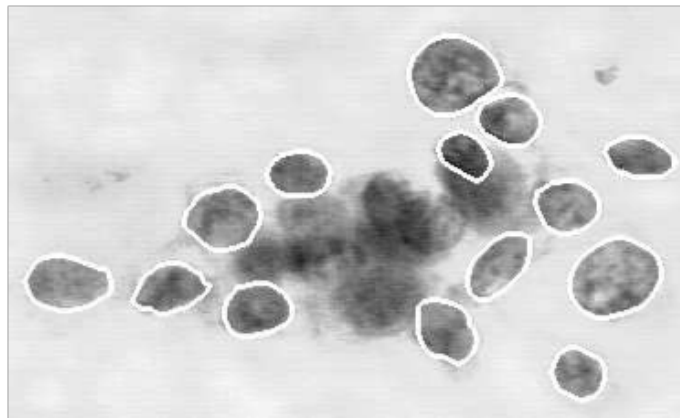
Binary classification: $y \in \{-1, +1\}$

Multiclass classification: $y \in \{1, 2, \dots, k\}$

Example: breast cancer classification

Well-known classification example: using machine learning to diagnose whether a breast tumor is benign or malignant [Street et al., 1992]

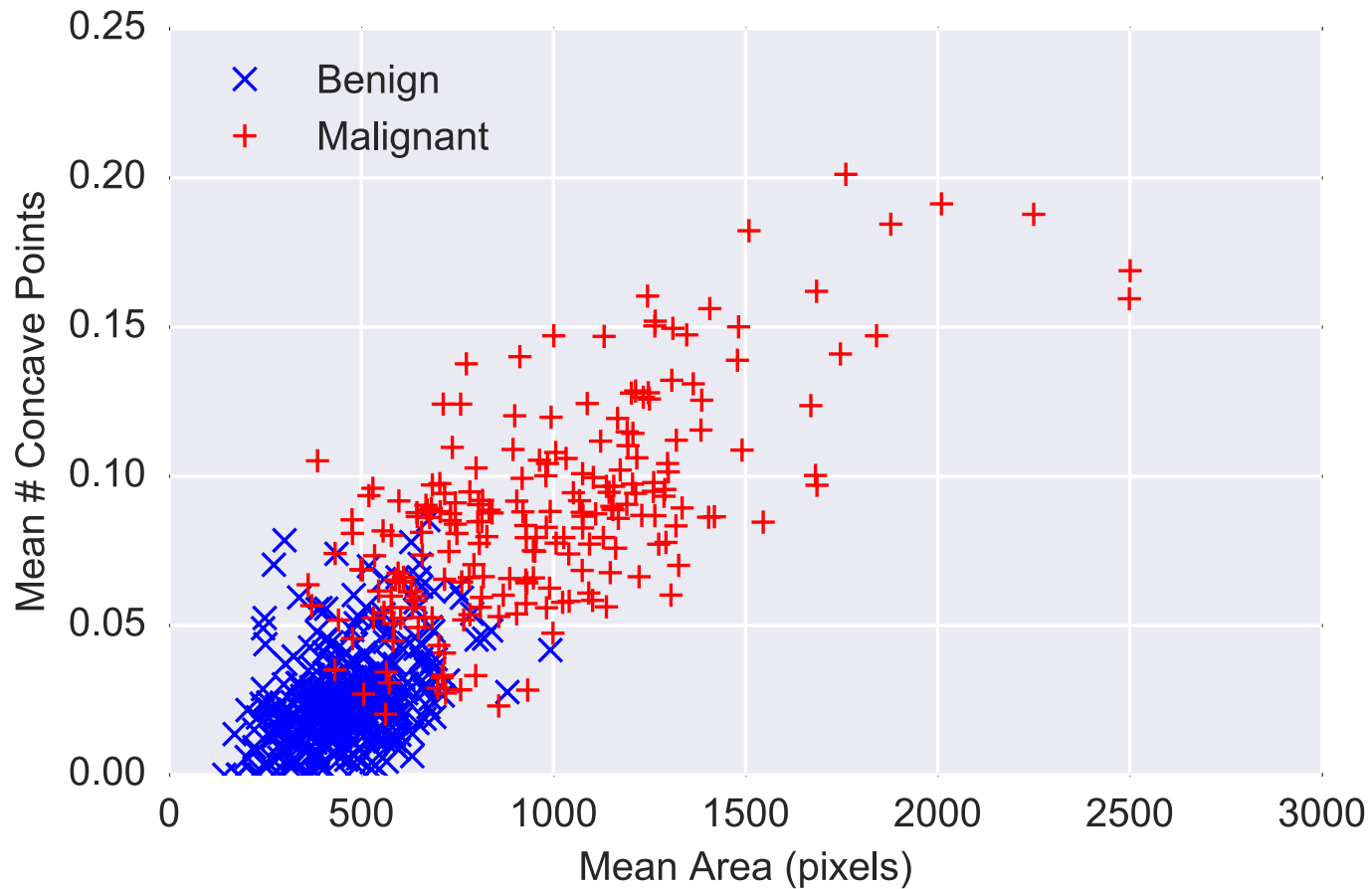
Setting: doctor extracts a sample of fluid from tumor, stains cells, then outlines several of the cells (image processing refines outline)



System computes features for each cell such as area, perimeter, concavity, texture (10 total); computes mean/std/max for all features

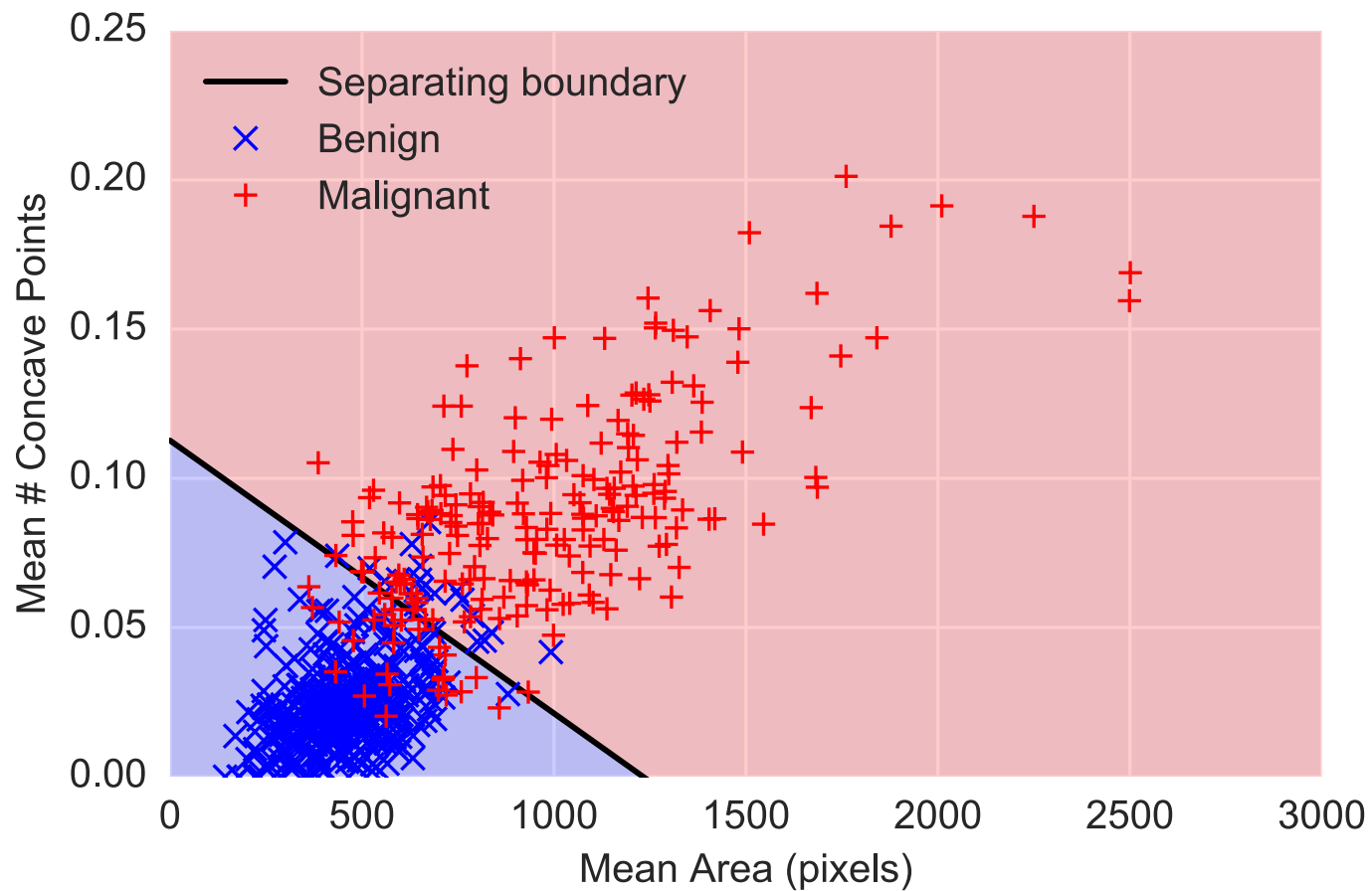
Example: breast cancer classification

Plot of two features: mean area vs. mean concave points, for two classes



Linear classification example

Linear classification \equiv “drawing line separating classes”



Formal setting

Input features: $x^{(i)} \in \mathbb{R}^n, i = 1, \dots, m$

$$\text{E. g.: } x^{(i)} = \begin{bmatrix} \text{Mean_Area}^{(i)} \\ \text{Mean_Concave_Points}^{(i)} \\ 1 \end{bmatrix}$$

Outputs: $y^{(i)} \in \{-1, +1\}, i = 1, \dots, m$

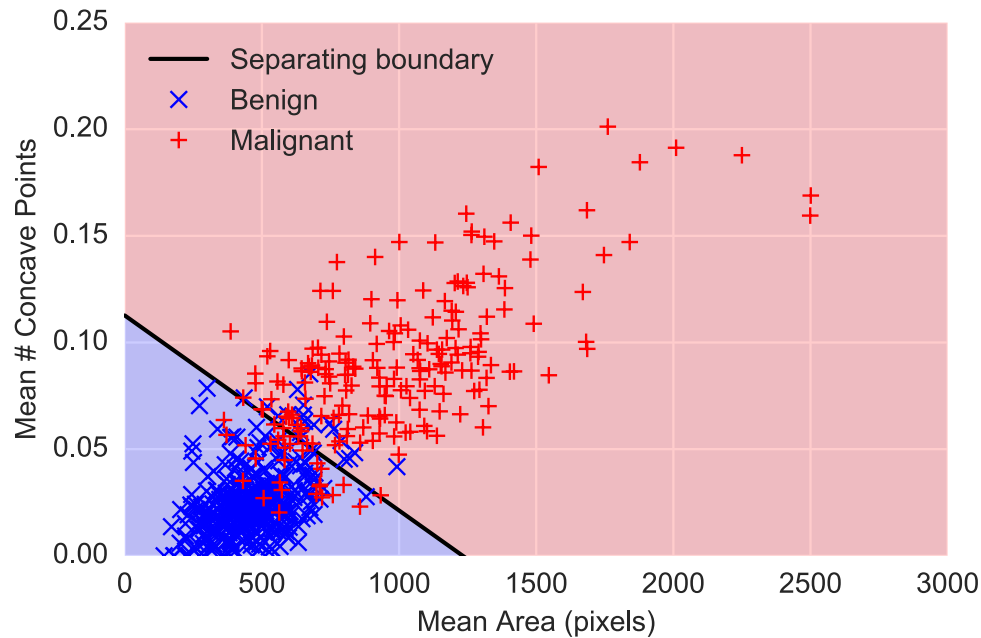
$$\text{E. g.: } y^{(i)} \in \{-1 \text{ (benign)}, +1 \text{ (malignant)}\}$$

Model parameters: $\theta \in \mathbb{R}^n$

Hypothesis function: $h_\theta: \mathbb{R}^n \rightarrow \mathbb{R}$, aims for same *sign* as the output (informally, a measure of *confidence* in our prediction)

$$\text{E. g.: } h_\theta(x) = \theta^T x, \quad \hat{y} = \text{sign}(h_\theta(x))$$

Understanding linear classification diagrams



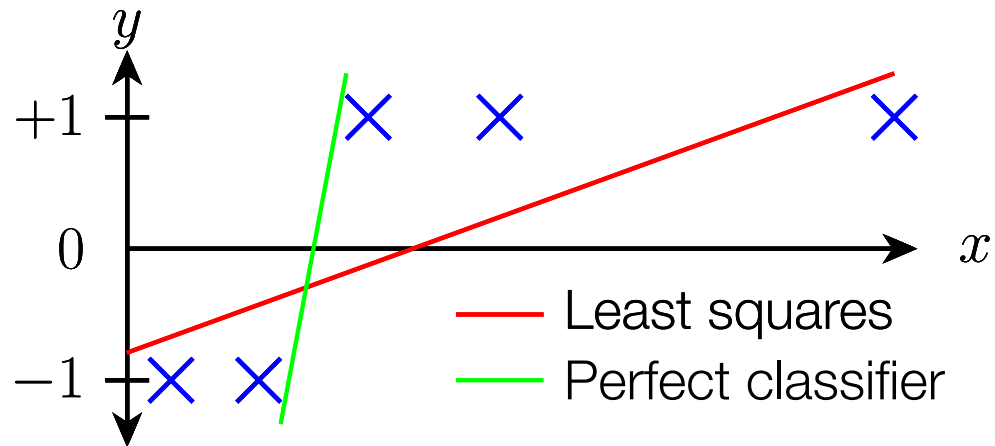
Color shows regions where the $h_{\theta}(x)$ is positive

Separating boundary is given by the equation $h_{\theta}(x) = 0$

Loss functions for classification

How do we define a loss function $\ell: \mathbb{R} \times \{-1, +1\} \rightarrow \mathbb{R}_+$?

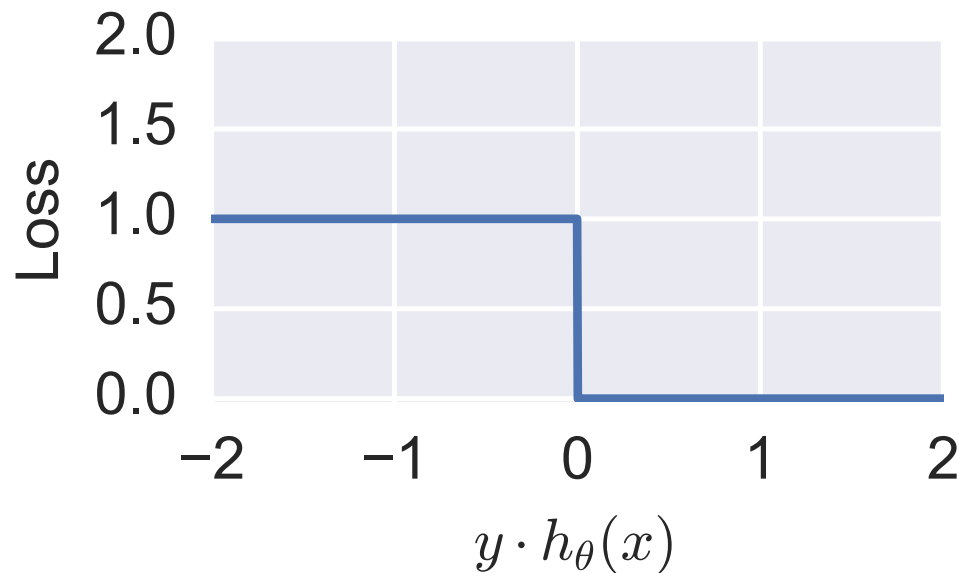
What about just using squared loss?



0/1 loss (i.e. error)

The loss we would like to minimize (0/1 loss, or just “error”):

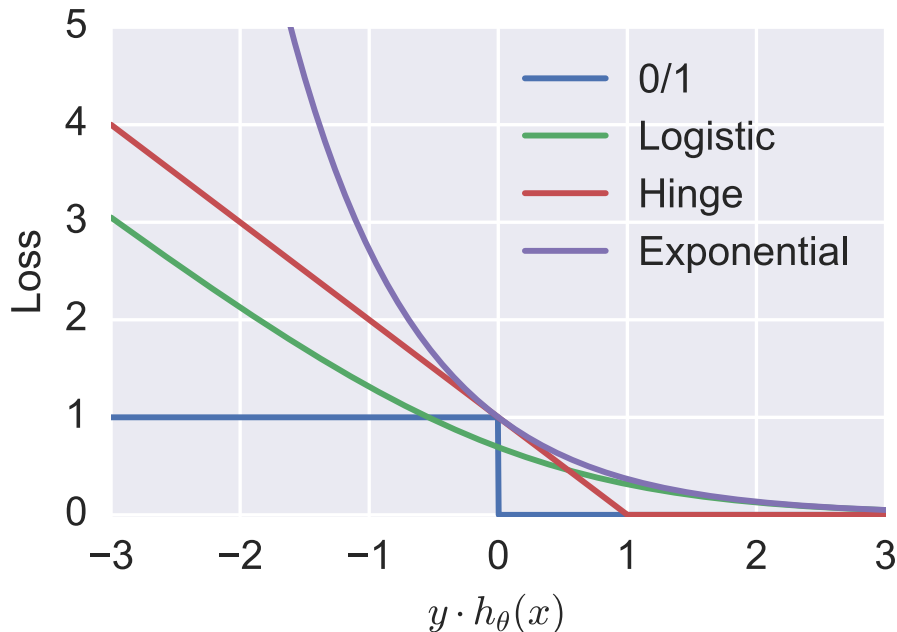
$$\begin{aligned} \ell_{0/1}(h_{\theta}(x), y) &= \begin{cases} 0 & \text{if } \text{sign}(h_{\theta}(x)) = y \\ 1 & \text{otherwise} \end{cases} \\ &= \mathbf{1}\{y \cdot h_{\theta}(x) \leq 0\} \end{aligned}$$



Alternative losses

Unfortunately 0/1 loss is hard to optimize (NP-hard to find classifier with minimum 0/1 loss, relates to a property called convexity of the function)

A number of alternative losses for classification are typically used instead



$$\ell_{0/1} = 1\{y \cdot h_\theta(x) \leq 0\}$$

$$\ell_{\text{logistic}} = \log(1 + \exp(-y \cdot h_\theta(x)))$$

$$\ell_{\text{hinge}} = \max\{1 - y \cdot h_\theta(x), 0\}$$

$$\ell_{\text{exp}} = \exp(-y \cdot h_\theta(x))$$

Machine learning optimization

With this notation, the “canonical” machine learning problem is written in the exact same way

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

Again unlike least squares, typically no closed-form solution, so we rely on gradient descent

$$\text{Repeat: } \theta := \theta - \alpha \sum_{i=1}^m \nabla_{\theta} \ell(h_{\theta}(x^{(i)}), y^{(i)})$$

Support vector machine

A (linear) support vector machine (SVM) just solves the canonical machine learning optimization problem using hinge loss and linear hypothesis

$$\underset{\theta}{\text{minimize}} \sum_{i=1}^m \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\}$$

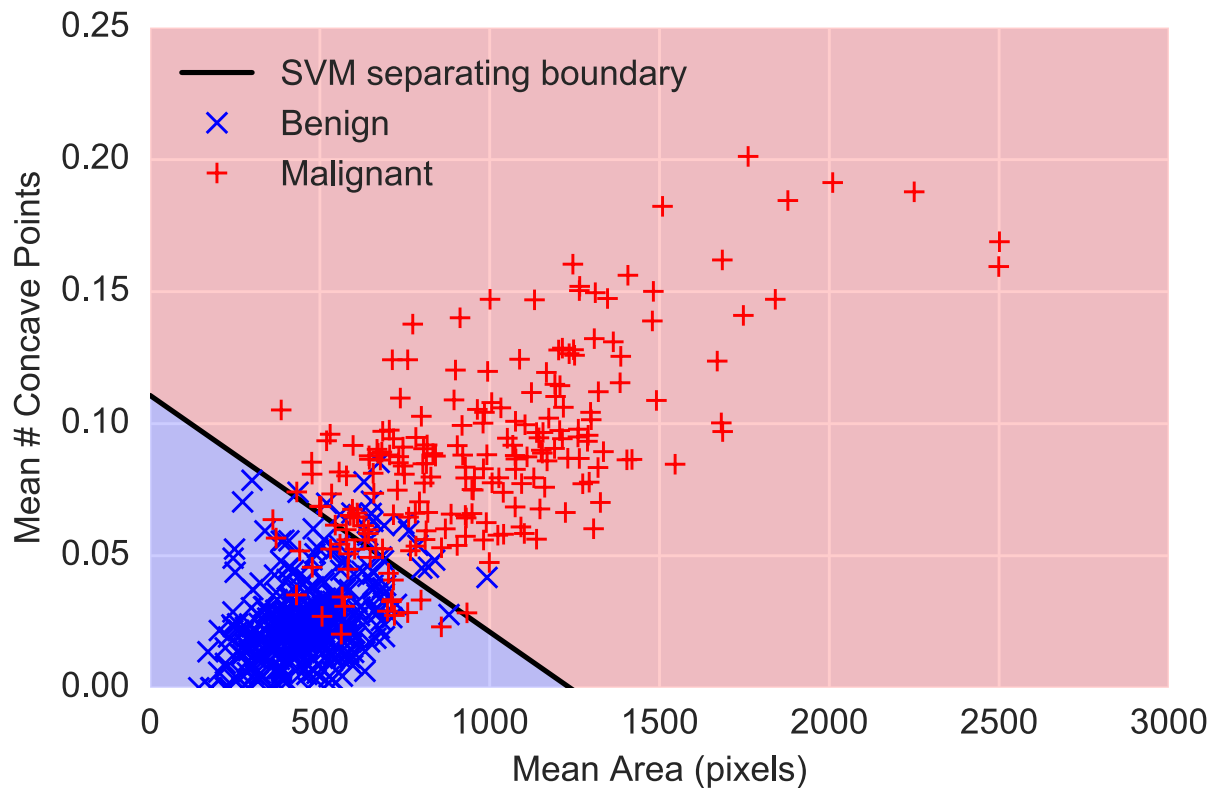
The standard SVM actually includes another term called a regularization term, but we'll talk about this next lecture

Updates using gradient descent:

$$\theta := \theta - \alpha \sum_{i=1}^m -y^{(i)} x^{(i)} 1\{y^{(i)} \cdot \theta^T x^{(i)} \leq 1\}$$

Support vector machine example

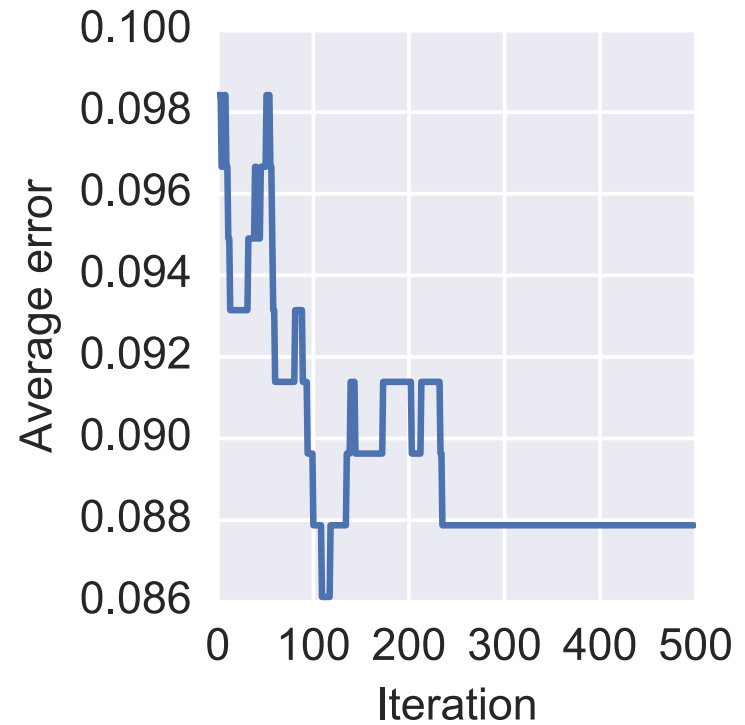
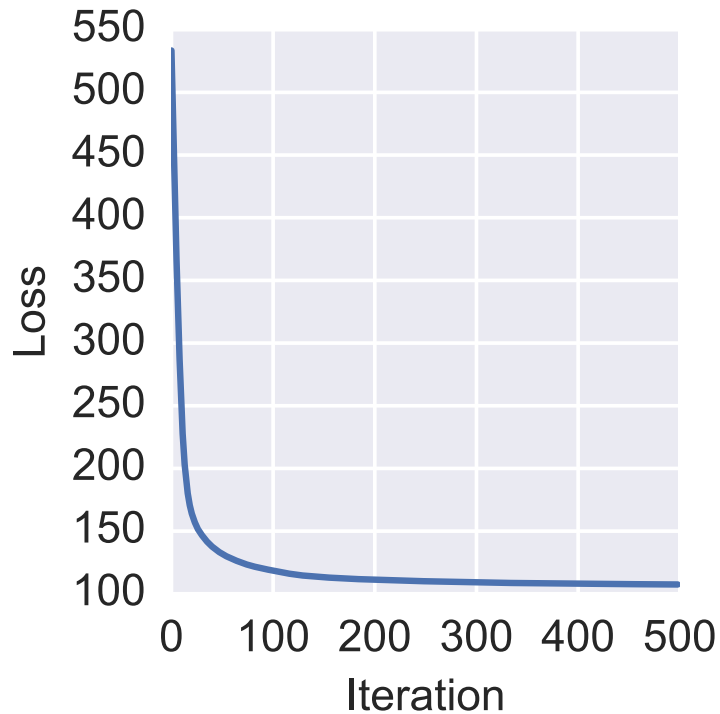
Running support vector machine on cancer dataset



$$\theta = \begin{bmatrix} 1.456 \\ 1.848 \\ -0.189 \end{bmatrix}$$

SVM optimization progress

Optimization objective and error versus gradient descent iteration number



Logistic regression

Logistic regression just solves this problem using logistic loss and linear hypothesis function

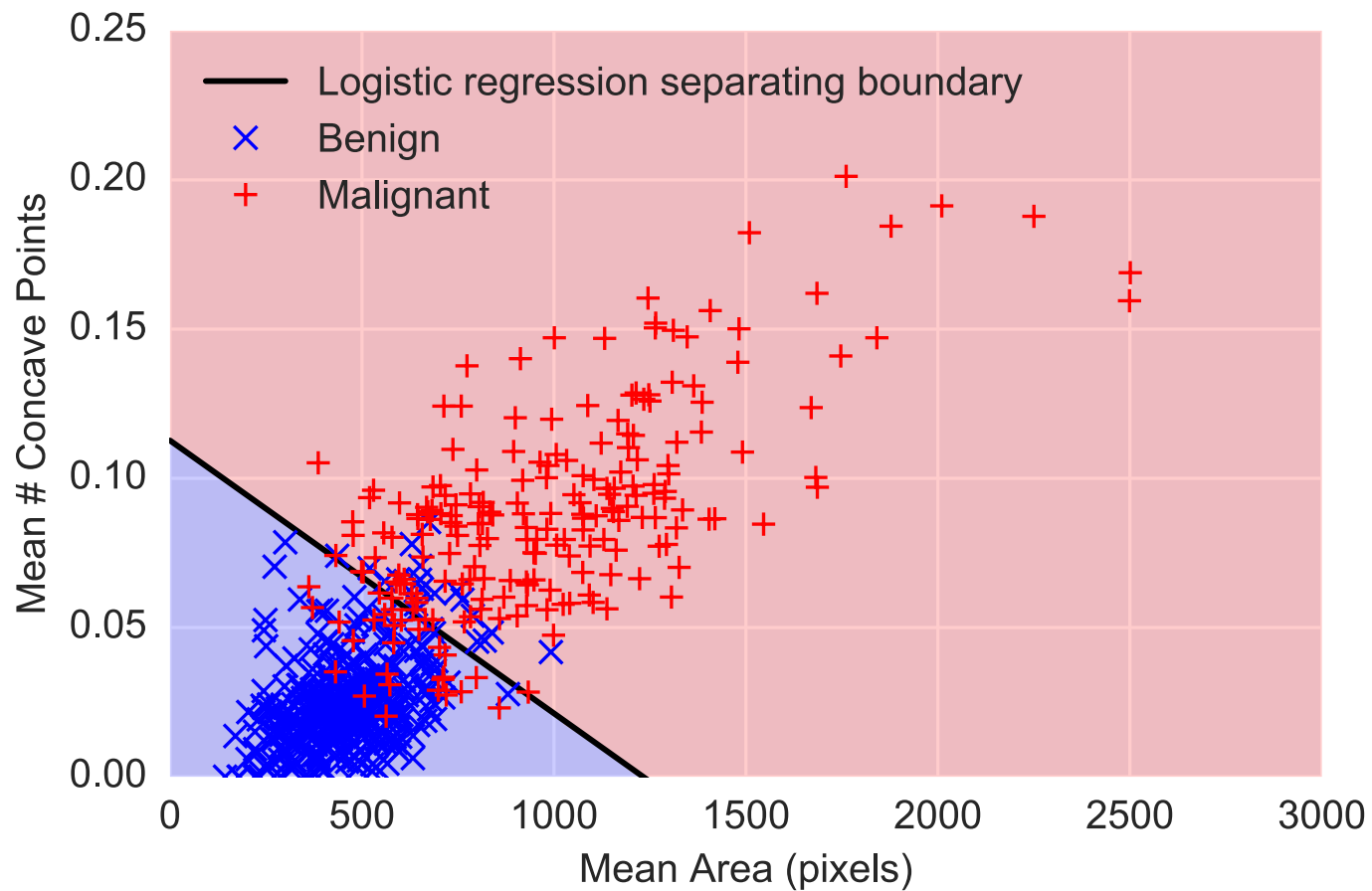
$$\underset{\theta}{\text{minimize}} \sum_{i=1}^m \log(1 + \exp(-y^{(i)} \cdot \theta^T x^{(i)}))$$

Gradient descent updates (can you derive these?):

$$\theta := \theta - \alpha \sum_{i=1}^m -y^{(i)} x^{(i)} \frac{1}{1 + \exp(y^{(i)} \cdot \theta^T x^{(i)})}$$

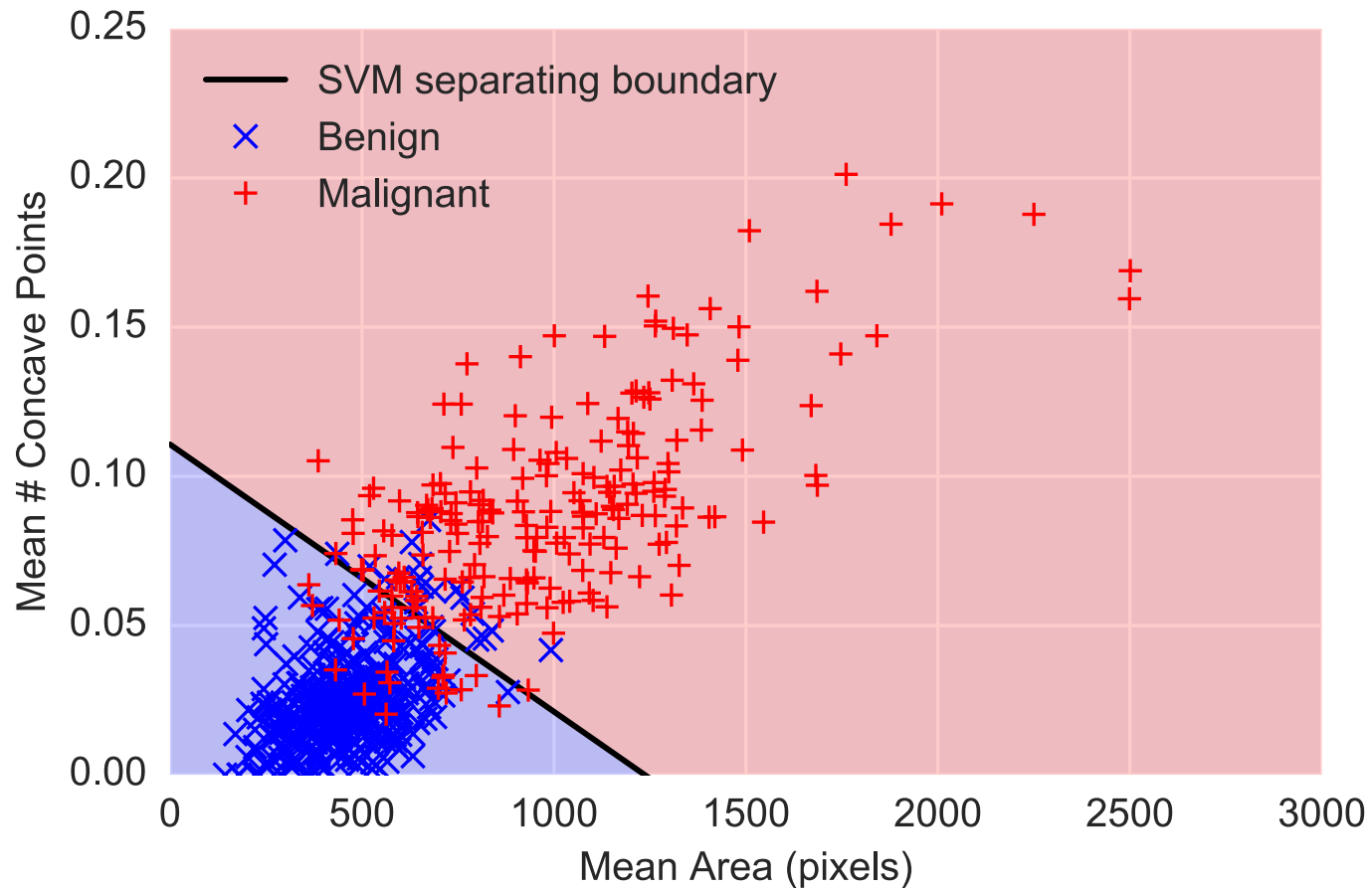
Logistic regression example

Running logistic regression on cancer data set



Logistic regression example

Running logistic regression on cancer data set



Multiclass classification

When output is in $\{1, \dots, k\}$ (e.g., digit classification), we can adopt a few different approaches

Approach 1: Build k different binary classifiers h_{θ_i} with the goal of predicting class i vs all others, output predictions as

$$\hat{y} = \operatorname{argmax}_i h_{\theta_i}(x)$$

Approach 2: Use a hypothesis function $h_{\theta}: \mathbb{R}^n \rightarrow \mathbb{R}^k$, define an alternative loss function $\ell: \mathbb{R}^k \times \{1, \dots, k\} \rightarrow \mathbb{R}_+$

E.g., softmax loss (also called cross entropy loss):

$$\ell(h_{\theta}(x), y) = \log \sum_{j=1}^k \exp(h_{\theta}(x)_j) - h_{\theta}(x)_y$$

Outline

What is machine learning?

Linear regression

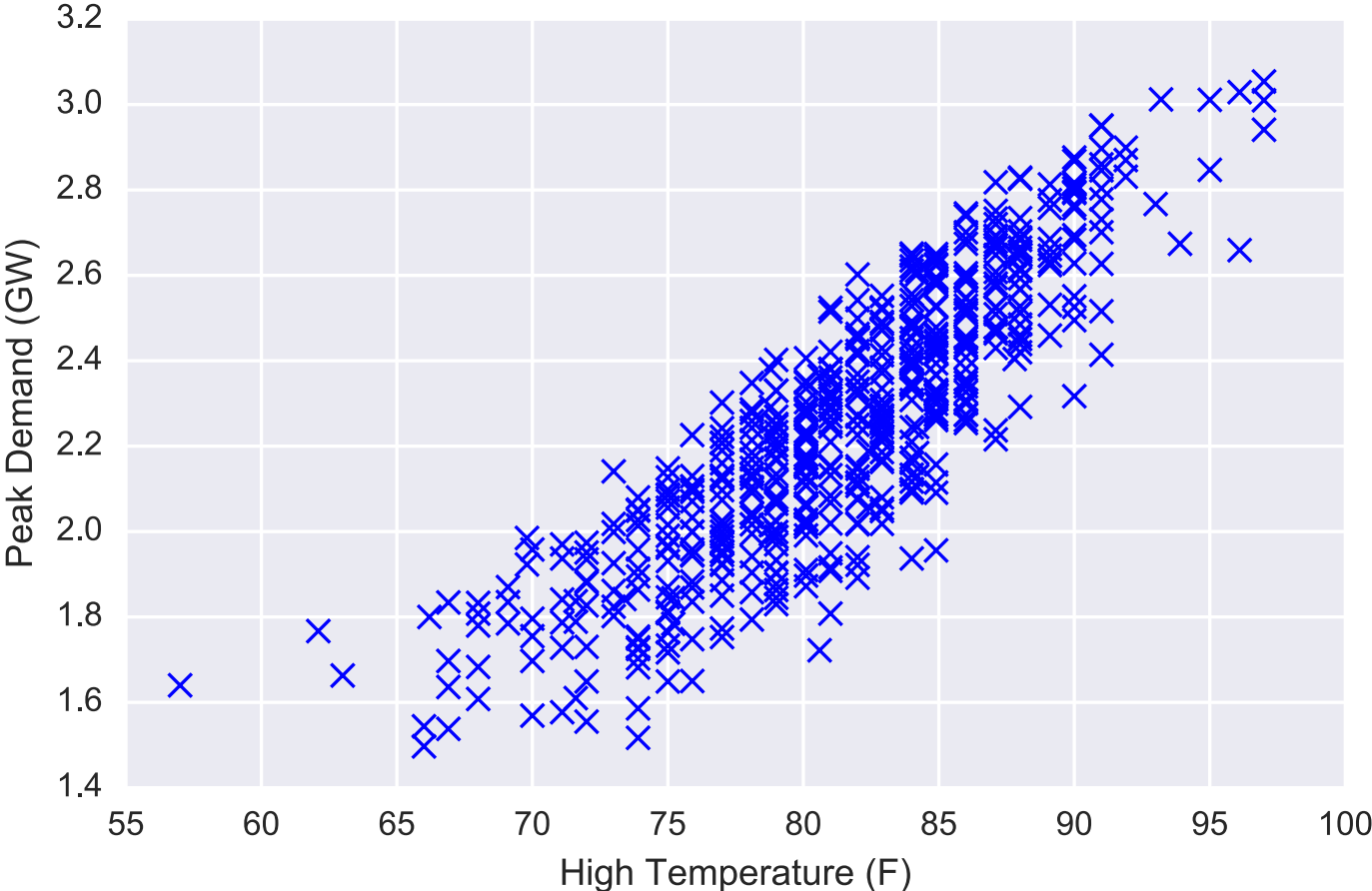
Linear classification

Nonlinear methods

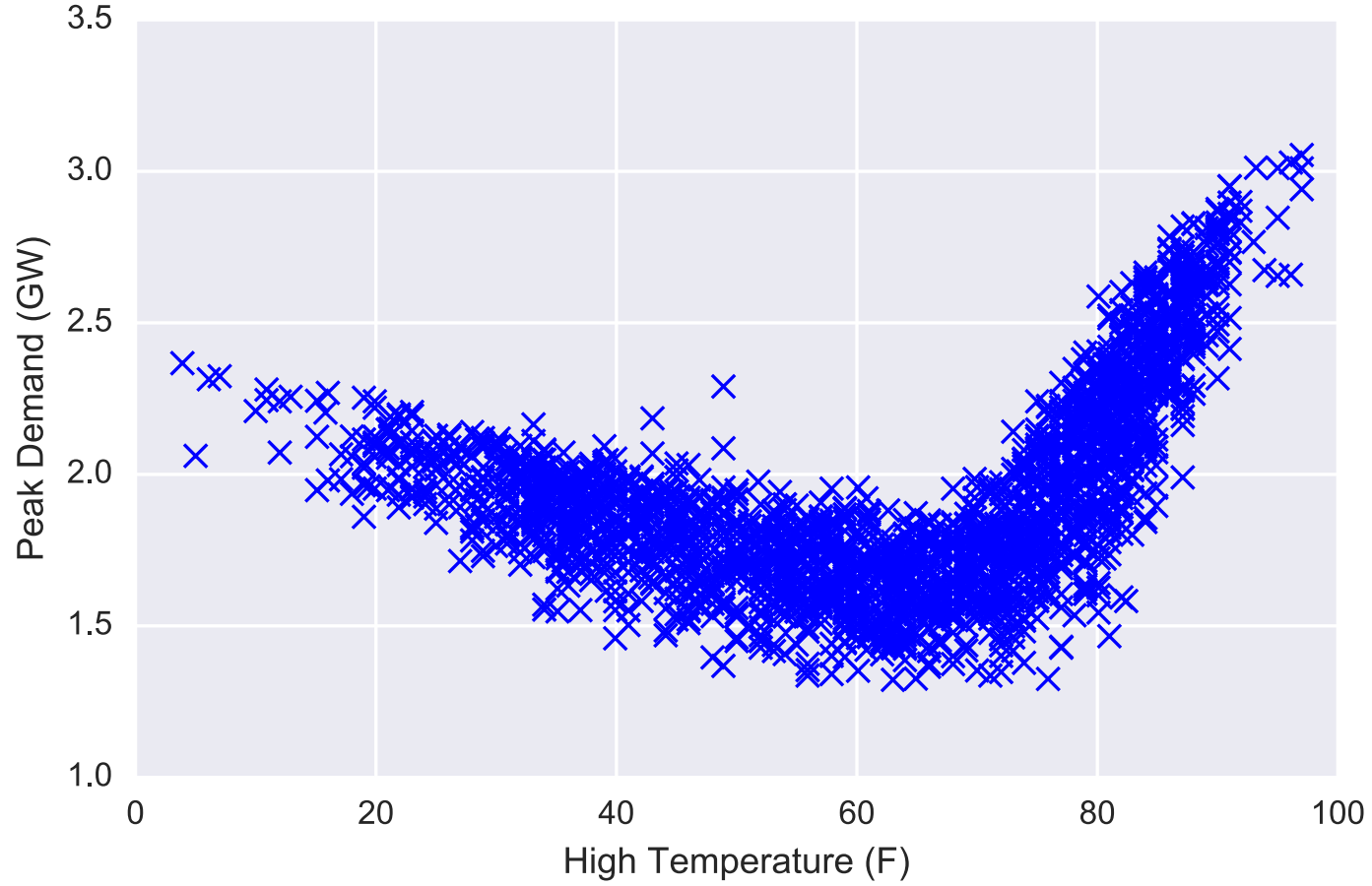
Overfitting, generalization, and regularization

Evaluating machine learning algorithms

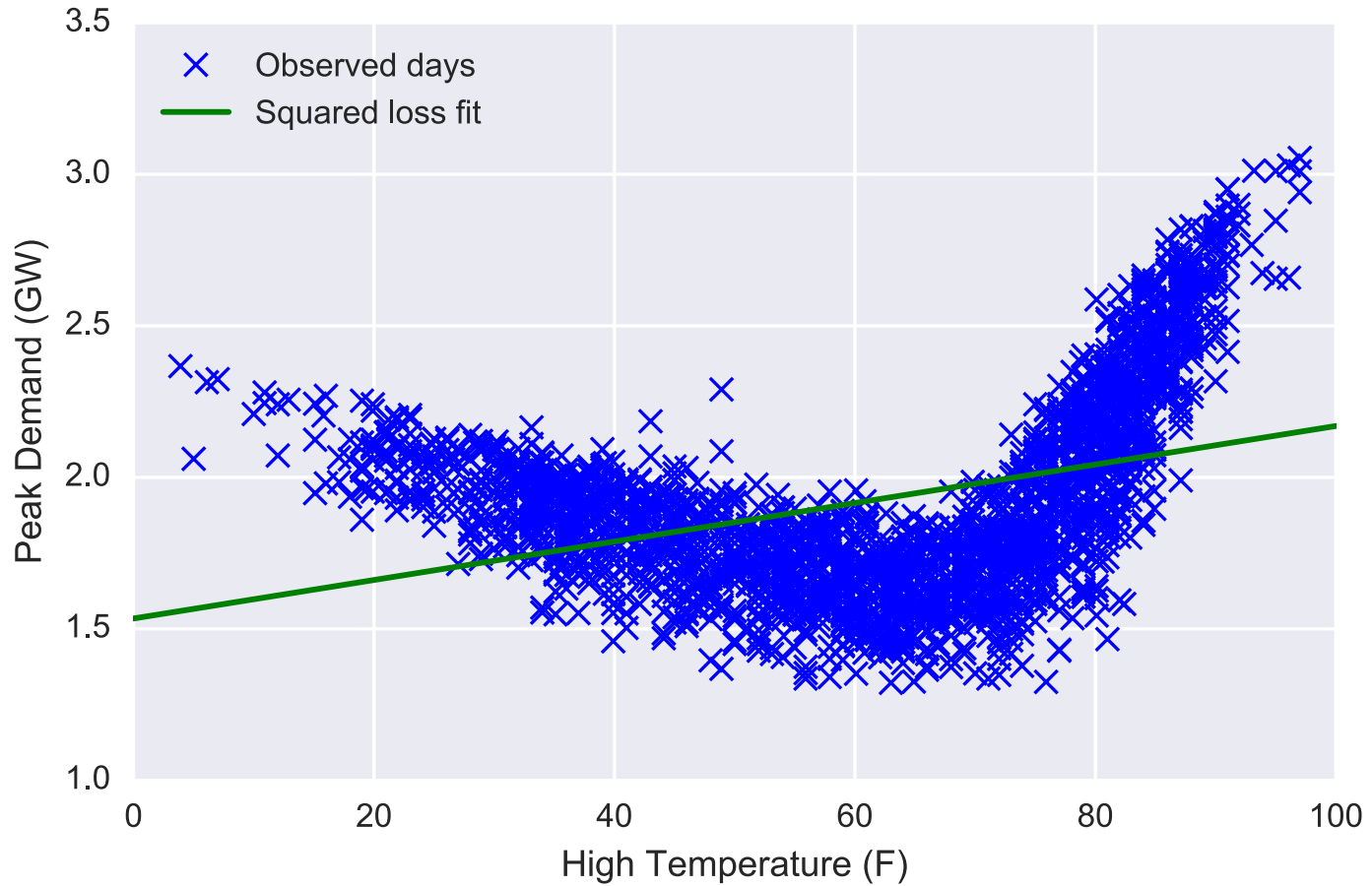
Peak demand vs. temperature (summer months)



Peak demand vs. temperature (all months)



Linear regression fit



“Non-linear” regression

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

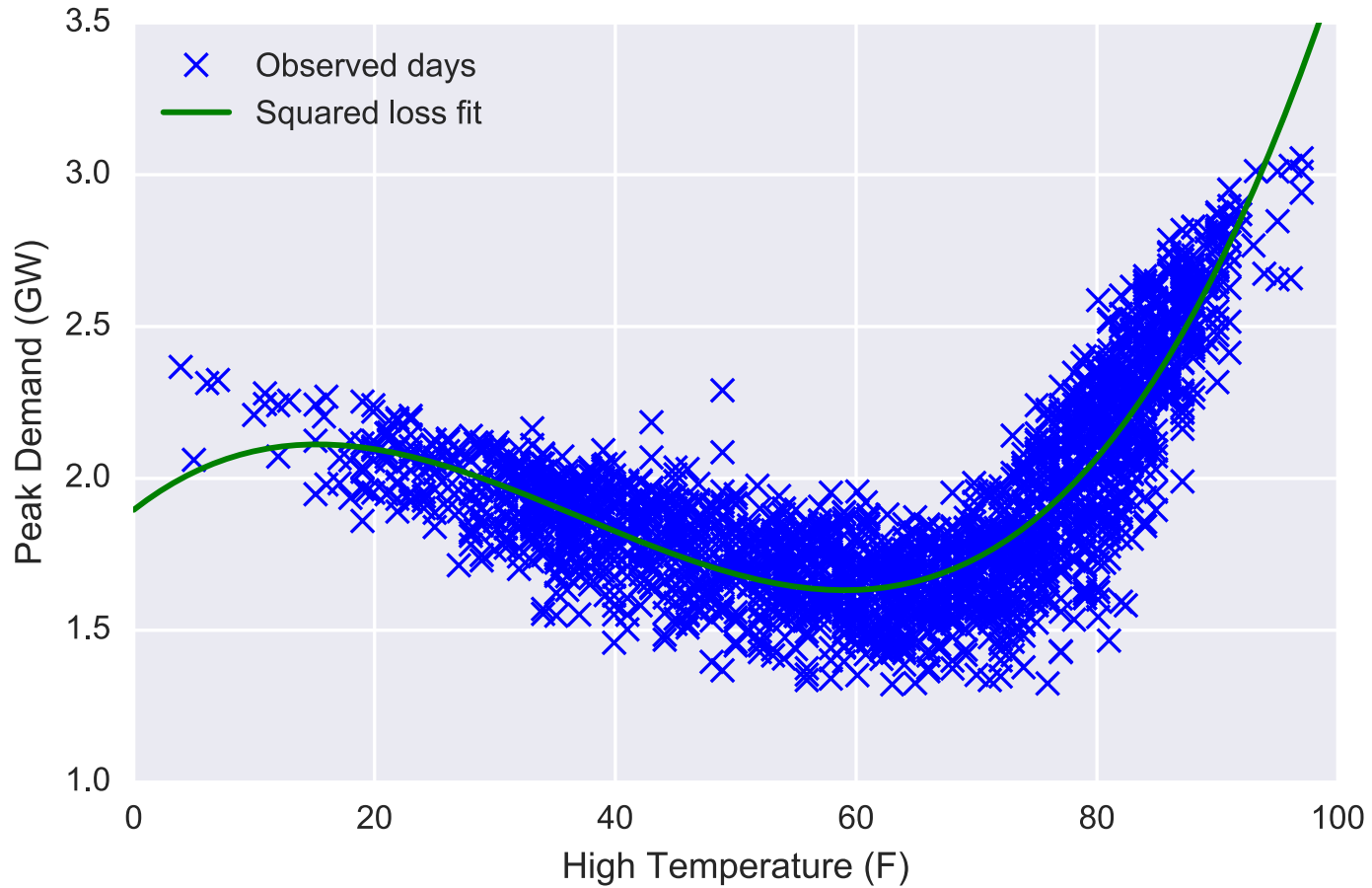
Though it may seem limited, linear regression algorithms are quite powerful when applied to *non-linear features* of the input data, e.g.

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

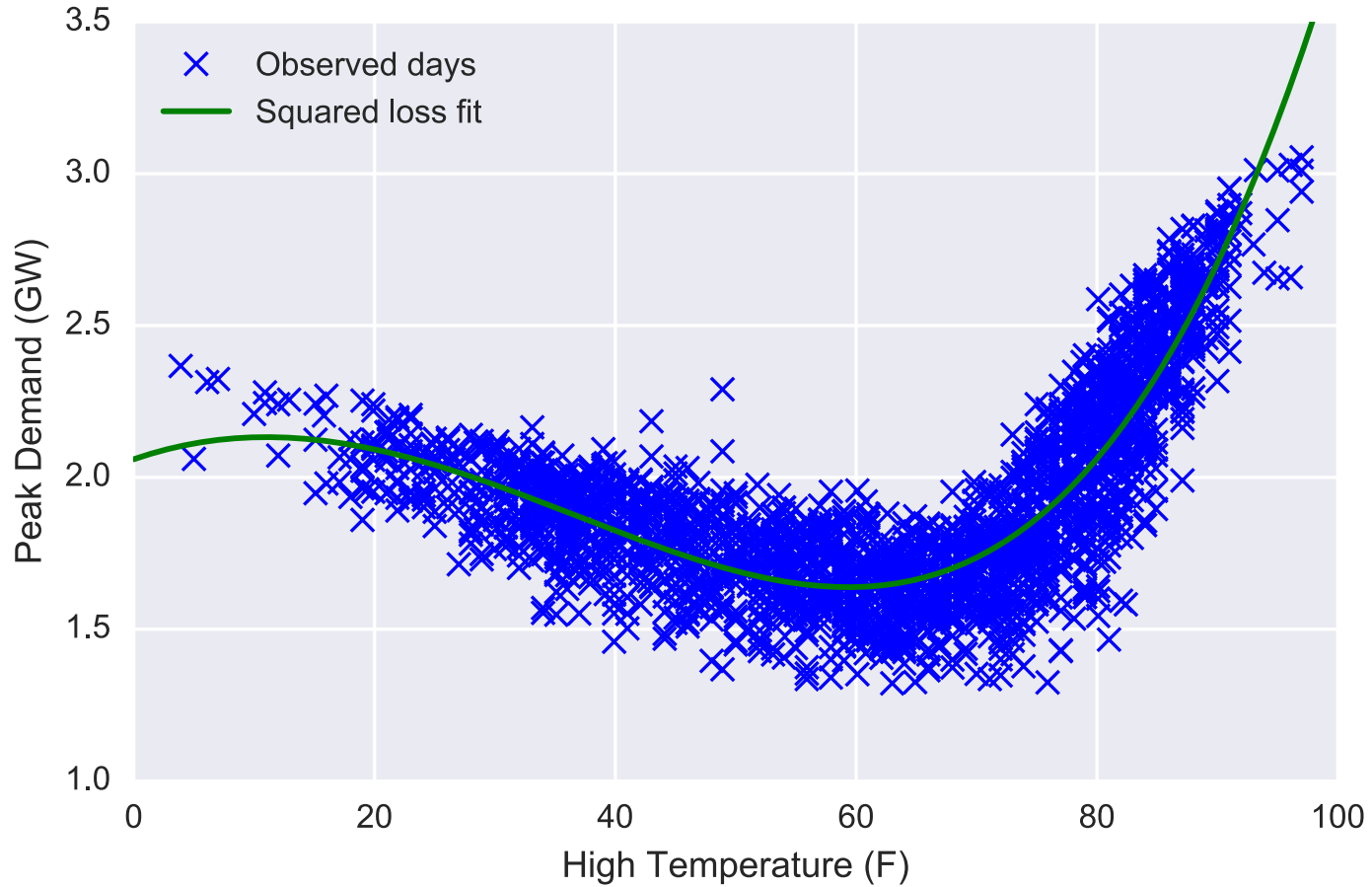
Same hypothesis class as before $h_{\theta}(x) = \theta^T x$, but now prediction will be a non-linear function of base input (e.g. a quadratic function)

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

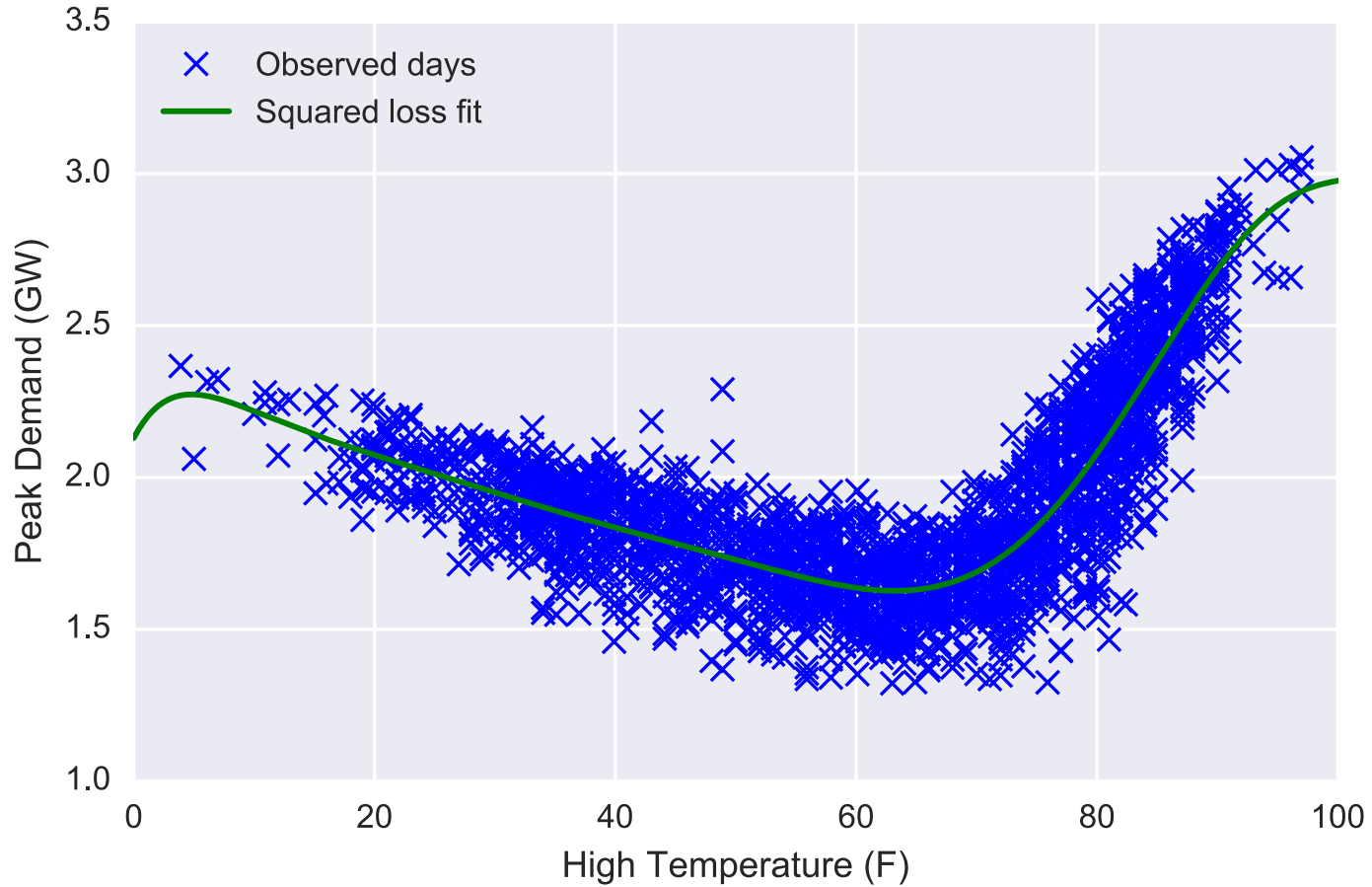
Polynomial features of degree 3



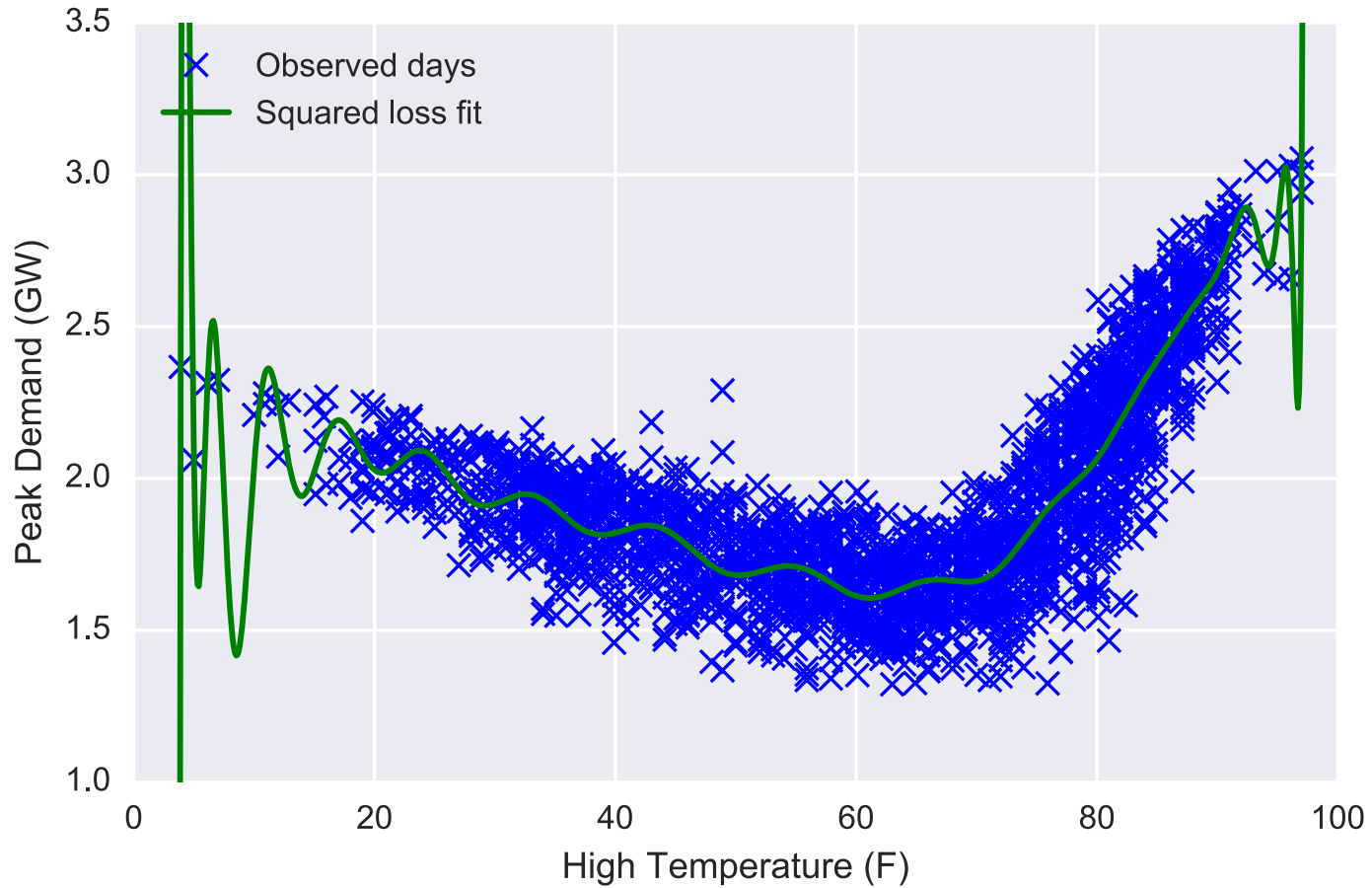
Polynomial features of degree 4



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

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

Generalization error

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

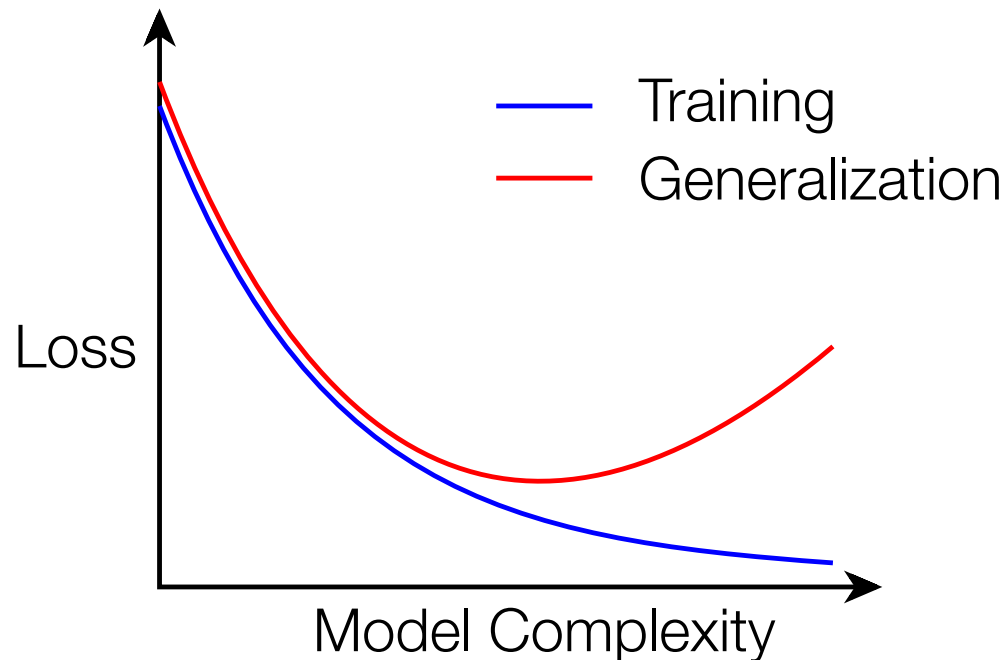
$$\text{minimize}_{\theta} \sum_{i=1}^m \ell(h_{\theta}(x^{(i)}), y^{(i)})$$

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

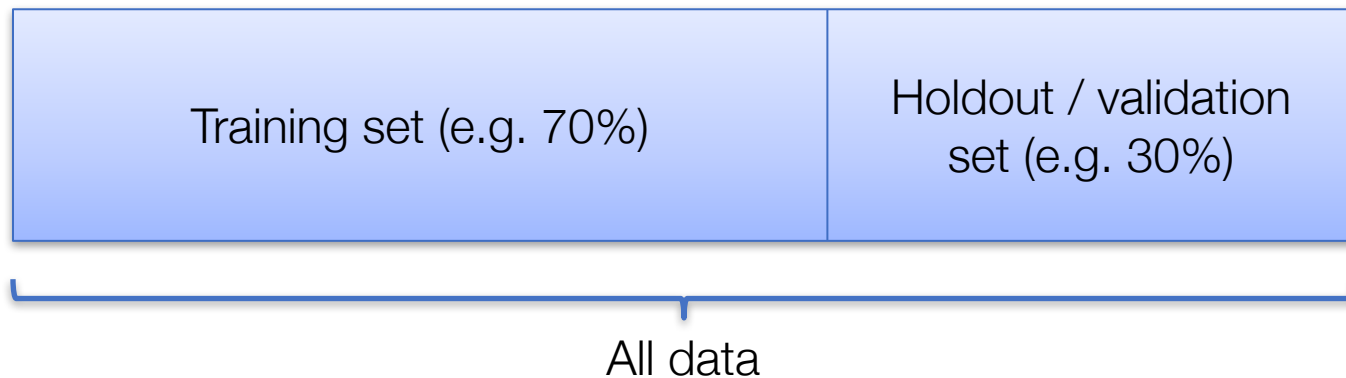
As model becomes more complex, training loss always decreases; generalization loss decreases to a point, then starts to increase



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

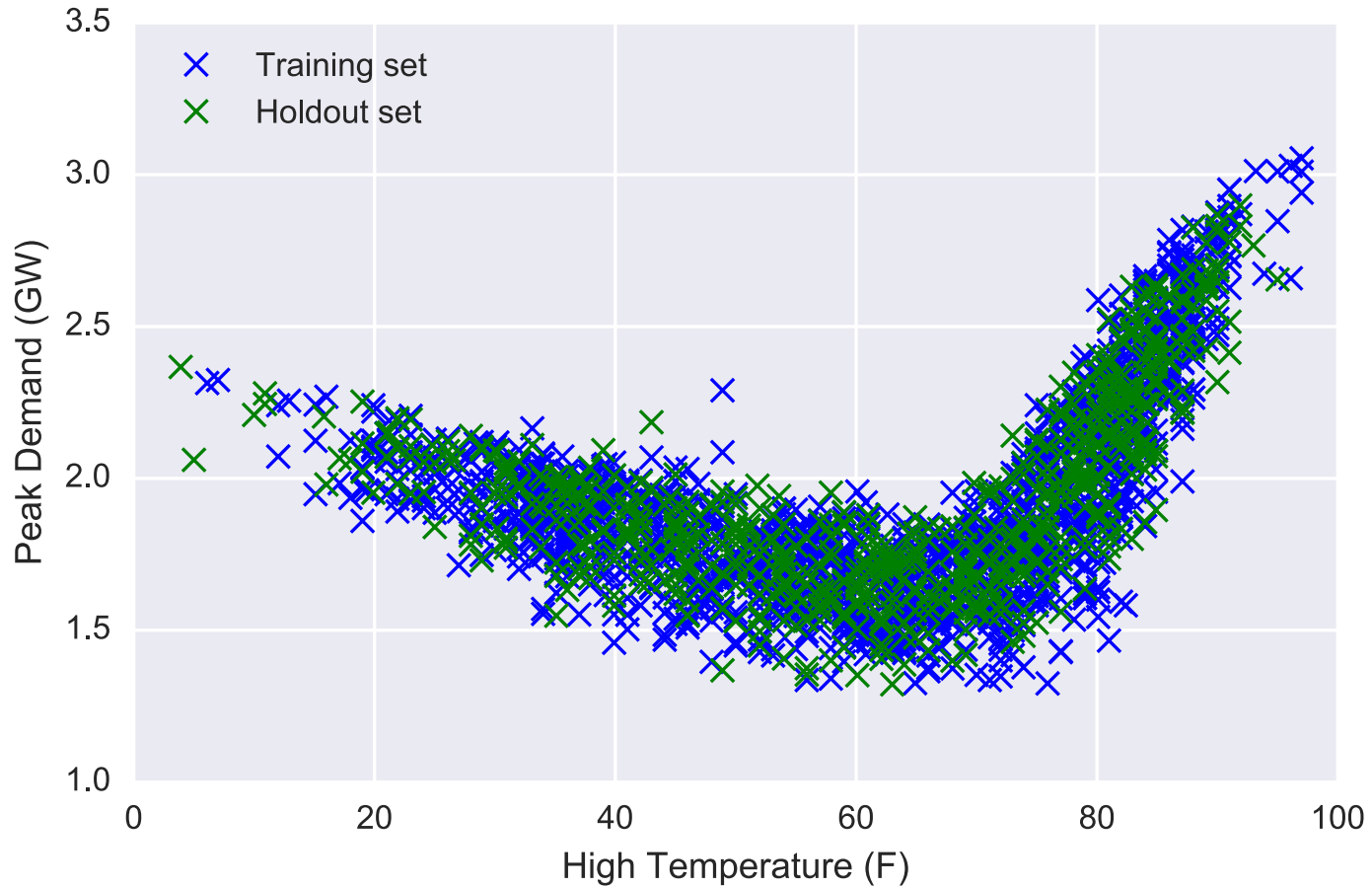
Parameters and hyperparameters

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

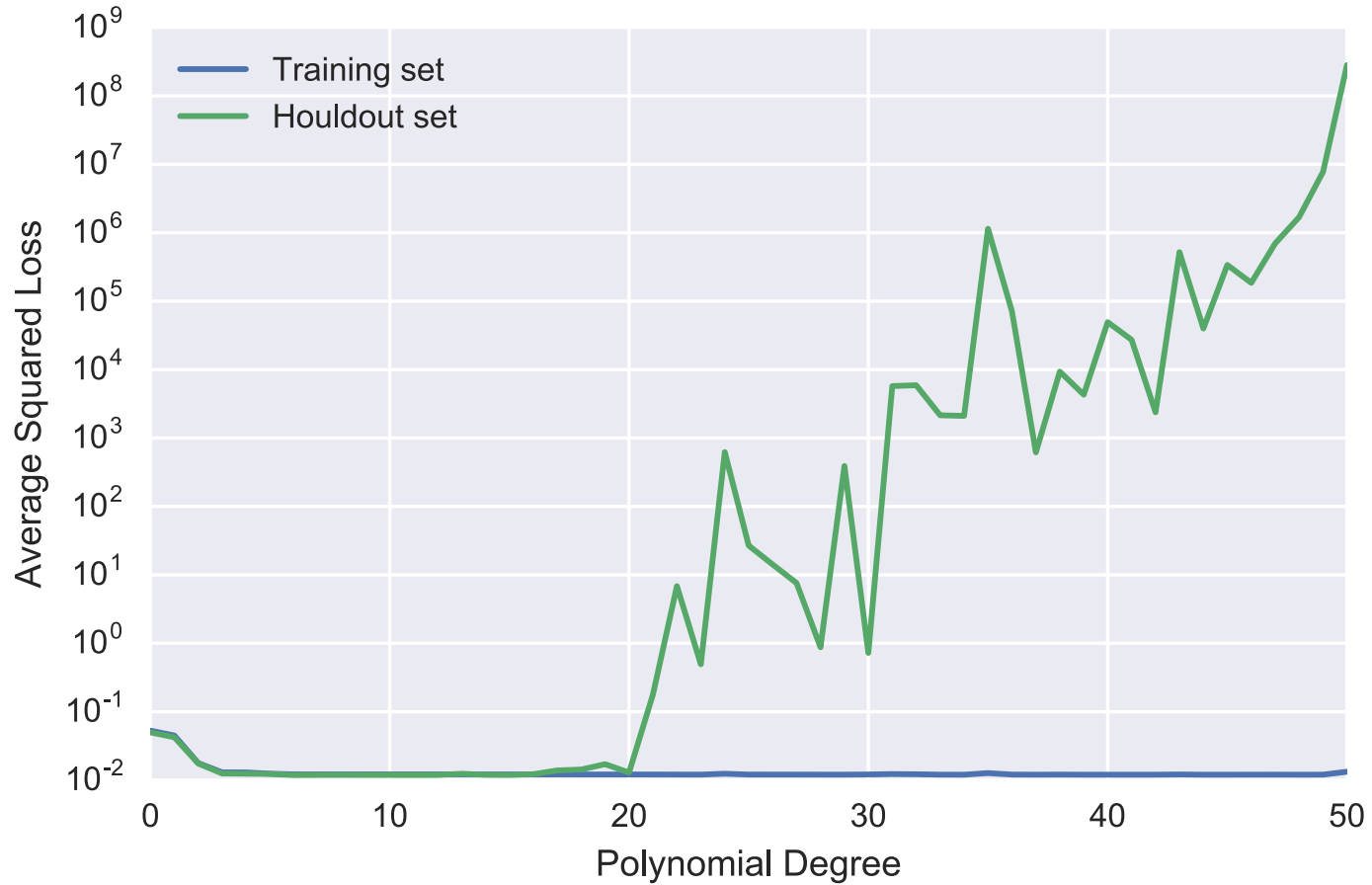
But there are other quantities that also affect the classifier: degree of polynomial, amount of regularization, etc; these are collectively referred to as the *hyperparameters* of the algorithm

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

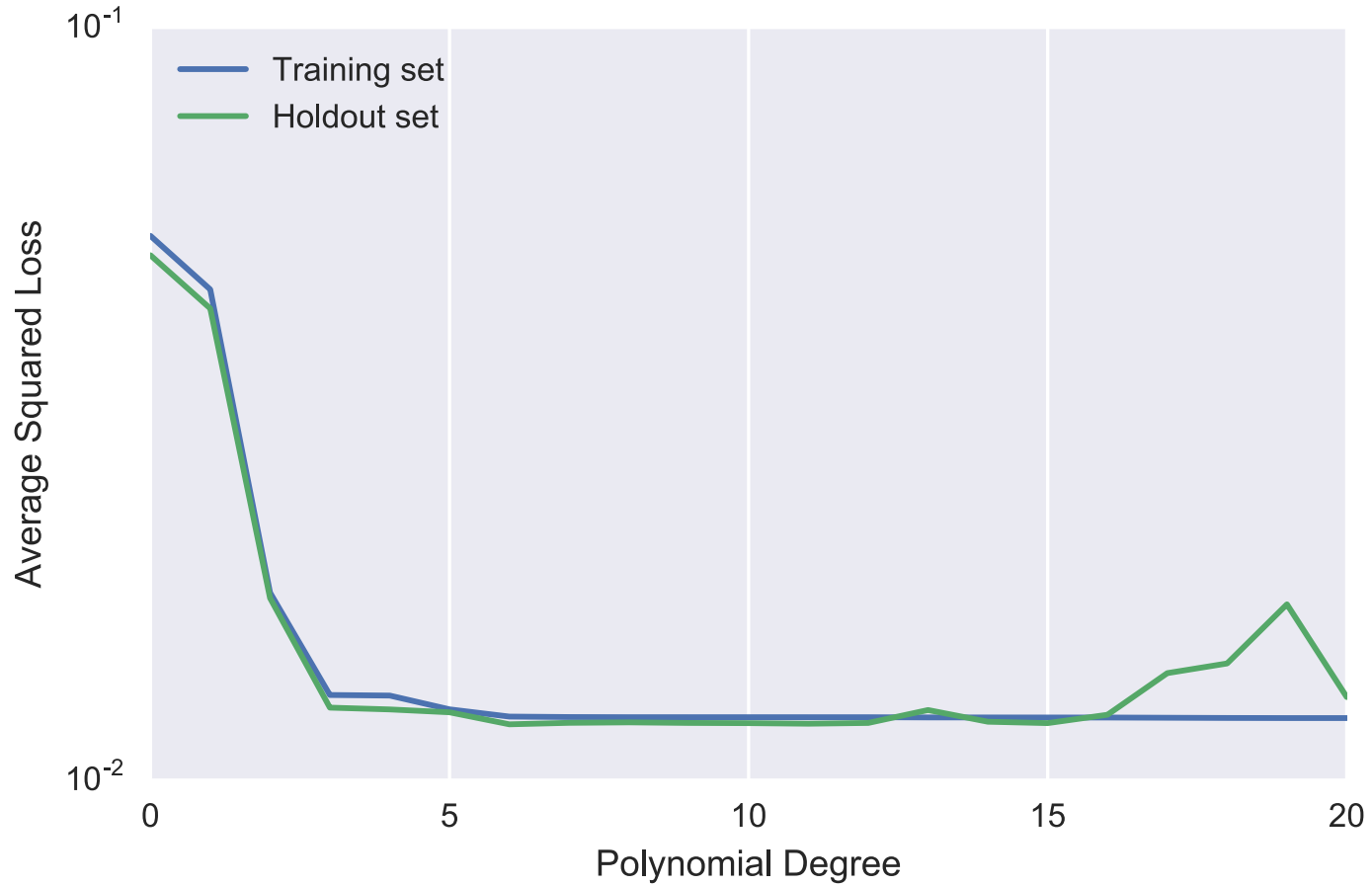
Illustrating cross-validation



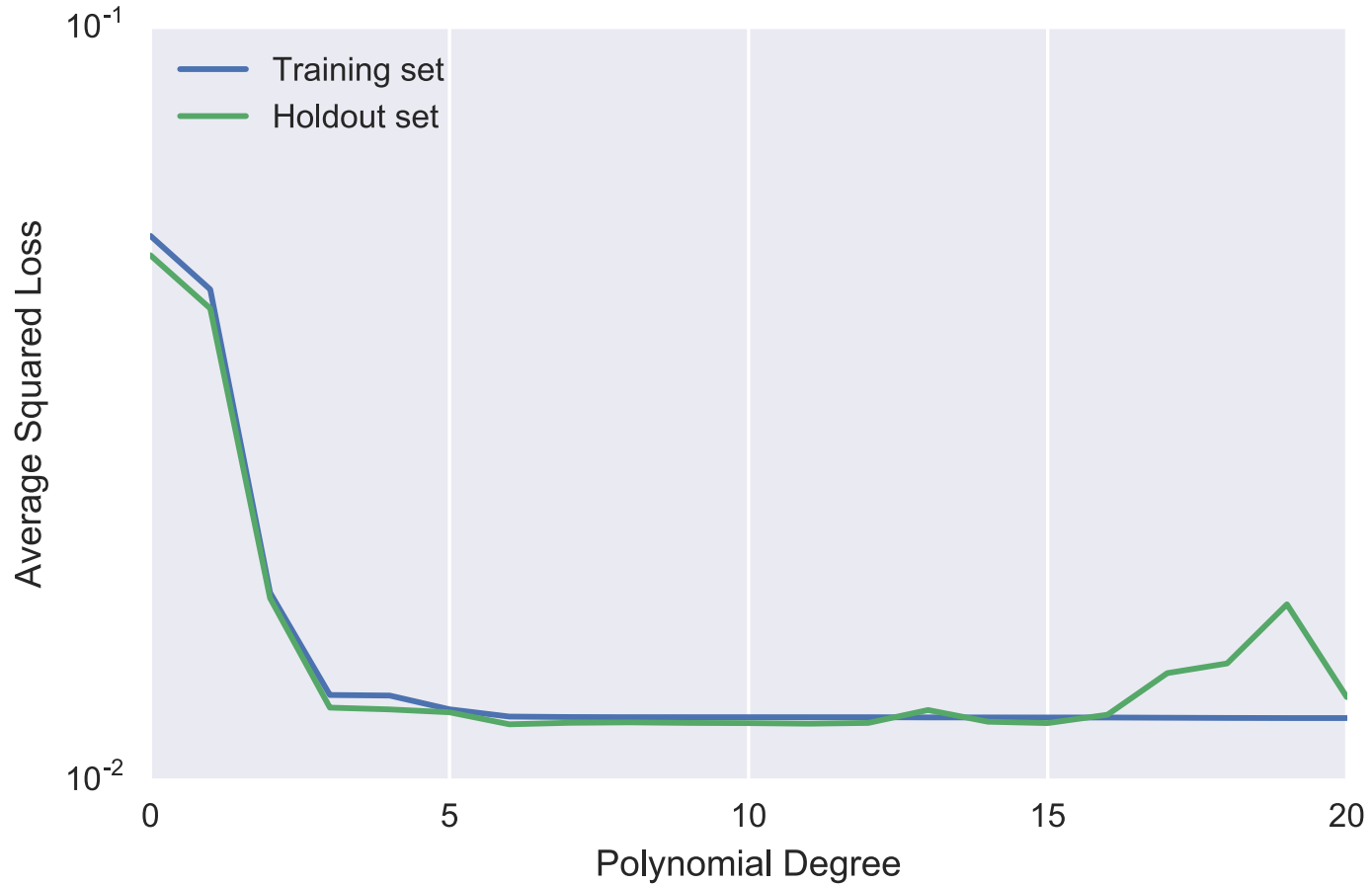
Training and cross-validation loss by degree



Training and cross-validation loss by degree



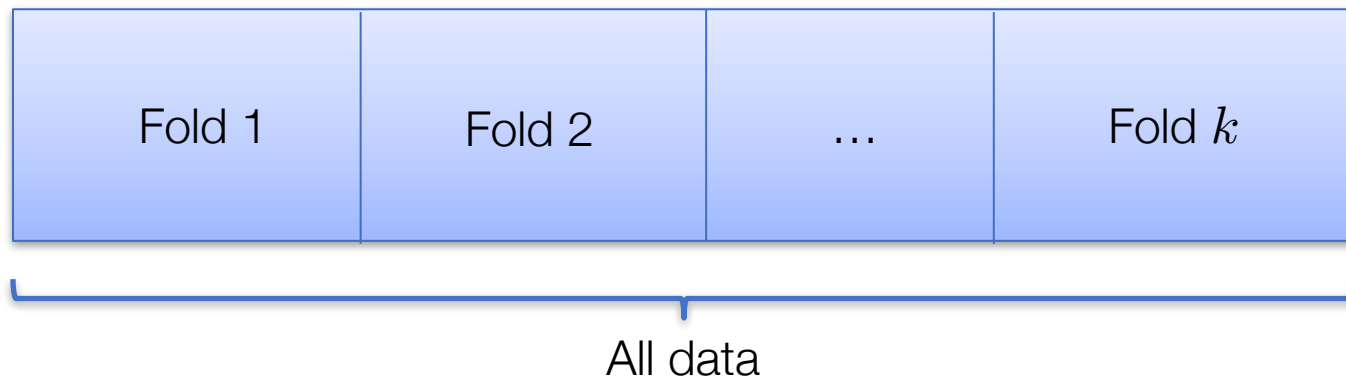
Training and cross-validation loss by degree



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)

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

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

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:

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

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:

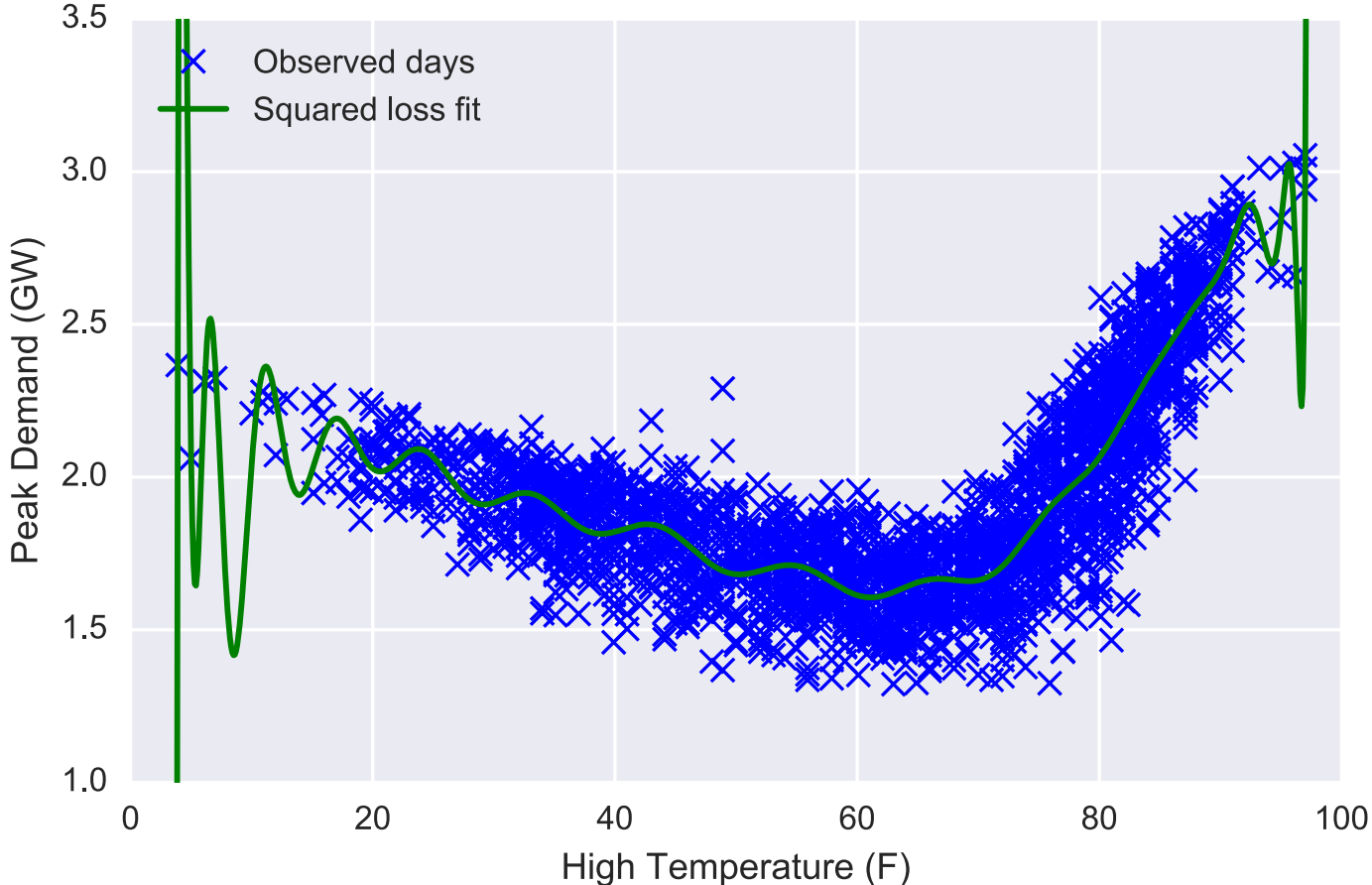
$$\nabla_{\theta} \left(\frac{1}{2} \|X\theta - y\|_2^2 + \frac{\lambda}{2} \|\theta\|_2^2 \right) = X^T (X\theta - y) + \lambda\theta$$

Setting gradient equal to zero leads to the solution

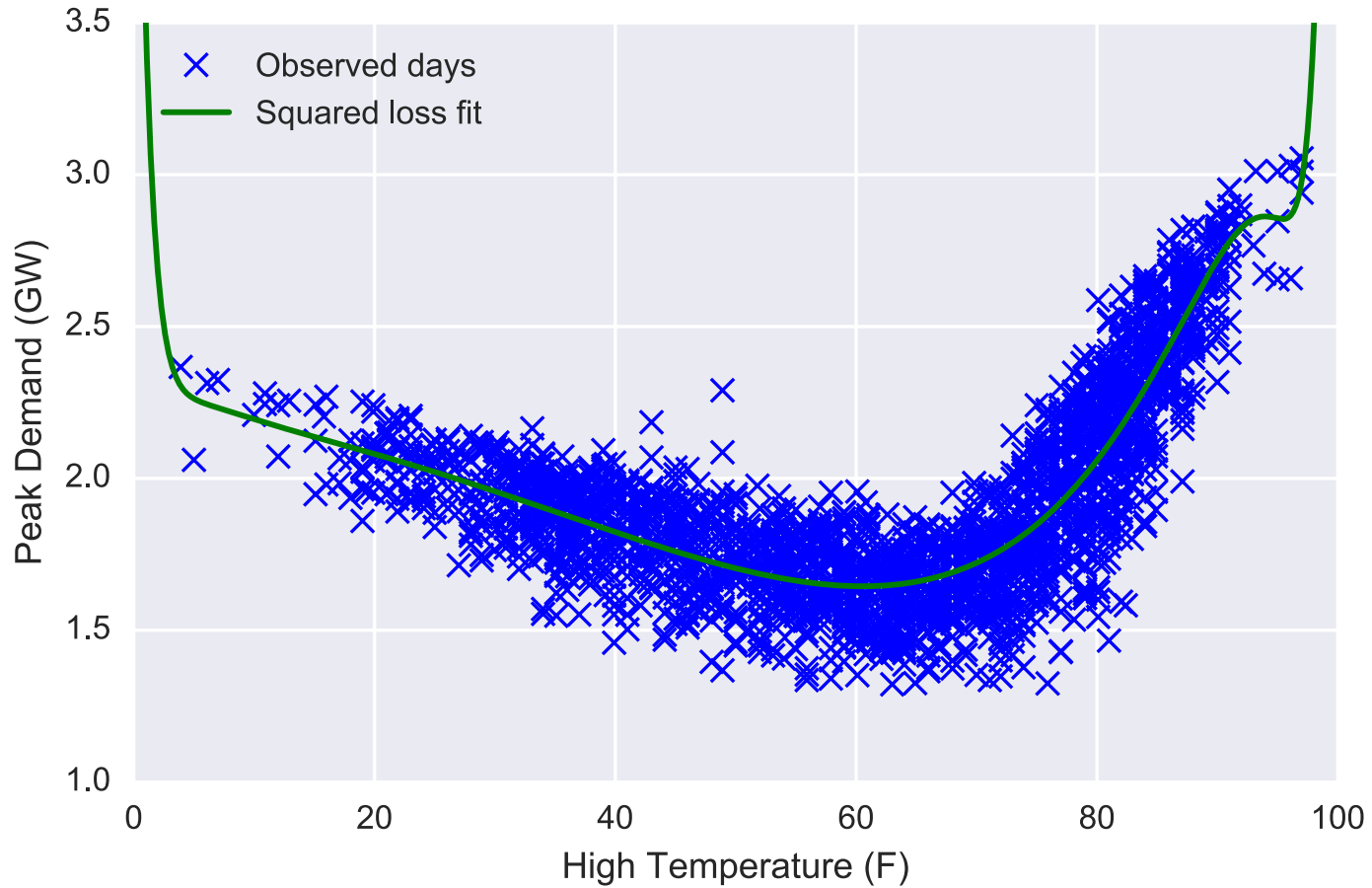
$$X^T X\theta + \lambda\theta = X^T y \implies \theta = (X^T X + \lambda I)^{-1} X^T y$$

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

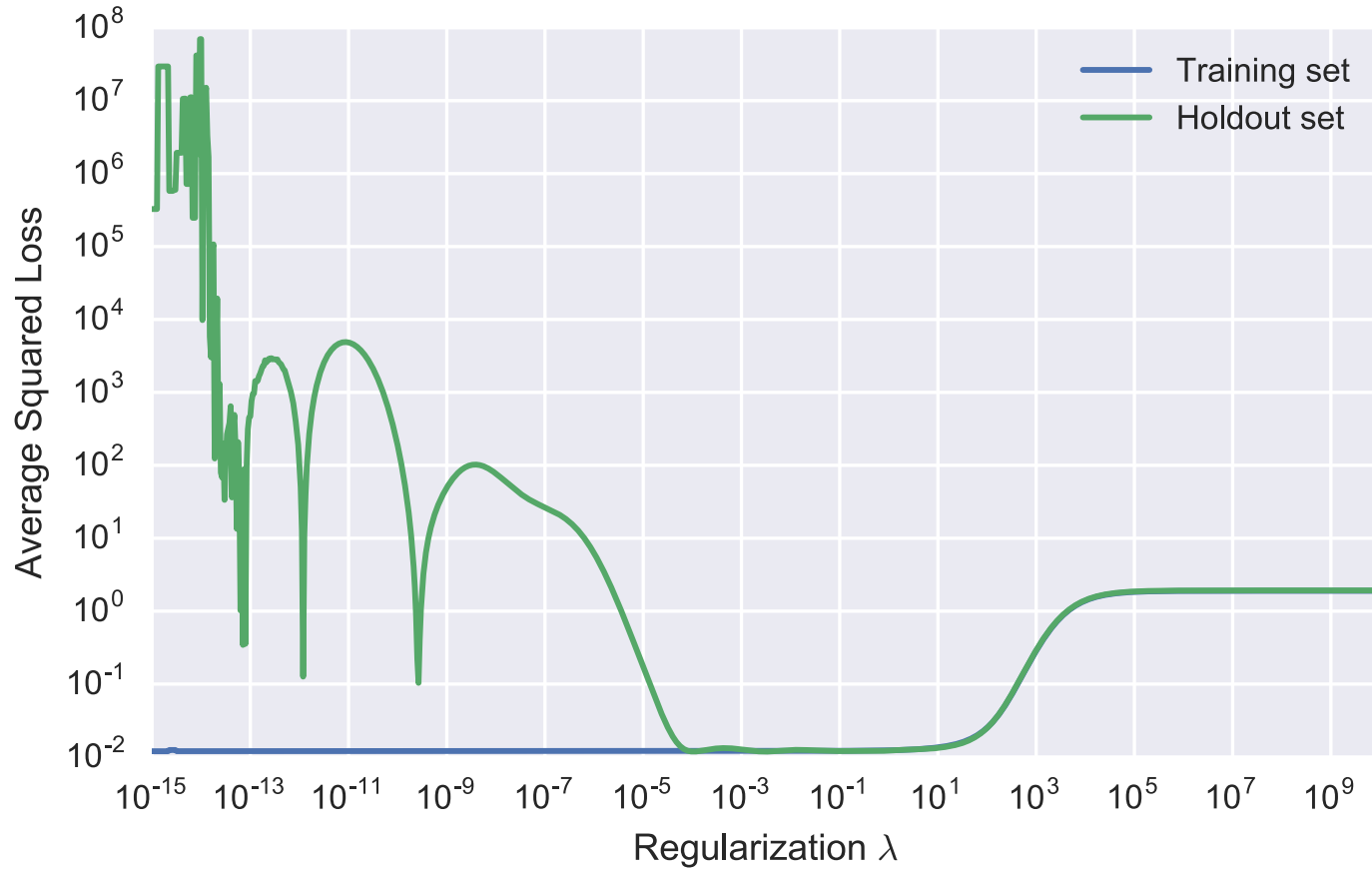
50 degree polynomial fit



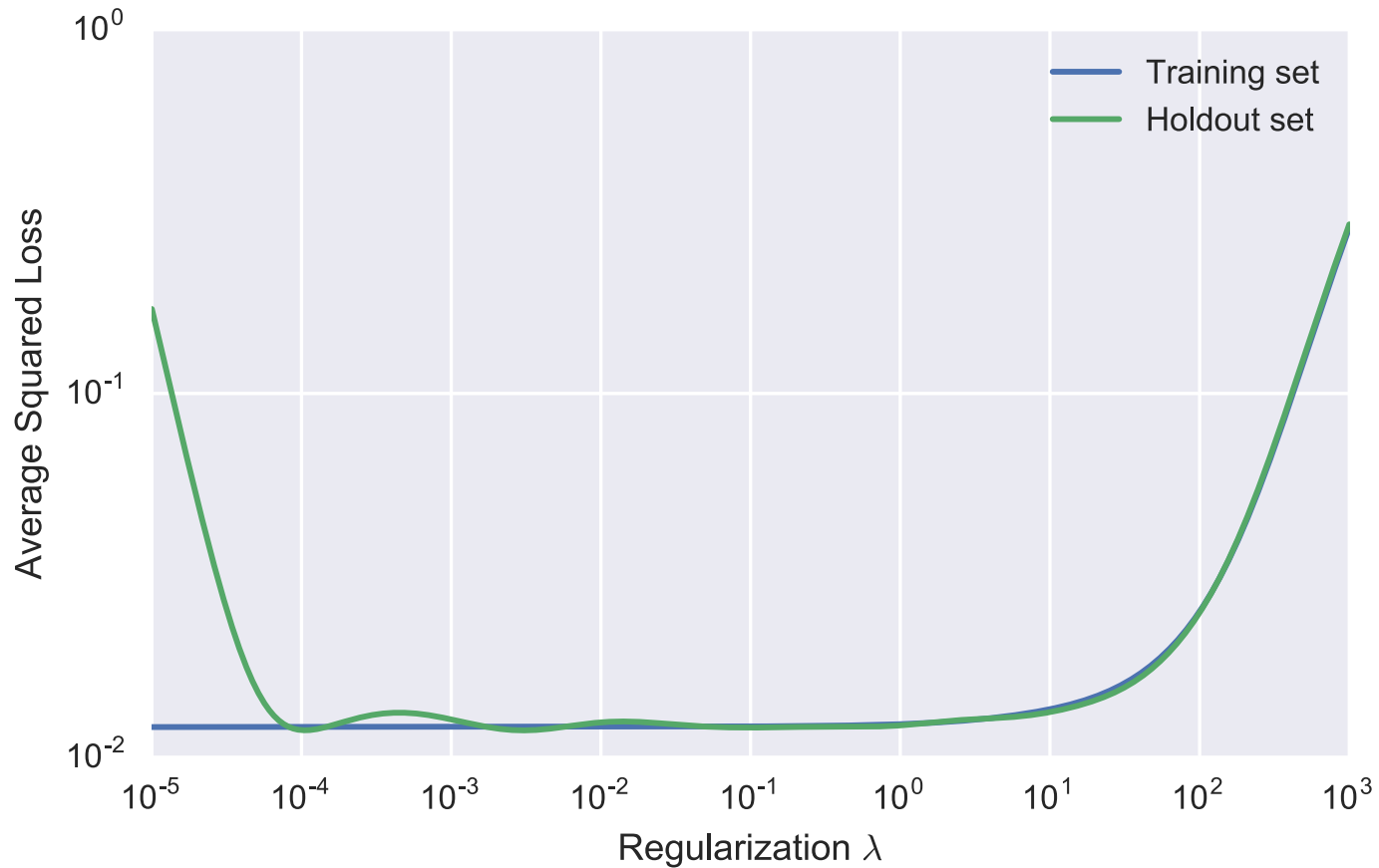
50 degree polynomial fit – $\lambda = 1$



Training/cross-validation loss by regularization



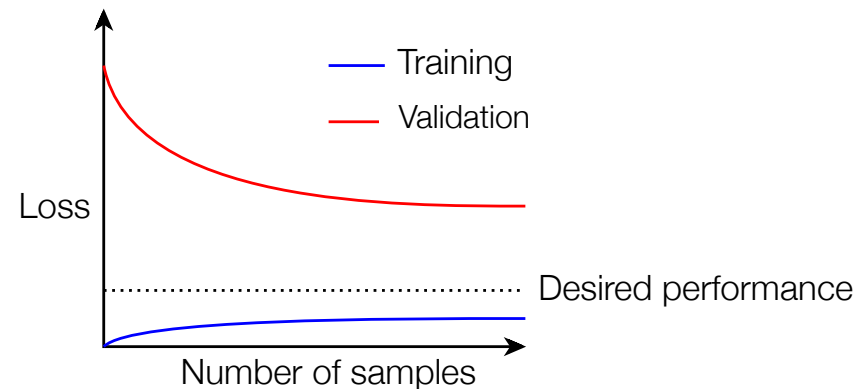
Training/cross-validation loss by regularization



Poll: how do you fix this ML model?

Suppose you run a logistic regression with linear features on some data set, and plot the training/testing performance versus # of samples, which looks like the plot on the right. Which of the following may help?

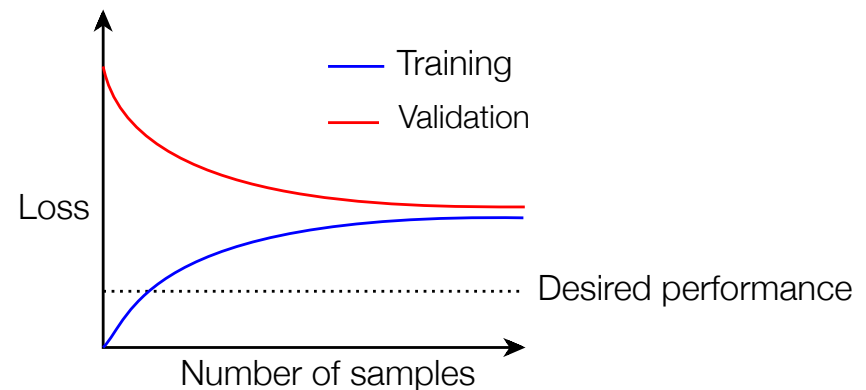
1. Increase regularization parameter
2. Decrease regularization parameter
3. Add non-linear features
4. Remove features
5. Run a neural network



Poll: how do you fix this ML model?

Suppose you run a logistic regression with linear features on some data set, and plot the training/testing performance versus # of samples, which looks like the plot on the right. Which of the following may help?

1. Add more data
2. Decrease regularization parameter
3. Add non-linear features
4. Remove features
5. Run a support vector machine



Nonlinear classification

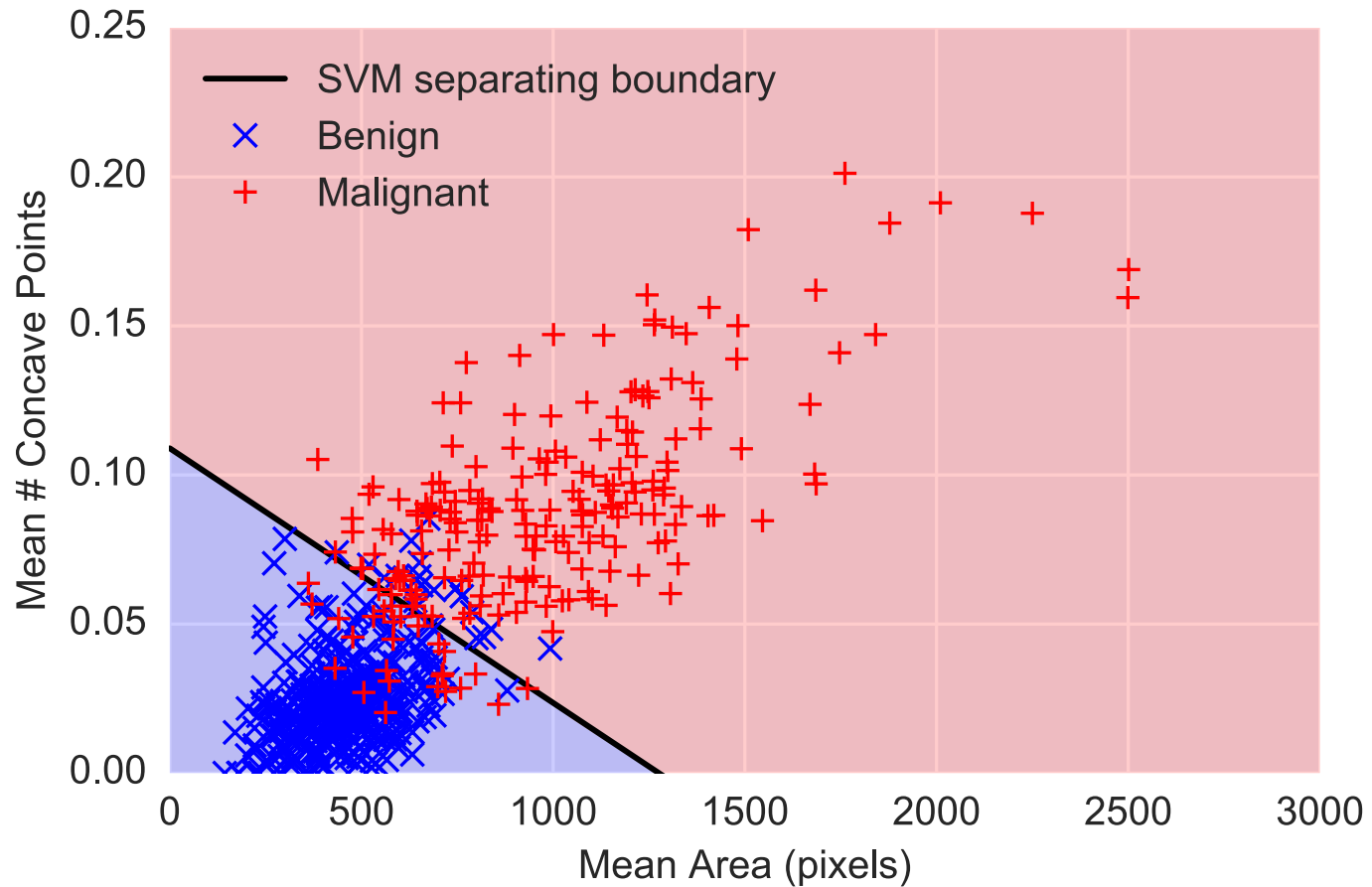
Just like linear regression, the nice thing about using nonlinear features for classification is that our algorithms remain exactly the same as before

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

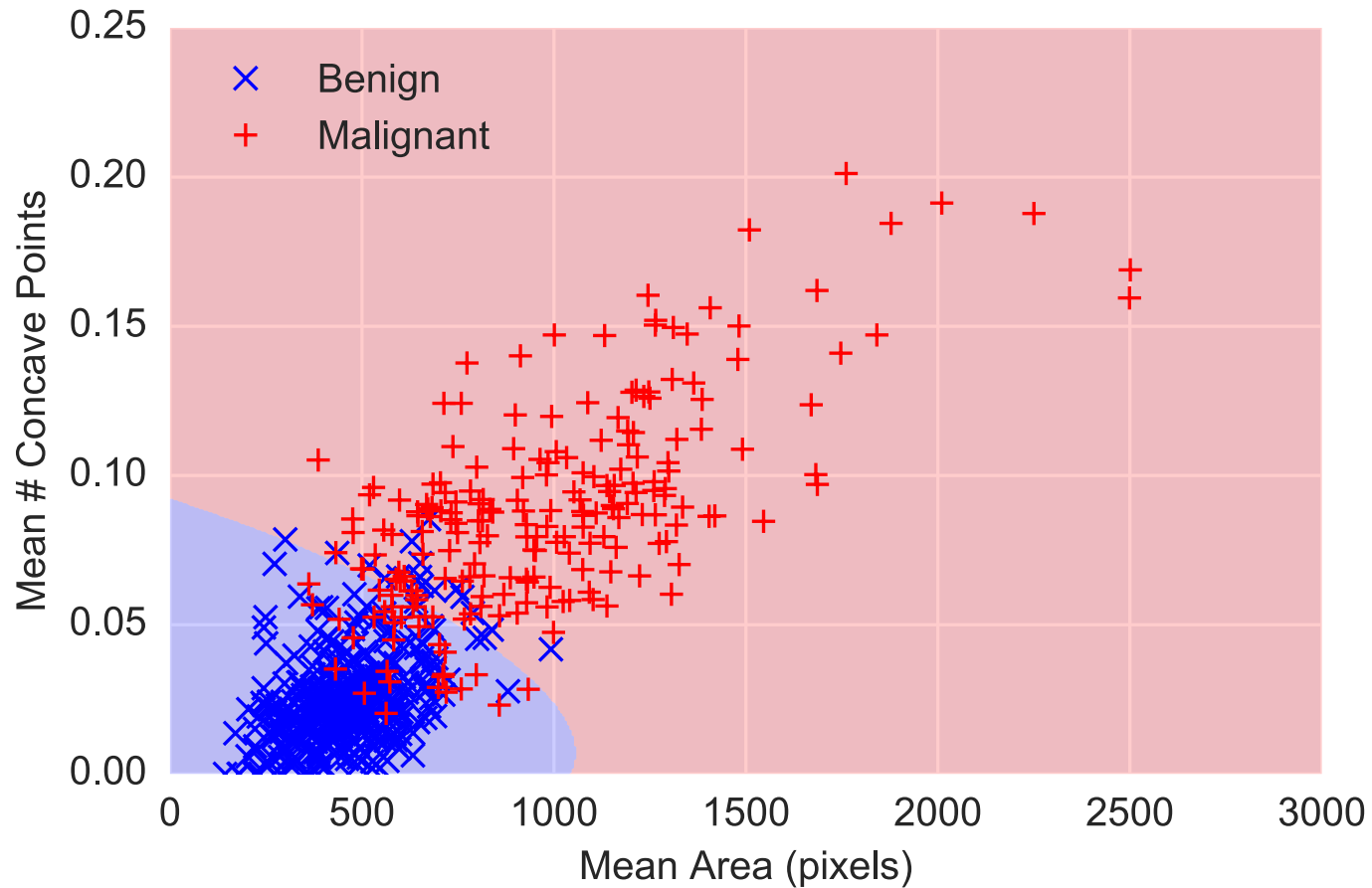
$$\text{minimize}_{\theta} \sum_{i=1}^m \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\} + \frac{\lambda}{2} \|\theta\|_2^2$$

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

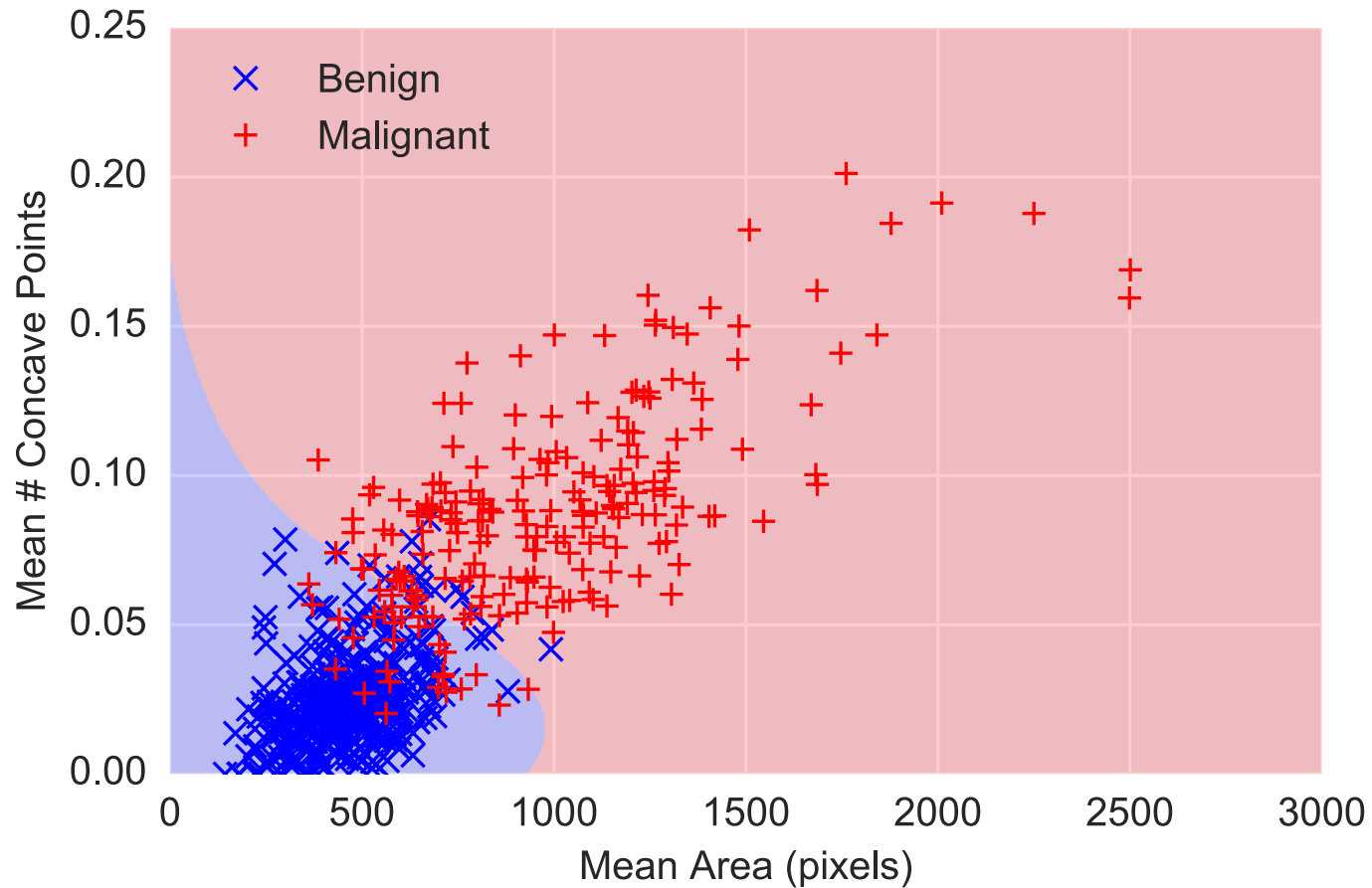
Linear SVM on cancer data set



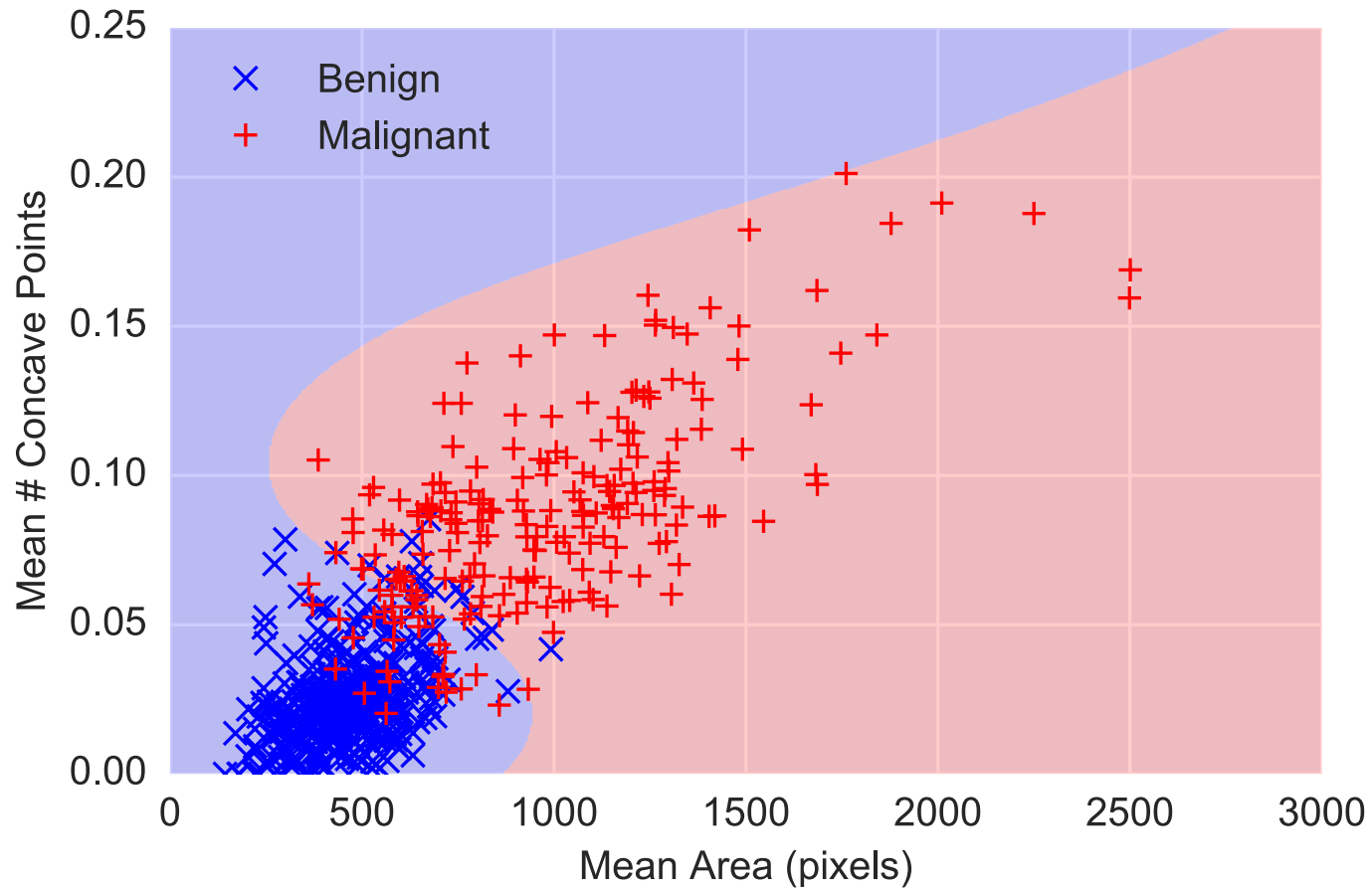
Polynomial features $d = 2$



Polynomial features $d = 3$



Polynomial features $d = 10$



Outline

What is machine learning?

Linear regression

Linear classification

Nonlinear methods

Overfitting, generalization, and regularization

Evaluating machine learning algorithms

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)

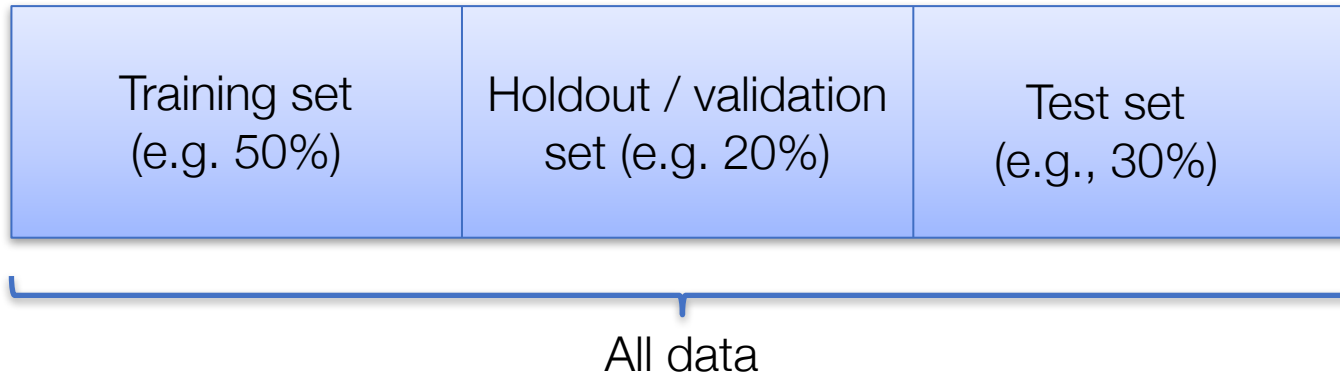
What is wrong with this?

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 as 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)