INTRODUCTION TO DATA SCIENCE

JC + EG

Lecture #18 – 4/5/2022
Lecture #20 – 11/4/2021

CMSC320

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

ANNOUNCEMENTS

- **Project 2 grades are not finalized/out yet!**
- **Project 3 is out**

TODAY'S LECTURE

Decision Trees & Basic Model Evaluation

- What is a decision tree, and what can it represent?
- How do we learn tree structures?
- How do we tell if it's any good in practice?

BIG THANKS to Bart Selman's CS4700 lecture.

Evaluation Methodology General for Machine Learning

Evaluation Methodology

How to evaluate the quality of a learning algorithm, i.e.,:

How good are the hypotheses produced by the learning algorithm?

How good are they at classifying unseen examples?

Standard methodology **("Holdout Cross-Validation")**:

1. Collect a large set of examples.
2. Randomly divide collection into two disjoint sets: **training set** and **test set**.
3. Apply learning algorithm to training set generating hypothesis *h*
4. Measure performance of *h* w.r.t. test set (a form of cross-validation)

   measures generalization to unseen data

Important: keep the training and test sets disjoint! "No peeking"!

Note: The first two questions about any learning result: Can you describe your training and your test set? What's your error on the test set?

Peeking

Example of peeking:

We generate four different hypotheses – for example by using different criteria to pick the next attribute to branch on.

We test the performance of the four different hypothesis on the test set and we select the best hypothesis.

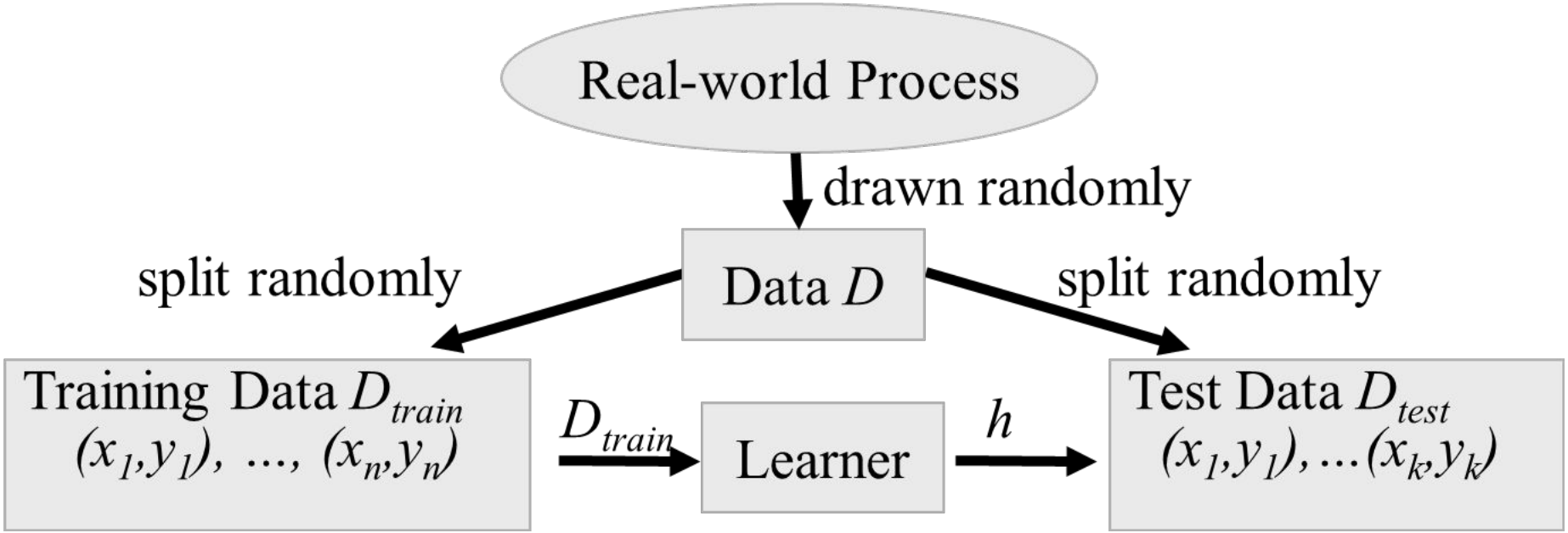
So a new (separate!) test set would be required!

Voila: Peeking occurred! Why?

The hypothesis was selected on the basis of its performance on the **test set**, so information about the test set has leaked into the learning algorithm.

Note: In competitions, such as the "Netflix $1M challenge," test set is not revealed to the competitors. (Data is held back.)

Test/Training Split



Performance Measures

Error Rate

- Fraction (or percentage) of false predictions

Accuracy

- Fraction (or percentage) of correct predictions

Precision/Recall

Example: binary classification problems (classes pos/neg)

- Precision: Fraction (or percentage) of correct predictions among all examples predicted to be positive
- Recall: Fraction (or percentage) of correct predictions among all real positive examples

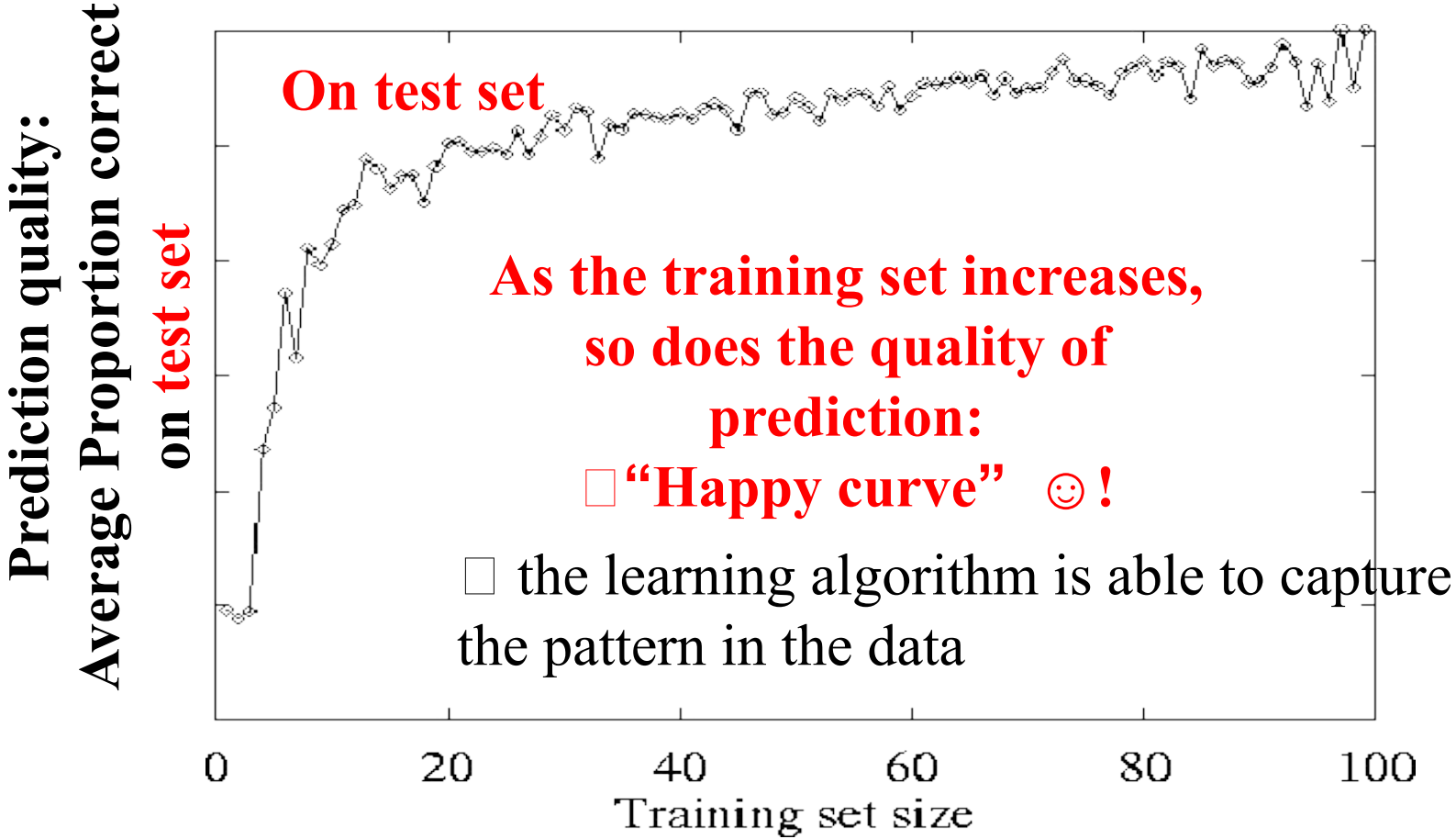
(Can be generalized to multi-class case.)

Learning Curve Graph

Learning curve graph

average prediction quality – proportion correct on test set – as a function of the size of the training set..

Restaurant Example: Learning Curve



Precision vs. Recall

Precision

- # of true positives / (# true positives + # false positives)

Recall

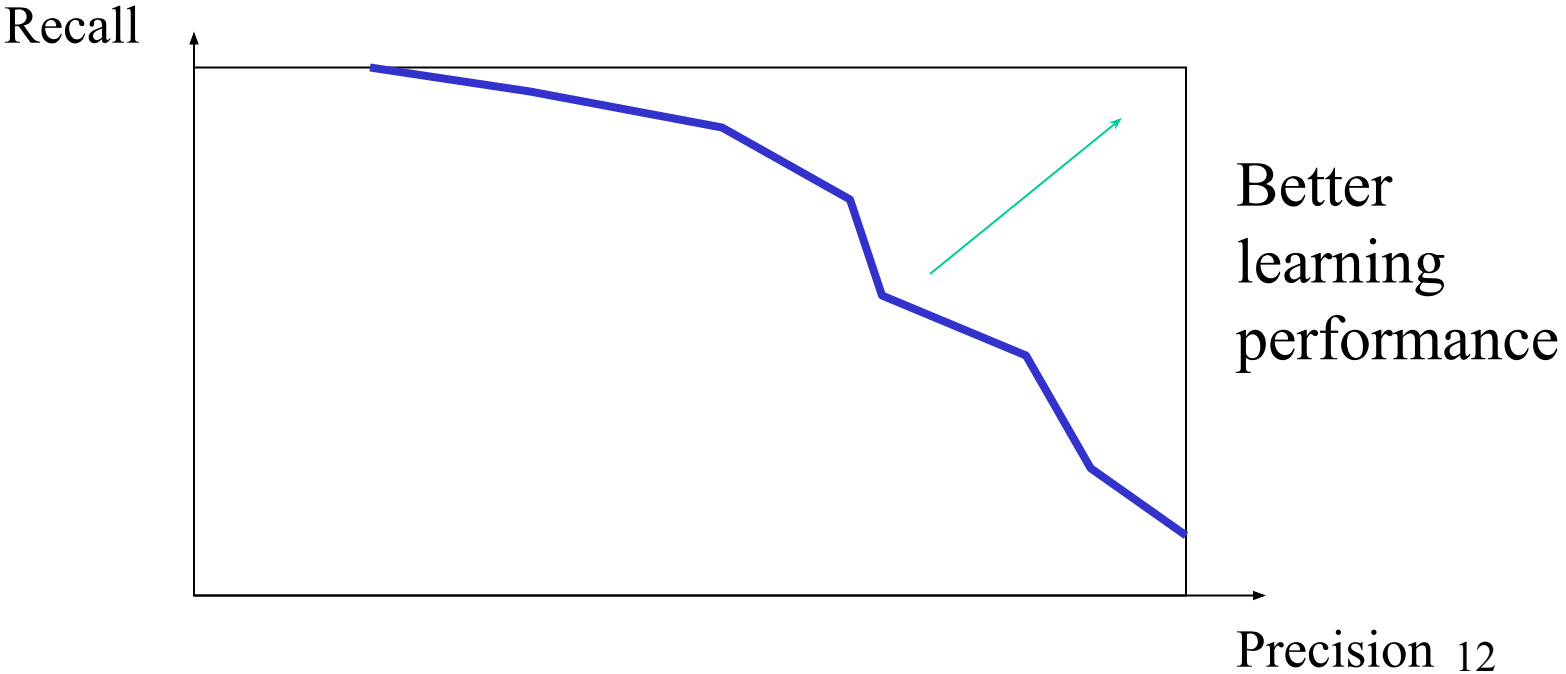
- # of true positives / (# true positives + # false negatives)

A precise classifier is selective

A classifier with high recall is inclusive

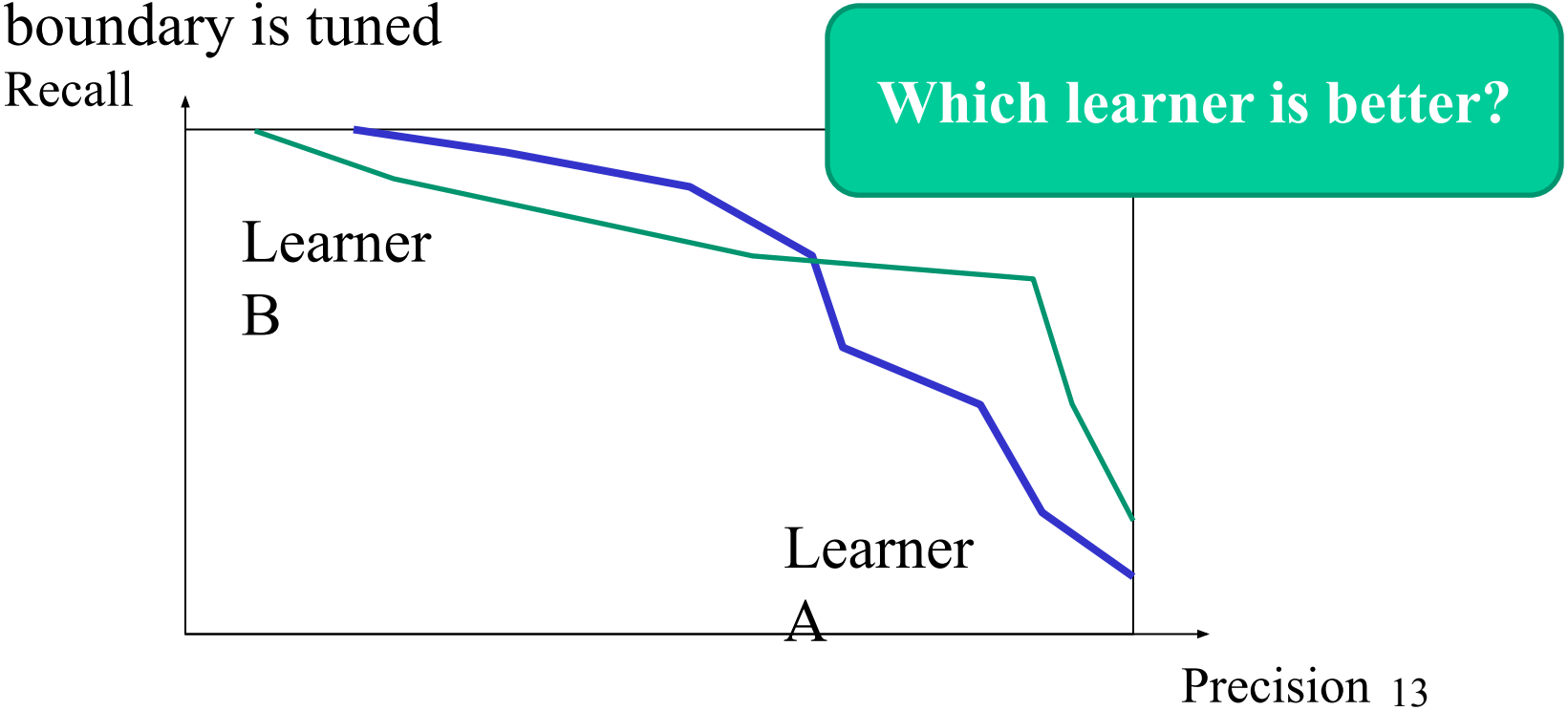
Precision-Recall curves

Measure Precision vs Recall as the classification boundary is tuned



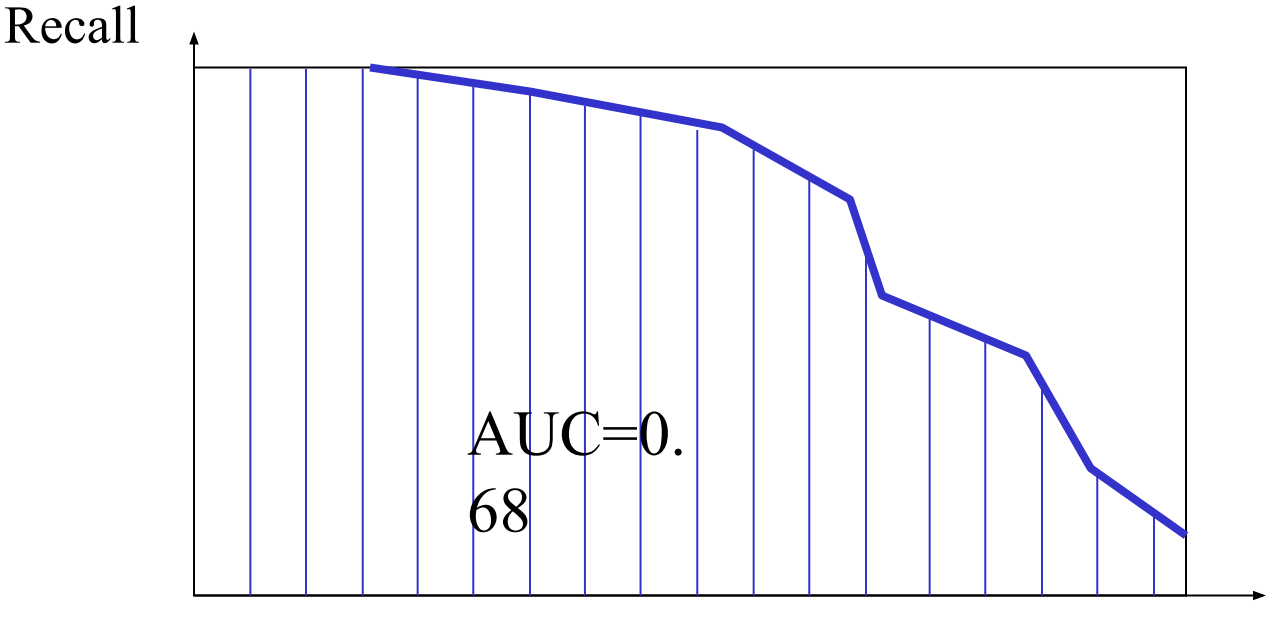
Precision-Recall curves

Measure Precision vs Recall as the classification boundary is tuned



Area Under Curve

AUC-PR: measure the area under the precision-recall curve



AUC metrics

A single number that measures “overall” performance across multiple thresholds

- Useful for comparing many learners
- “Smears out” PR curve

Note training / testing set dependence

How well does it work?

Many case studies have shown that decision trees are at least as accurate as human experts.

- A study for diagnosing breast cancer had humans correctly classifying the examples 65% of the time, and the decision tree classified 72% correct.
- British Petroleum designed a decision tree for gas-oil separation for offshore oil platforms that replaced an earlier rule-based expert system.
- Cessna designed an airplane flight controller using 90,000 examples and 20 attributes per example.

Summary

Decision tree learning is a particular case of supervised learning,

For supervised learning, the aim is to find a simple hypothesis approximately consistent with training examples

Decision tree learning using information gain

Learning performance = prediction accuracy measured on test set

Extensions of the Decision Tree Learning Algorithm (Briefly)

- Noisy data
- Overfitting and Model Selection
- Cross Validation
- Missing Data (R&N, Section 18.3.6)
- Using gain ratios (R&N, Section 18.3.6)
- Real-valued data (R&N, Section 18.3.6)
- Generation of rules and pruning

Noisy data

Many kinds of "noise" that could occur in the examples:

- Two examples have same attribute/value pairs, but different classifications
  - report majority classification for the examples corresponding to the node deterministic hypothesis.
  - report estimated probabilities of each classification using the relative frequency (if considering stochastic hypotheses)
- Some values of attributes are incorrect because of errors in the data acquisition process or the preprocessing phase
- The classification is wrong (e.g., + instead of -) because of some error

Overfitting

Ex.: Problem of trying to predict the roll of a die. The experiment data include:

Day of the week; (2) Month of the week; (3) Color of the die;

….

DTL may find an hypothesis that fits the data but with irrelevant attributes.

Some attributes are irrelevant to the decision-making process, e.g., color of a die is irrelevant to its outcome but they are used to differentiate examples

 Overfitting.

Overfitting means fitting the **training set** "too well"

 performance on the test set degrades.

Example overfitting risk: Using restaurant name.

If the hypothesis space has many dimensions because of a large number of attributes, we may find **meaningless regularity** in the data that is irrelevant to the true, important, distinguishing features.

- **Fix by pruning to lower # nodes in the decision tree or put a limit on number of nodes created.**

- **For example, if Gain of the best attribute at a node is below a threshold, stop and make this node a leaf rather than generating children nodes.**

Overfitting is a key problem in learning. There are formal results on the number of examples needed to properly train an hypothesis of a certain complexity ("number of parameters" or # nodes in DT). The more params, the more data is needed. We'll see some of this in our discussion of PAC learning.

Overfitting

Let's consider D, the entire distribution of data, and T, the training set.

Hypothesis h ∈ H overfits D if

∃ h'≠ h ∈ H such that

$error_T(h) < error_T(h')$ but

$error_D(h) > error_D(h')$

Note: estimate error on full distribution by using test data set.

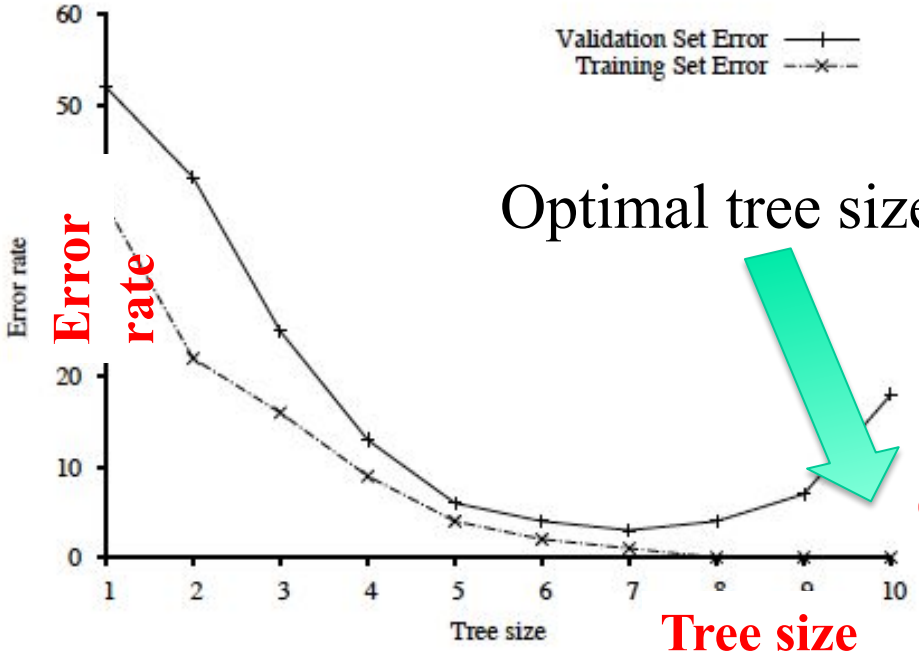
Data overfitting is the arguably the most common pitfall in machine learning.

Why?

1) **Temptation to use as much data as possible to train on. ("Ignore test till end." Test set too small.) Data "peeking" not noticed.**
2) **Temptation to fit very complex hypothesis (e.g. large decision tree). In general, the larger the tree, the better the fit to the training data**
3) **It's hard to think of a better fit to the training data as a "worse" result. Often difficult to fit training data well, so it seems that "a good fit to the training data means a good result."**

Note: Modern "savior:" Massive amounts of data to train on!
Somewhat characteristic of ML AI community vs. traditional statistics community.
Anecdote: Netflix competition.

Key figure in machine learning



Optimal tree size

Tree size

We set tree size as a parameter in our DT learning alg.

Overfitting kicks in…

$error_T(h) < error_T(h')$ but
$error_D(h) > error_D(h')$

Note: with larger and larger trees, we just do better and better on the training set!

But note the performance on the validation set…

Procedure for finding the optimal tree size is called "model selection."
See section 18.4.1 R&N and Fig. 18.8.

To determine validation error for each tree size, use k-fold cross-validation. (Uses the data better than "holdout cross-validation.")

Uses "all data - test set" --- k times splits that set into a training set and a validation set.

After right decision tree size is found from the error rate curve on validation data, train on all training data to get final decision tree (of the right size).

Finally, evaluate tree on the test data (not used before) to get true generalization error (to unseen examples).

Cross Validation

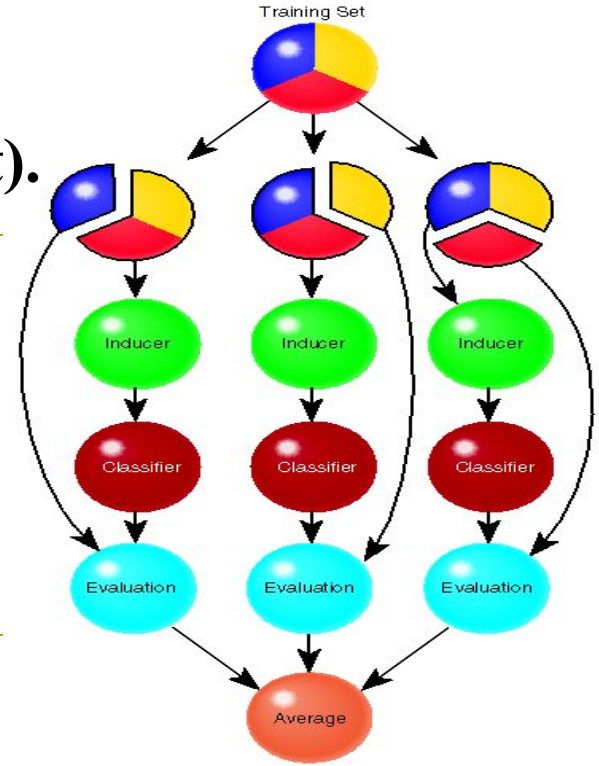
Learner L is e.g. DT learner for "tree with 7 nodes" max.

A method for estimating the accuracy (or error) of a learner (using validation set).

CV(data S, alg L, int k)
  Divide S into k disjoint sets $\{ S_1, S_2, \ldots, S_k \}$
  For i = 1..k do
    Run L on $S_{-i} = S - S_i$
    obtain $L(S_{-i}) = h_i$
    Evaluate h_i on S_i

$$err_{Si}(h_i) = 1/|S_i| \sum_{\langle x,y \rangle \in Si} I(h_i(x) \neq y)$$

  Return Average $1/k \sum_i err_{Si}(h_i)$



Specific techniques for dealing with overfitting
(Model selection provides general framework)

1) **Decision tree pruning or grow only up to certain size.**

Prevent splitting on features that are not clearly relevant.

Testing of relevance of features --- "does split provide new information":

statistical tests ---> Section 18.3.5 R&N χ^2 test.

2) **Grow full tree, then post-prune rule post-pruning**

3) **MDL (minimal description length):**

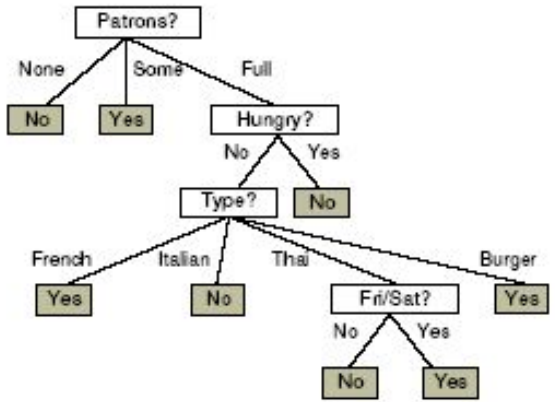
minimize

size(tree) + size(misclassifications(tree))

Converting Trees to Rules

Every decision tree corresponds to set of rules:

- **IF (Patrons = None)**
  THEN WillWait = No
- **IF (Patrons = Full)**
  & (Hungry = No)
  &(Type = French)
  THEN WillWait = Yes
- **...**



Fighting Overfitting: Using Rule Post-Pruning

1. Grow decision tree. Fit as much data as possible. Allow overfitting.
2. Convert tree to equivalent set of rules. One rule for each path from root to leaf.
3. Prune (generalize) each rule independently of others. i.e. delete preconditions that improve its accuracy.
4. Sort final rules into desired sequence for use depending on accuracy.
5. Use ordered sequence for classification.

This is the strategy of the most successful commercial decision tree learning method (C4.5 — Quinlan 1993). Widely used in data mining.

What is advantage of rule representation over the decision tree?

Logical aside

Decision trees are a restricted from of general logical statements.

We can also describe our target function directly in first-order sentences.

Example:

$$\forall WillWait(r) \Leftrightarrow Patrons(r, Some)$$
$$\vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, French))$$
$$\vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, Thai) \wedge Fri/Sat($$
$$\vee(Patrons(r, Full) \wedge \neg Hungry(r) \wedge Type(r, Burger))$$

This is our **hyposthesis** H_r. In general, we search from among a space of hypotheses:

$$H_1 \vee H_2 \vee H_3 \vee \ldots \vee H_n.$$

End logical aside

Here's an **example** in logical form:

$$Alternate(X1) \wedge \neg Bar(X_1) \wedge \neg Fri/Sat(X_1) \wedge$$
$$Hungry(X_1) \wedge \ldots \wedge WillWait(X_1)$$

We can test if this example is **consistent** with our hyposthesis. If not, we may have to **generalize** or **specialize** our hypothesis:

Current-best-hypothesis search.

Summary: When to use Decision Trees

Instances presented as attribute-value pairs

Method of approximating discrete-valued functions

Target function has discrete values: classification problems

Robust to noisy data:

Training data may contain

- errors
- missing attribute values

Typical bias: prefer smaller trees (Ockham's razor)

Widely used, practical and easy to interpret results

Inducing decision trees is one of the most widely used learning methods in practice

Can outperform human experts in many problems

Strengths include

- Fast
- simple to implement
- human readable
- can convert result to a set of easily interpretable rules
- empirically valid in many commercial products
- handles noisy data

Can be a legal requirement! Why?

Weaknesses include:

- "Univariate" splits/partitioning using only one attribute at a time so limits types of possible trees
- large decision trees may be hard to understand
- requires fixed-length feature vectors
- non-incremental (i.e., batch method)

DECISION TREES IN SCIKIT

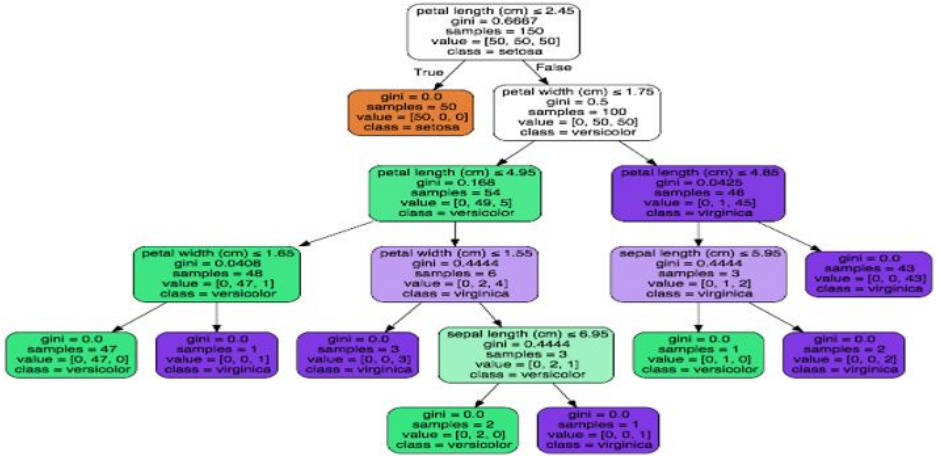
```
from sklearn.datasets import load_iris
from sklearn import tree

# Load a common dataset, fit a decision tree to it
iris = load_iris()
clf = tree.DecisionTreeClassifier()
clf = clf.fit(iris.data, iris.target)
```

Trains a decision tree using default parameters (attribute chosen to split on either Gini or entropy, no max depth, etc)

```
# Predict most likely class
clf.predict([[2., 2.]])
# Predict PDF over classes (%training samples in leaf)
clf.predict_proba([[2., 2.]])
```

VISUALIZING A DECISION TREE

```
from IPython.display import Image
dot_data = tree.export_graphviz(clf,
                out_file=None,
                feature_names=iris.feature_names,
                class_names=iris.target_names,
                filled=True, rounded=True)
graph = pydotplus.graph_from_dot_data(dot_data)
Image(graph.create_png())
```



RANDOM FORESTS

Decision trees are very interpretable, but may be brittle to changes in the training data, as well as noise

Random forests are an ensemble method that:

- Resamples the training data;
- Builds many decision trees; and
- Averages predictions of trees to classify.

This is done through bagging and random feature selection

BAGGING

Bagging: Bootstrap aggregation

Resampling a training set of size n via the bootstrap:

- Sample with replacement n elements

General scheme for random forests:

1. Create B bootstrap samples, $\{Z_1, Z_2, \ldots, Z_B\}$
2. Build B decision trees, $\{T_1, T_2, \ldots, T_B\}$, from $\{Z_1, Z_2, \ldots, Z_B\}$

Classification/Regression:

1. Each tree T_j predicts class/value y_j
2. Return average $1/B \, \Sigma_{j=\{1,\ldots,B\}} \, y_j$ for regression, or majority vote for classification

Original training dataset (Z):

obs_id	ft_1	ft_2
1	12.2	puppy
2	34.5	dog
3	8.1	cat

B Bootstrap samples Z_j

Z_1

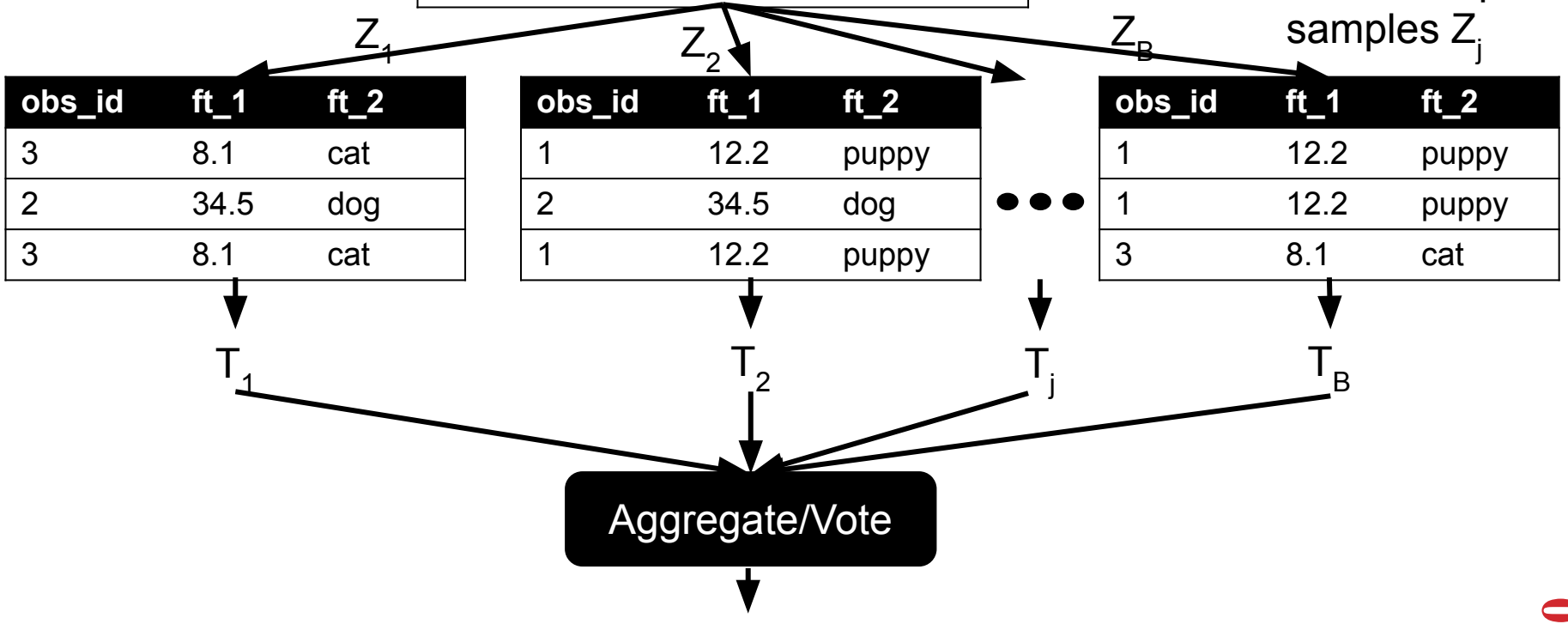
obs_id	ft_1	ft_2
3	8.1	cat
2	34.5	dog
3	8.1	cat

Z_2

obs_id	ft_1	ft_2
1	12.2	puppy
2	34.5	dog
1	12.2	puppy

Z_B

obs_id	ft_1	ft_2
1	12.2	puppy
1	12.2	puppy
3	8.1	cat



RANDOM ATTRIBUTE SELECTION

We get some randomness via bootstrapping

- We like this! Randomness increases the bias of the forest slightly at a huge decrease in variance (due to averaging)

We can further reduce correlation between trees by:

1. **For each tree, at every split point …**
2. **… choose a random subset of attributes …**
3. **… then split on the "best" (entropy, Gini) within only that subset**

RANDOM FORESTS IN SCIKIT-LEARN

```
from sklearn.ensemble import RandomForestClassifier

# Train a random forest of 10 default decision trees
X = [[0, 0], [1, 1]]
Y = [0, 1]
clf = RandomForestClassifier(n_estimators=10)
clf = clf.fit(X, Y)
```

Can we get even more random?!

Extremely randomized trees (`ExtraTreesClassifier`) **do bagging, random attribute selection, but also:**

1. **At each split point, choose random splits**
2. **Pick the best of those random splits**

Similar bias/variance performance to RFs, but can be faster computationally

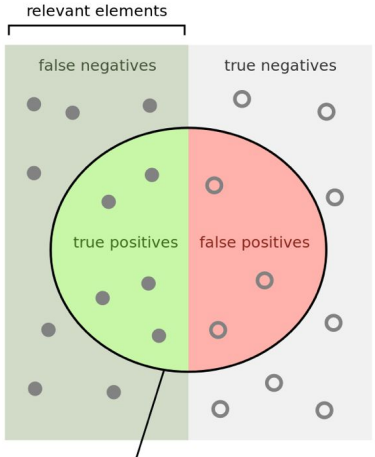
QUICK ASIDE FOR PROJECT #3

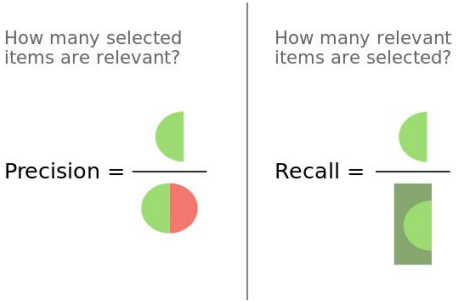
Precision P:

#correct positive results / #positive results returned

Recall R:

#correct positive results / #all possible positive results





QUICK ASIDE FOR PROJECT #3

F-Score F:
weighted average of the precision and recall of a test

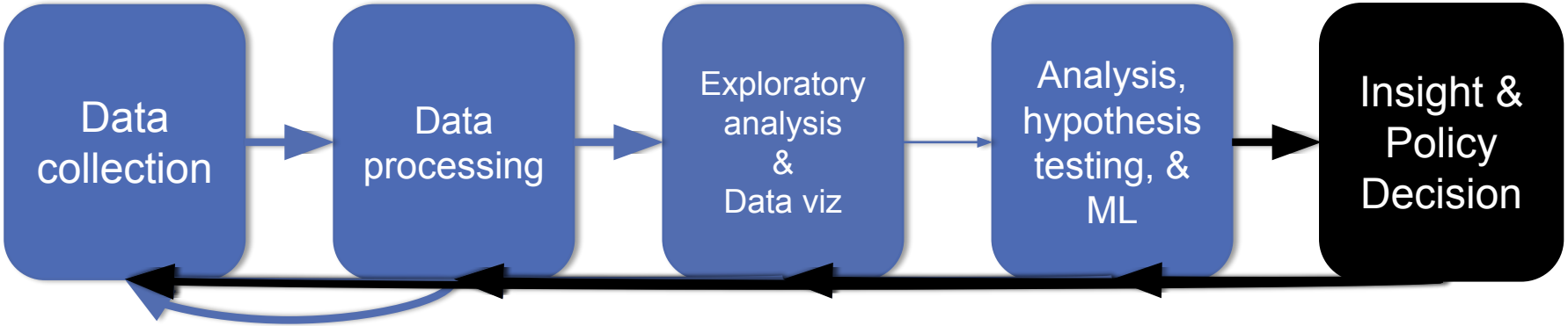
F_1: (harmonic) mean of precision and recall:

$$F_1 = 2 \cdot \frac{1}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{(\beta^2 \cdot \text{precision}) + \text{recall}}$$

Can be parameterized to attach higher importance to recall:

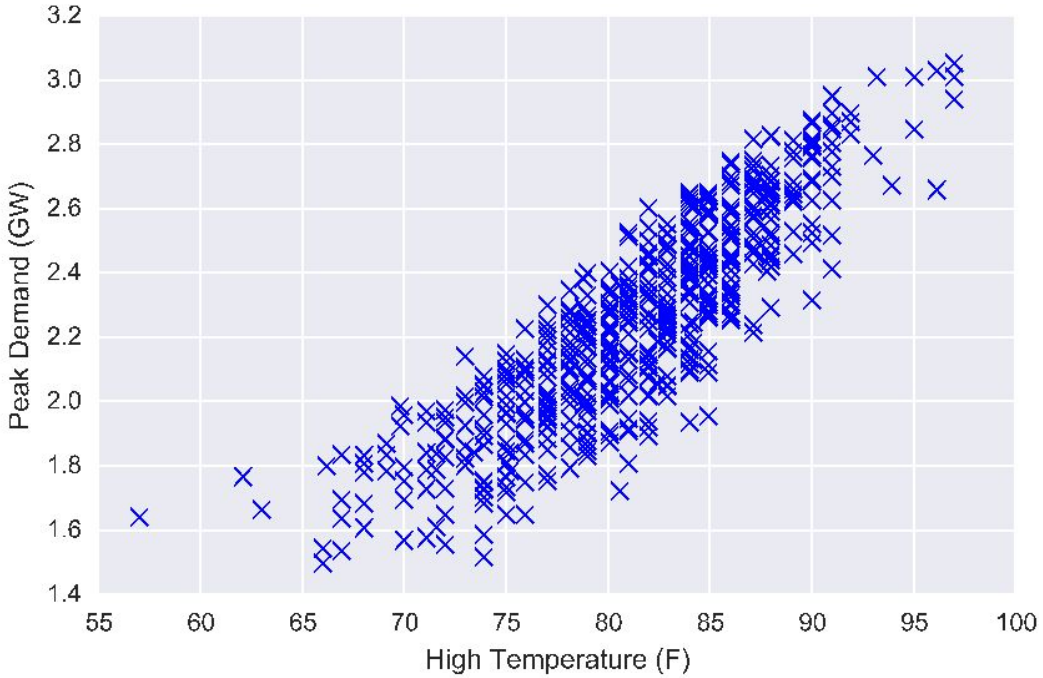
TODAY'S LECTURE



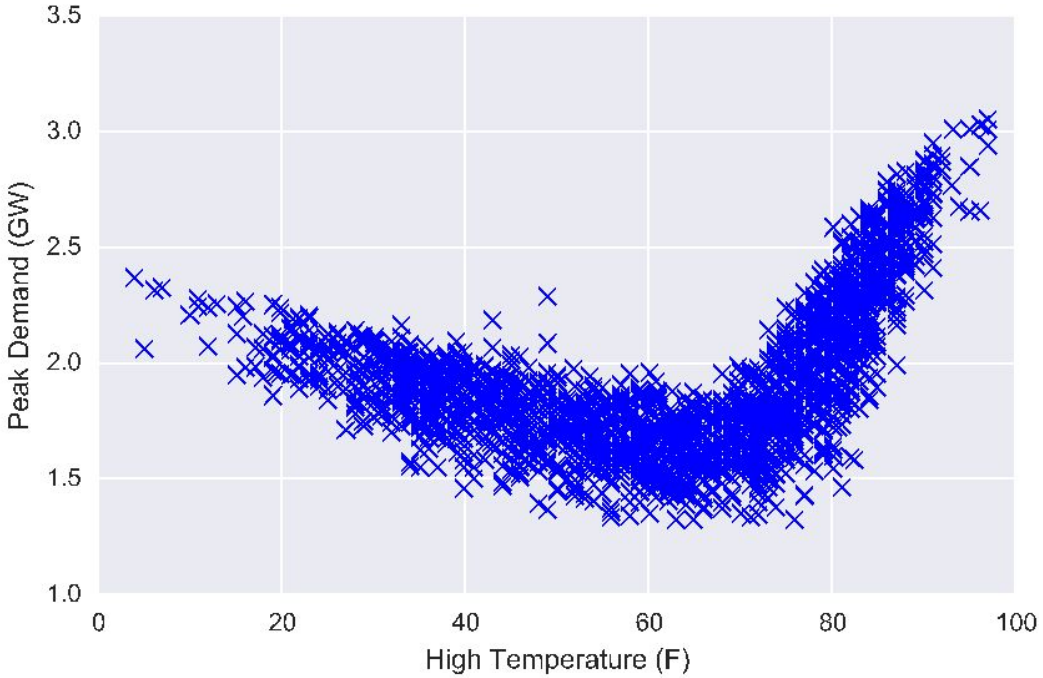
FILLING IN THE GAPS: NONLINEAR REGRESSION & REGULARIZATION

Thanks: Zico Kolter

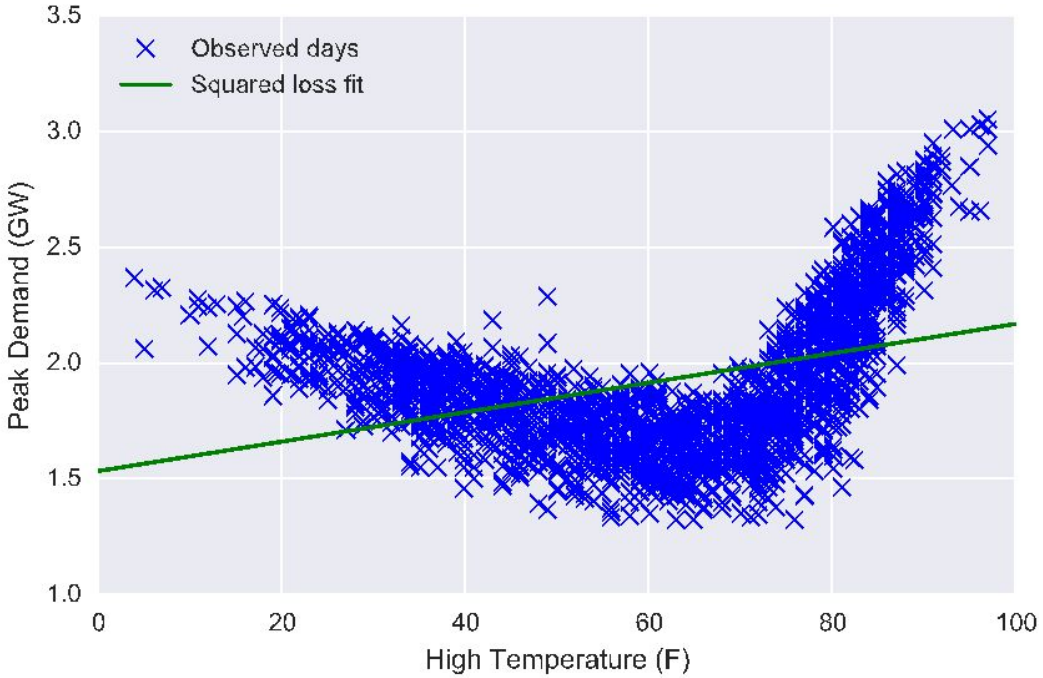
Peak demand vs. temperature (summer months)



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

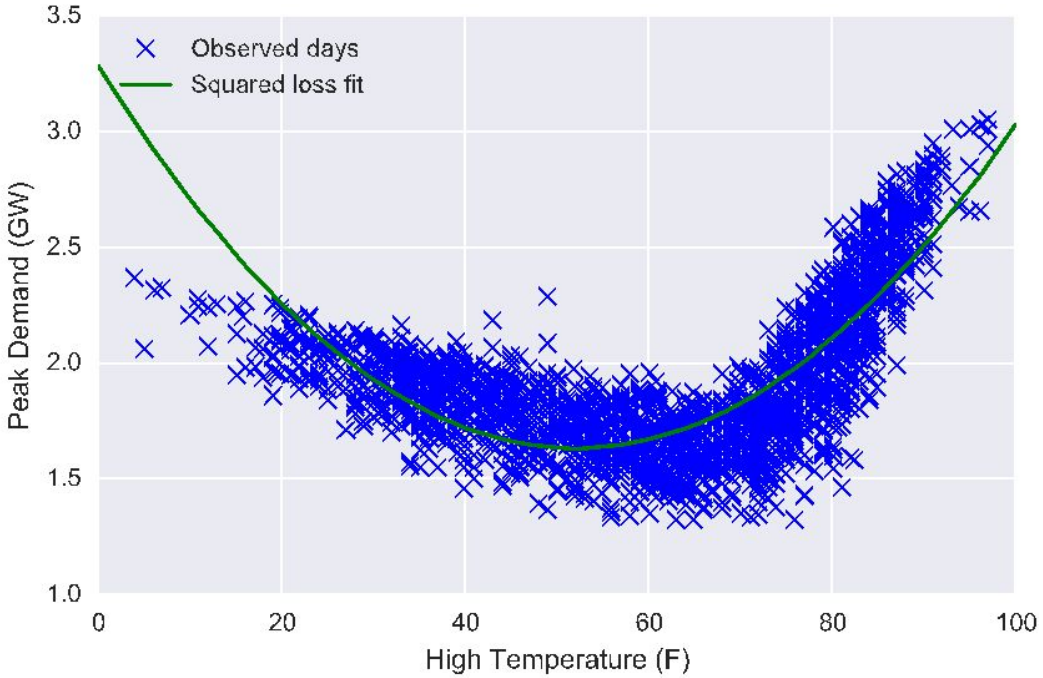
Though it may seem limited, linear regression algorithms are quite powerful when applied to *non-linear features* of the input data, e.g.

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

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

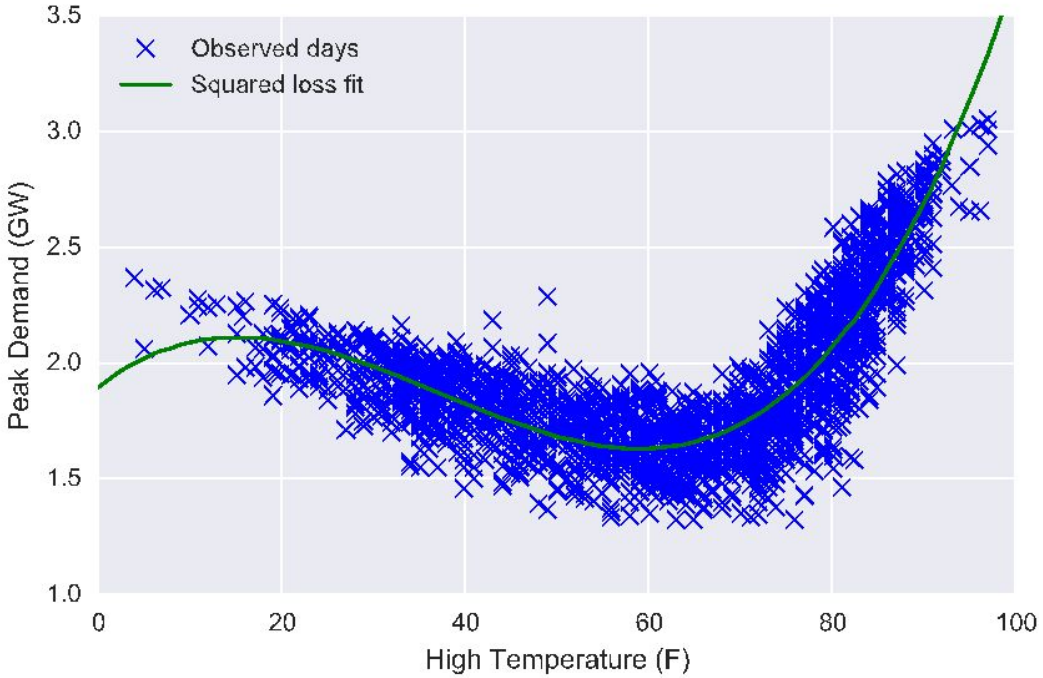
The only element we need to add to write this non-linear regression is the creation of the non-linear features

```
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))
```

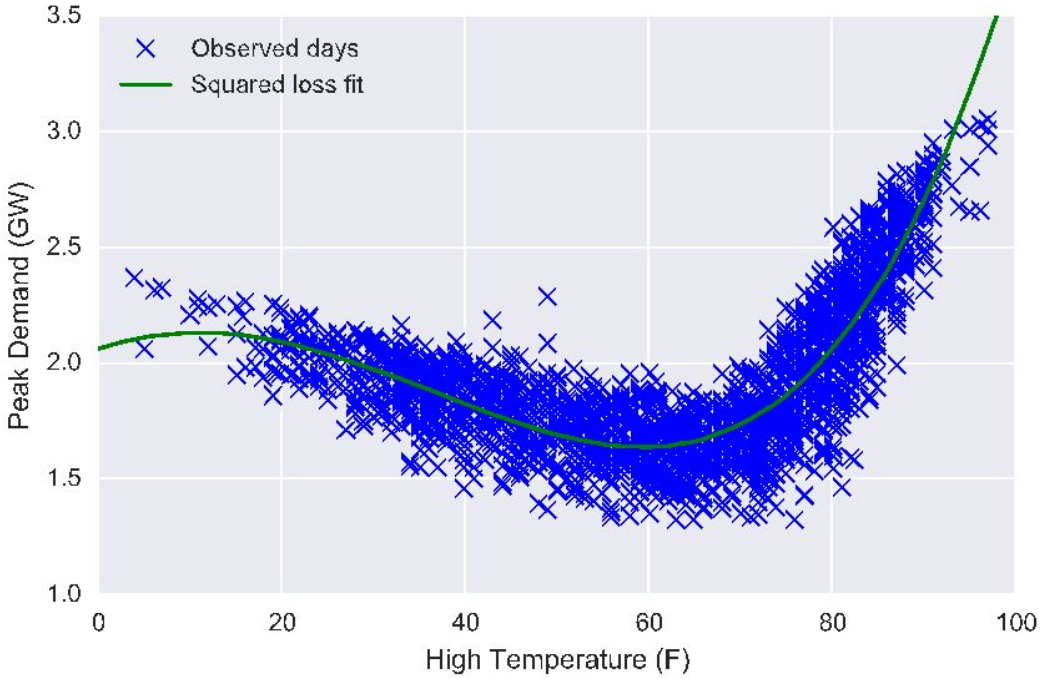
Output learned function:

```
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)
```

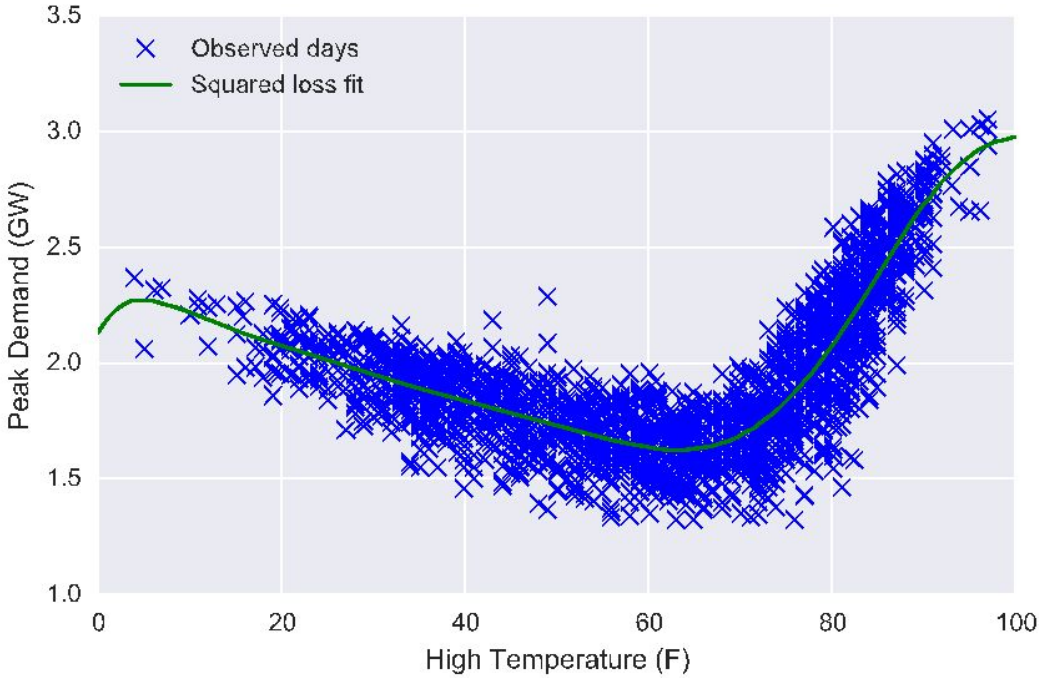
Polynomial features of degree 3



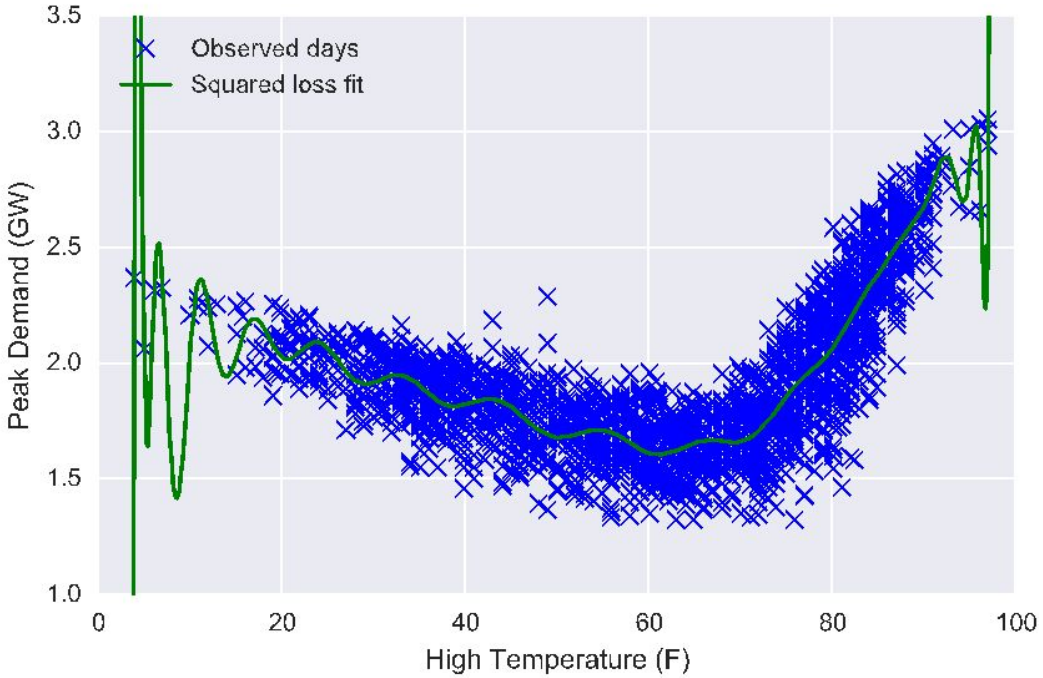
Polynomial features of degree 4



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

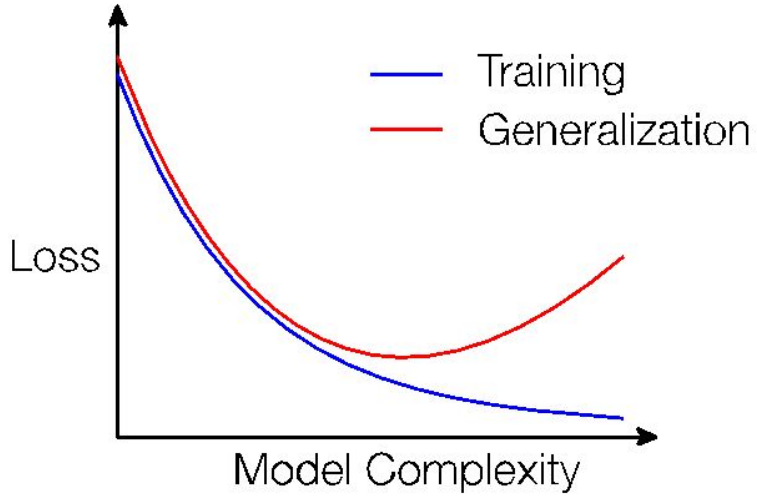
$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

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



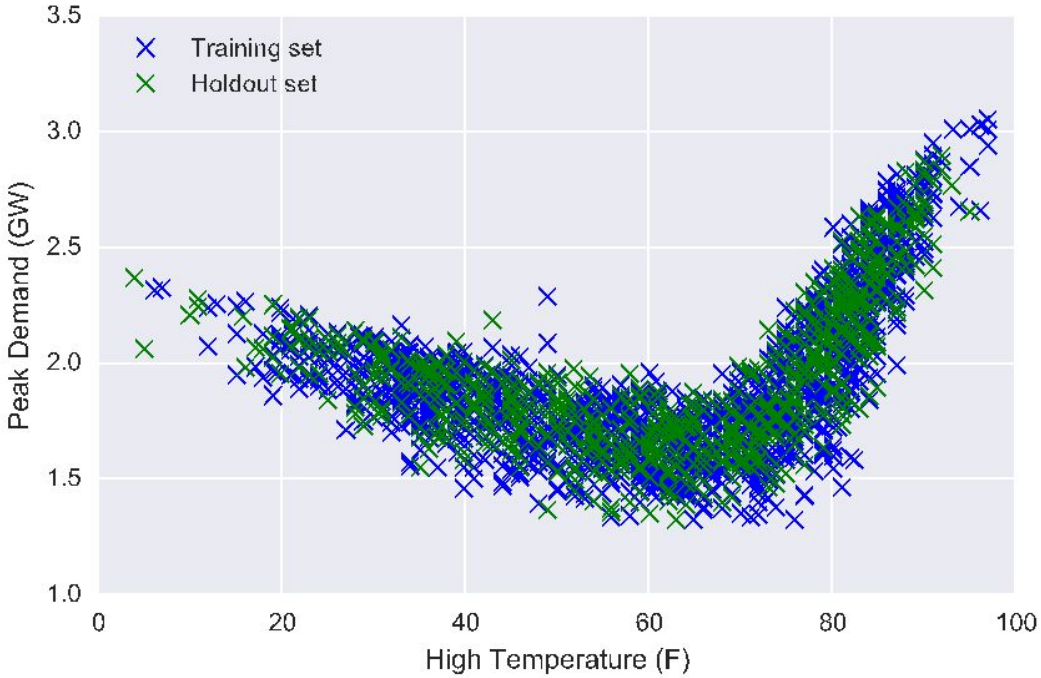
Cross-validation

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***

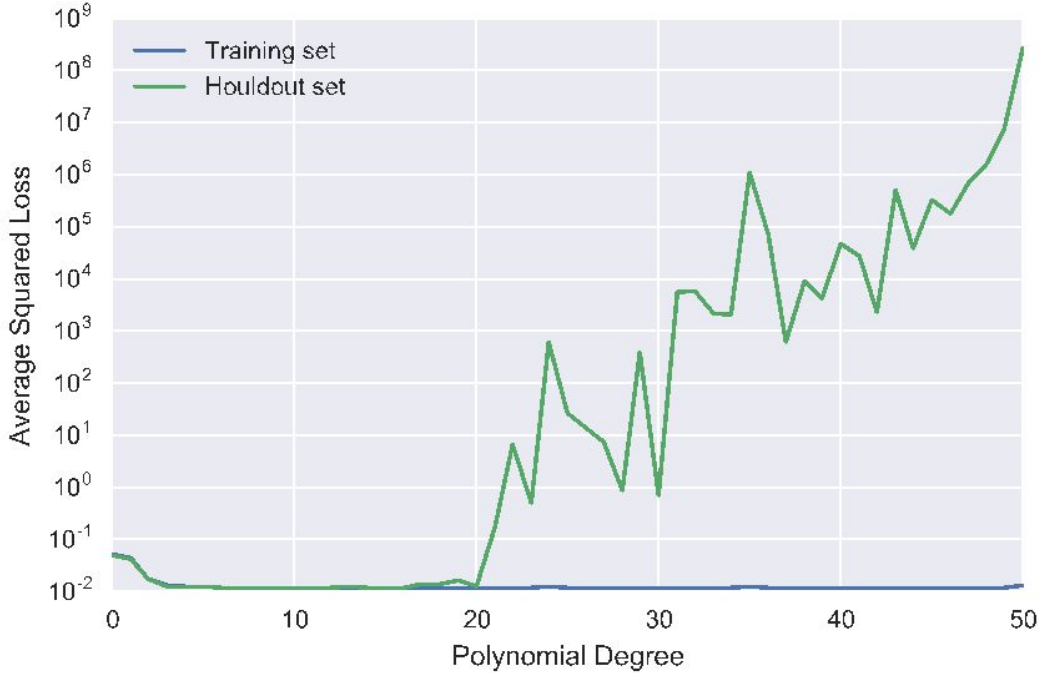
Basic idea is to split the data set into a training set and a holdout set

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

$$\theta = -3.88\times10^{6}, 7.60\times10^{6}, 3.94\times10^{6}, -2.60\times10^{7}, \ldots$$

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:

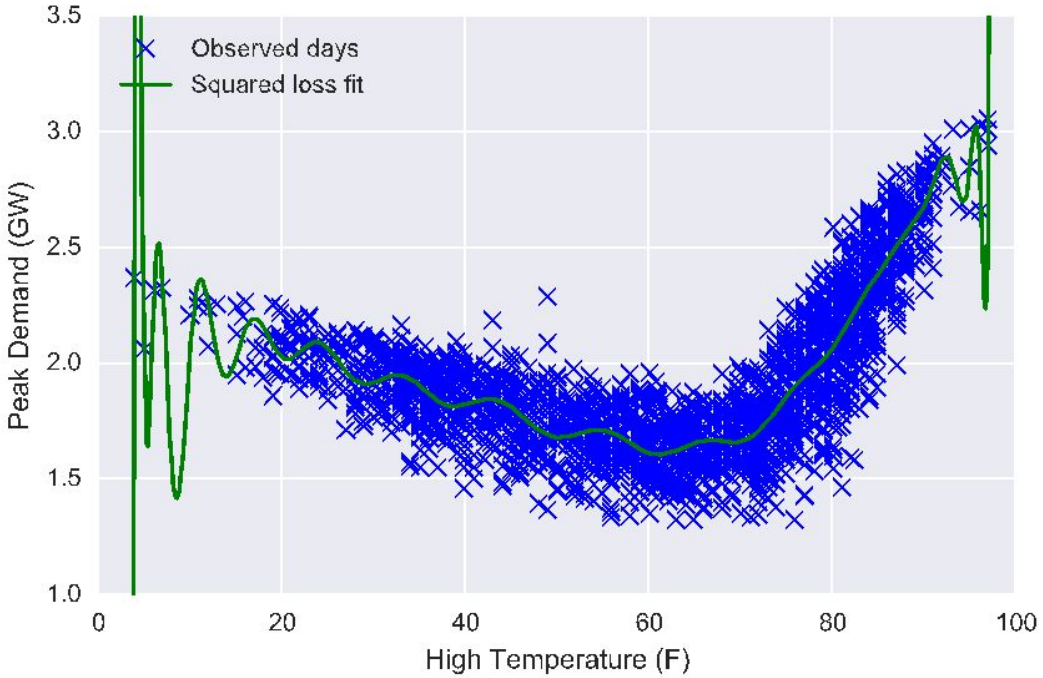
$$\nabla_\theta \left(\frac{1}{2}\|X\theta - y\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2\right) = X^T(X\theta - y) + \lambda\theta$$

Setting gradient equal to zero leads to the solution

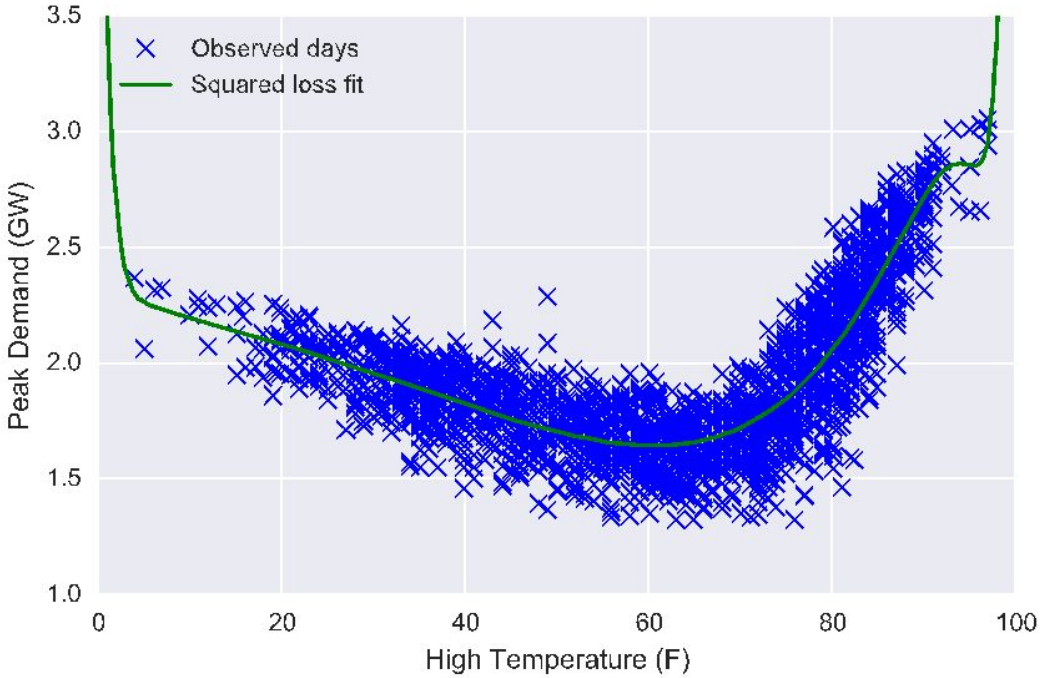
$$X^T X\theta + \lambda\theta = X^T y \implies \theta = (X^T X + \lambda I)^{-1} X^T y$$

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

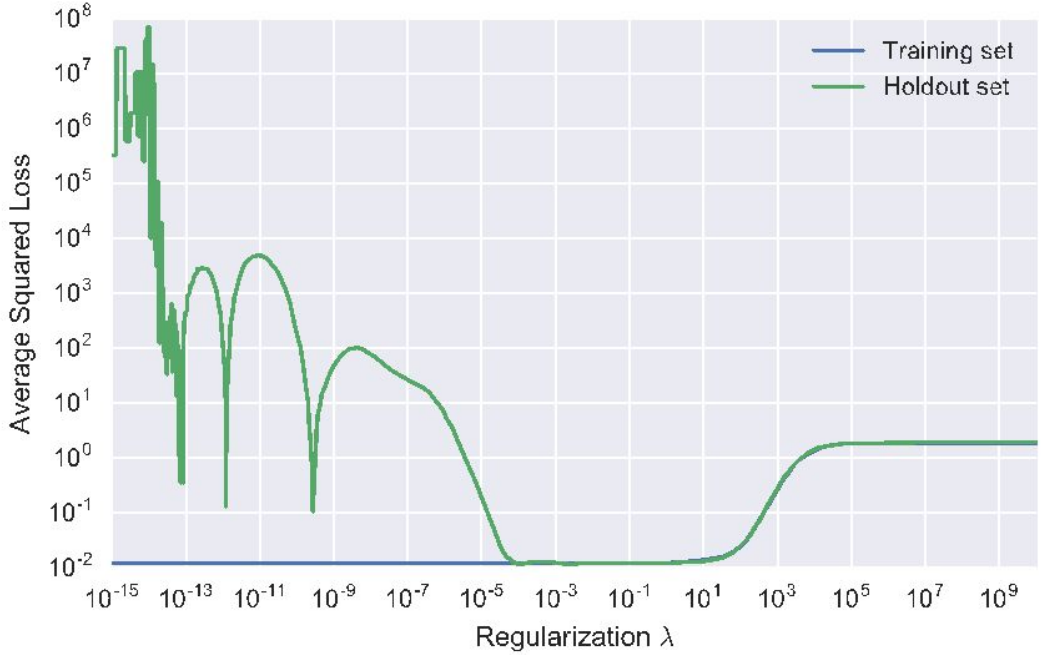
50 degree polynomial fit



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

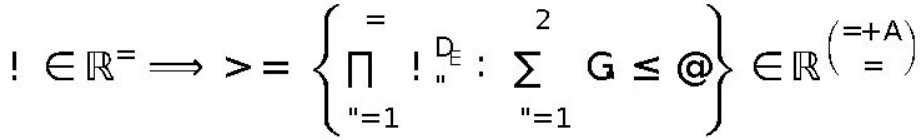
One possibility for higher degree polynomials is to just use an independent polynomial over each dimension (here of degree d)

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

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 @



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:

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

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

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 $\implies \binom{k+d}{k} = O(d^k)$ total features (for fixed d)

RBFs: k-dimensional raw input, uniform grid with d centers over each dimension $\implies d^k$ 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. $\text{Temperature}^2 \cdot \text{Weekday}$, but not $\text{Temperature} \cdot \text{Humidity}$

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)*

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

Nonlinear classification

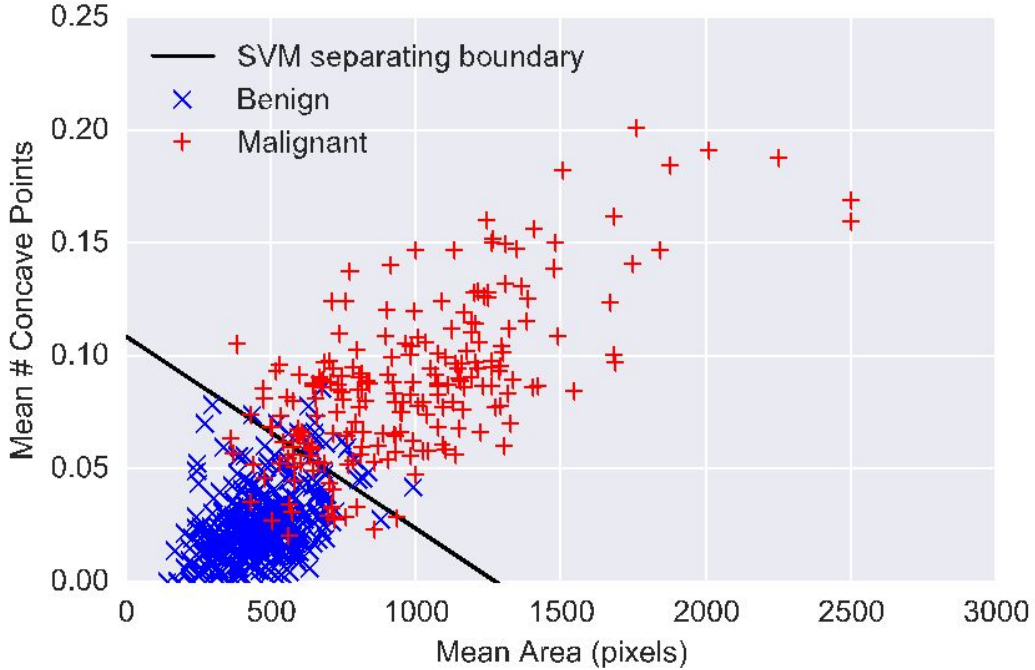
Just like linear regression, the nice thing about using nonlinear features for classification is that our algorithms remain exactly the same as before

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

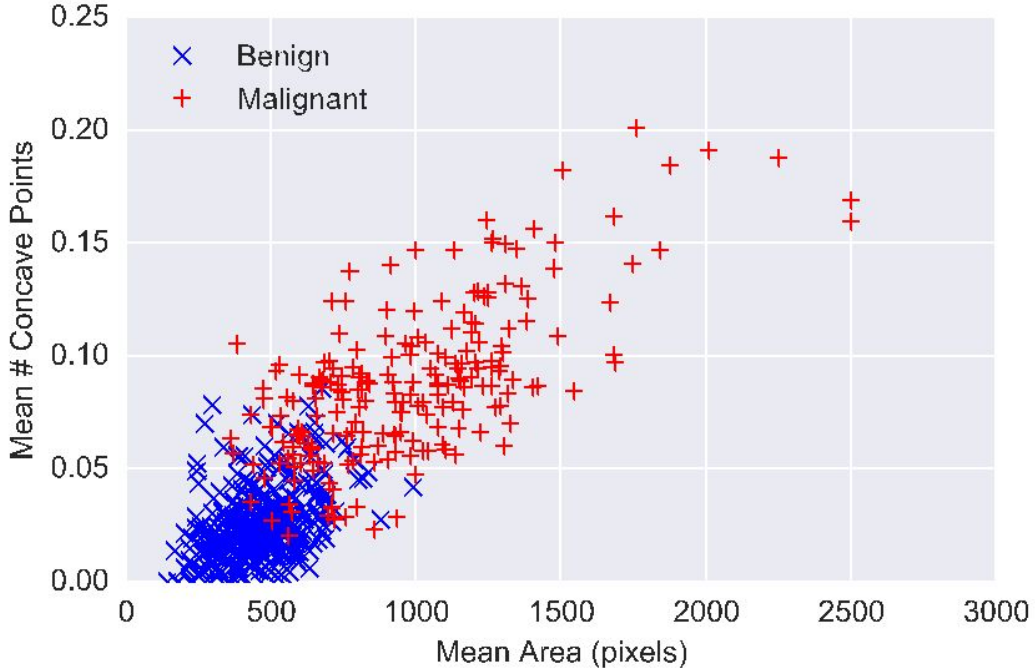
$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\} + \frac{\lambda}{2}\|\theta\|_2^2$$

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

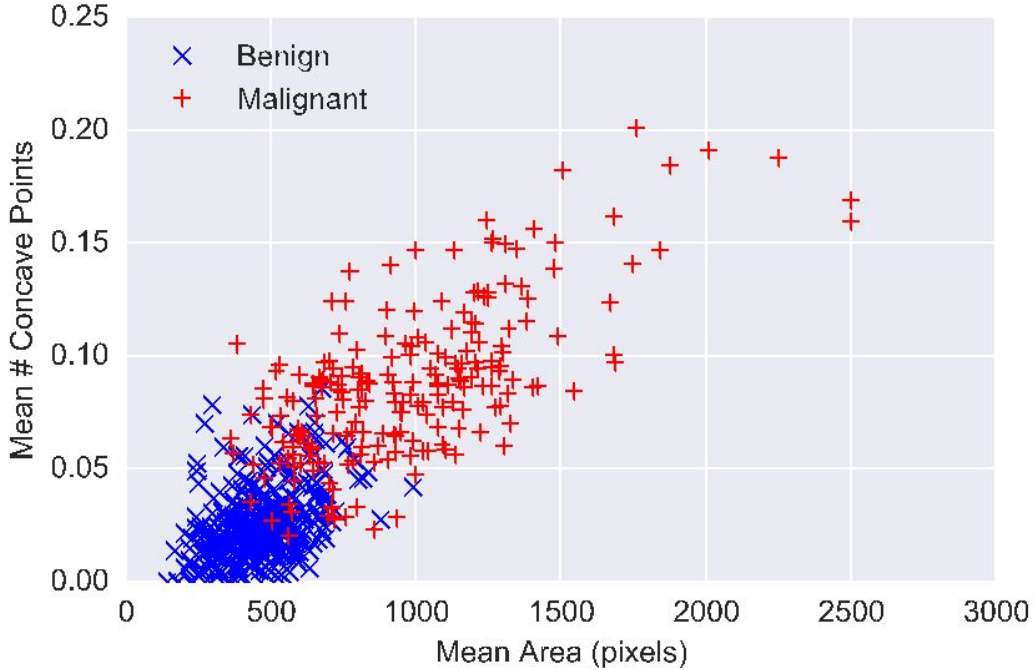
Linear SVM on cancer data set



Polynomial features @= 2



Polynomial features @= 3



Polynomial features @= 10