INTRODUCTION TO DATA SCIENCE

JOHN P DICKERSON

Lecture #23 – 11/9/2021
Lecture #24 – 11/11/2021

CMSC320
Tuesdays & Thursdays
5:00pm – 6:15pm
(… or anytime on the Internet)

COMPUTER SCIENCE
UNIVERSITY OF MARYLAND


QUICK RECAP FROM
LAST CLASS ...

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 Z_2 Z_B

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

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

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

T_1 T_2 T_j T_B

Aggregate/Vote

Class estimate or predicted value

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

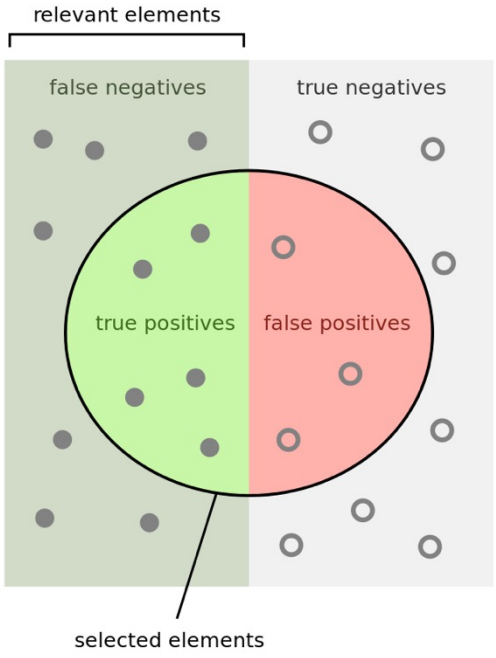
QUICK ASIDE FOR PROJECT #3

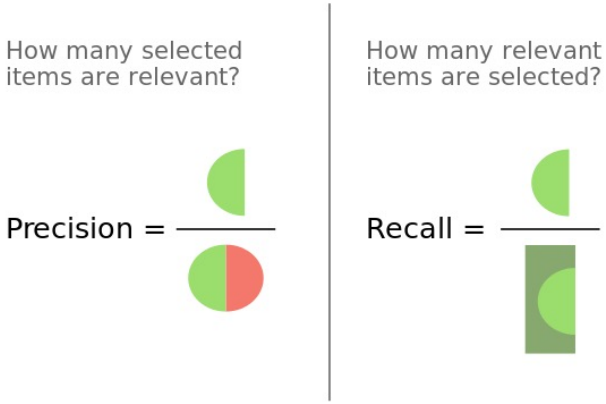
Precision P:

#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

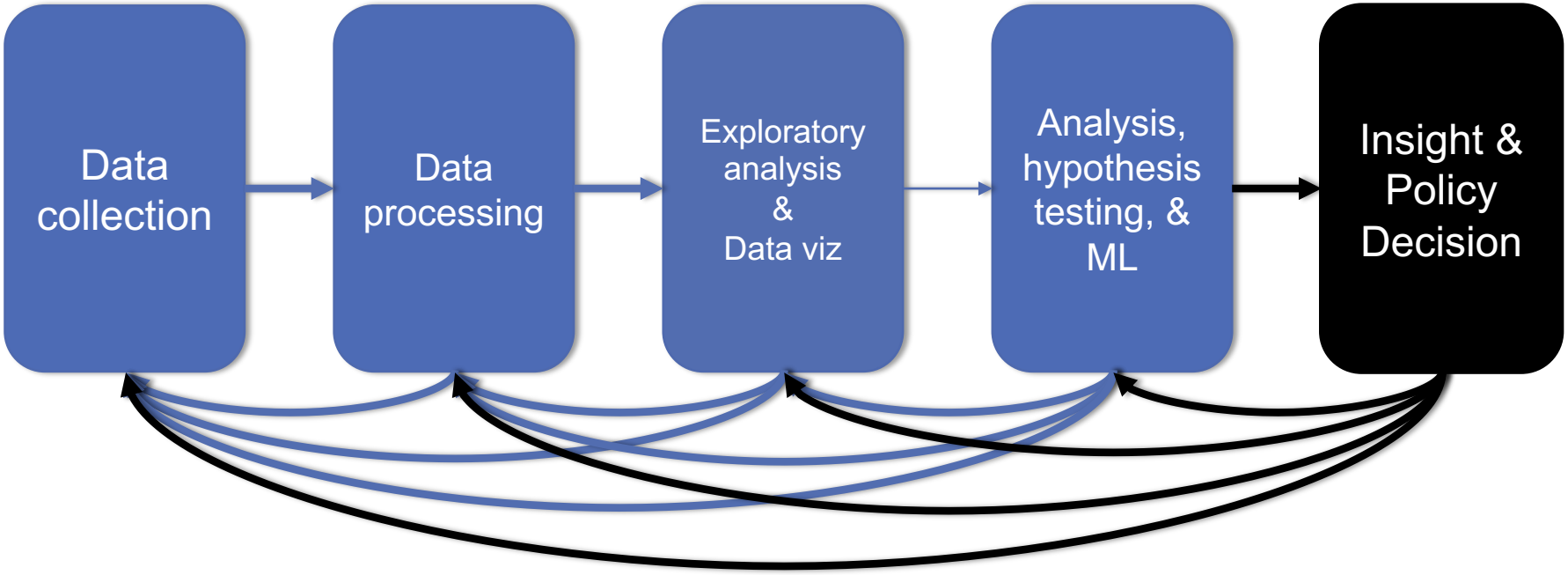
F_1: (harmonic) mean of precision and recall:

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

Can be parameterized to attach higher importance to recall:

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

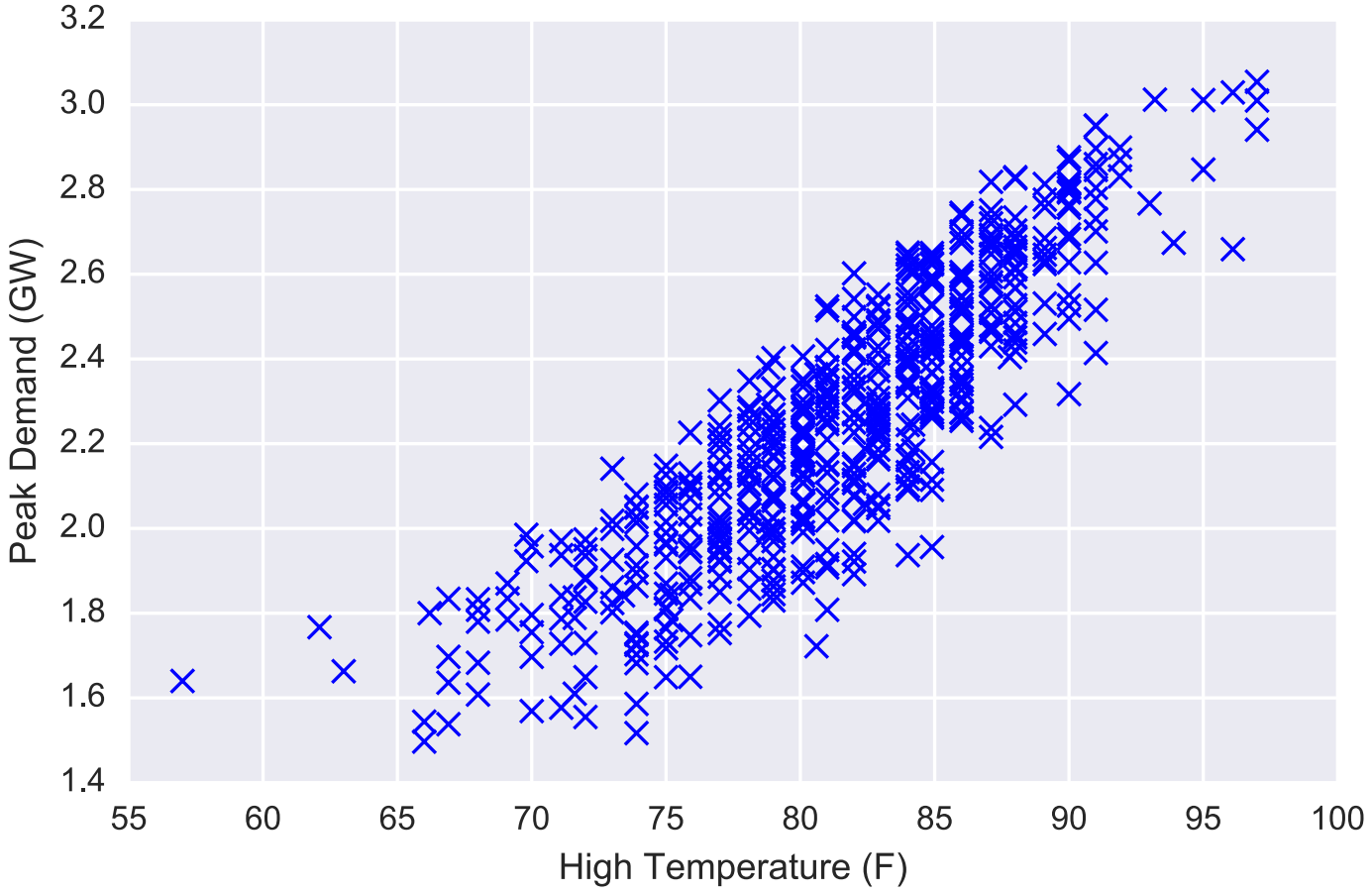
TODAY'S LECTURE



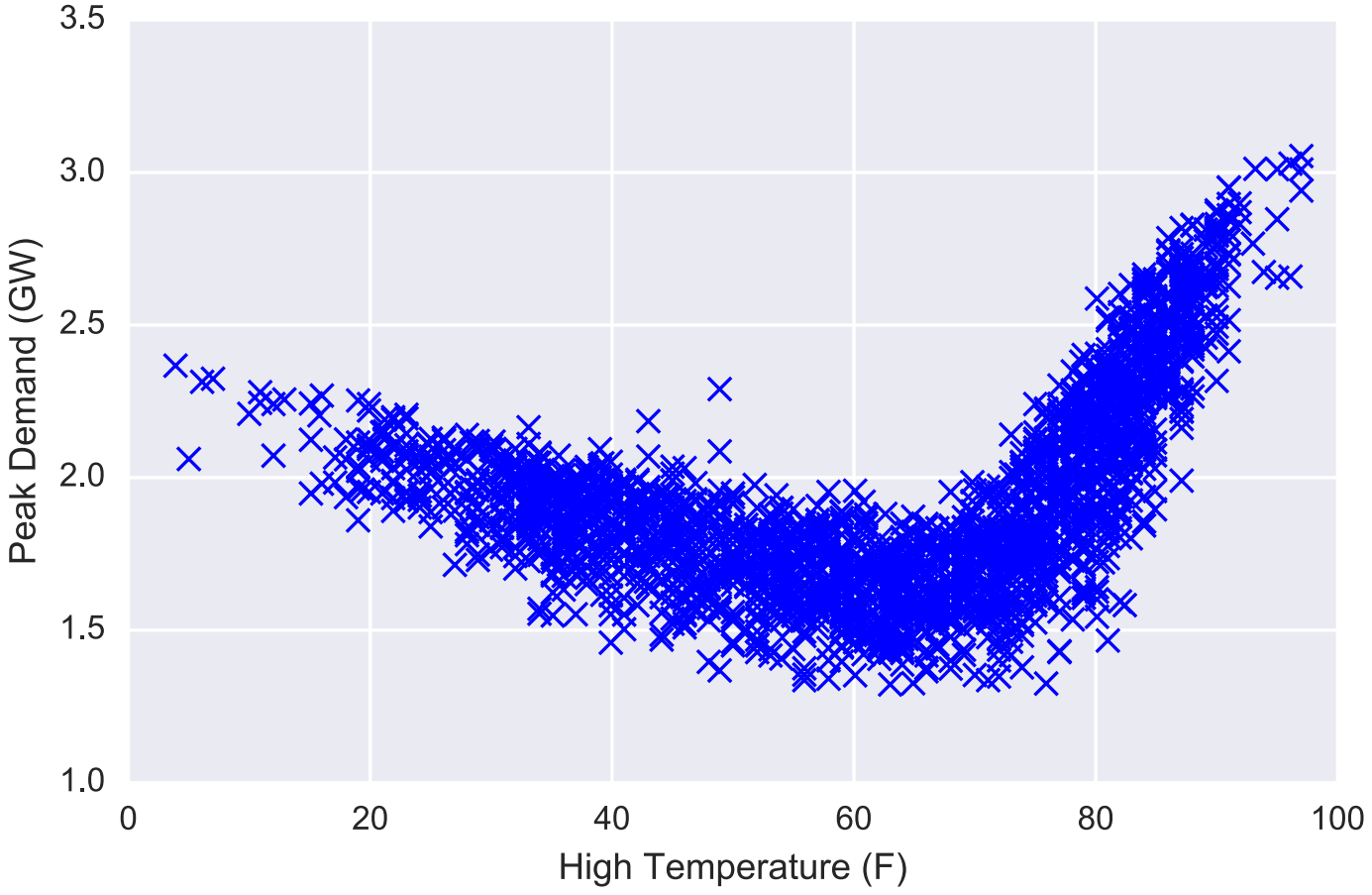
FILLING IN THE GAPS: NONLINEAR REGRESSION & REGULARIZATION

Thanks: Zico Kolter

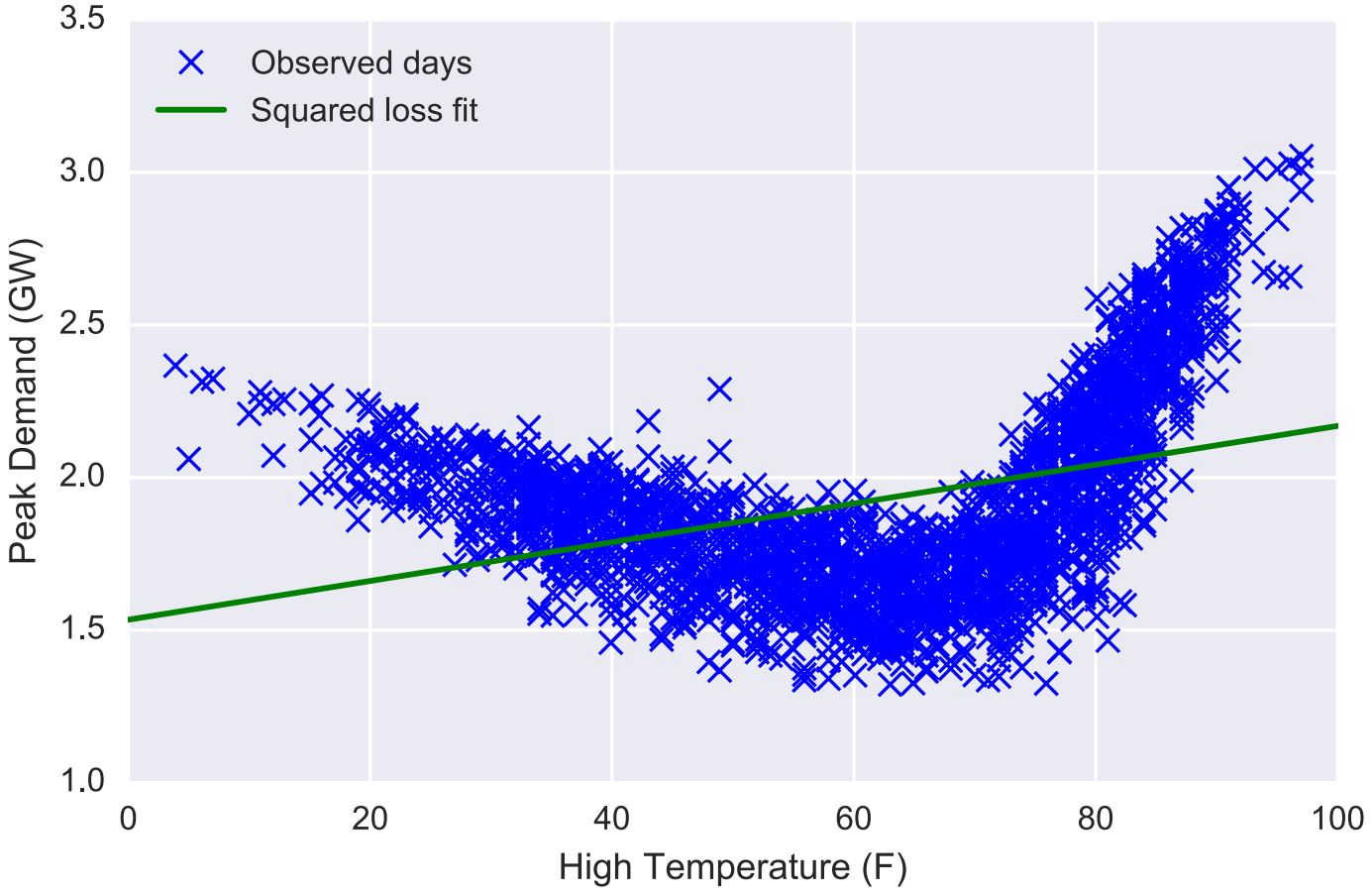
Peak demand vs. temperature (summer months)



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

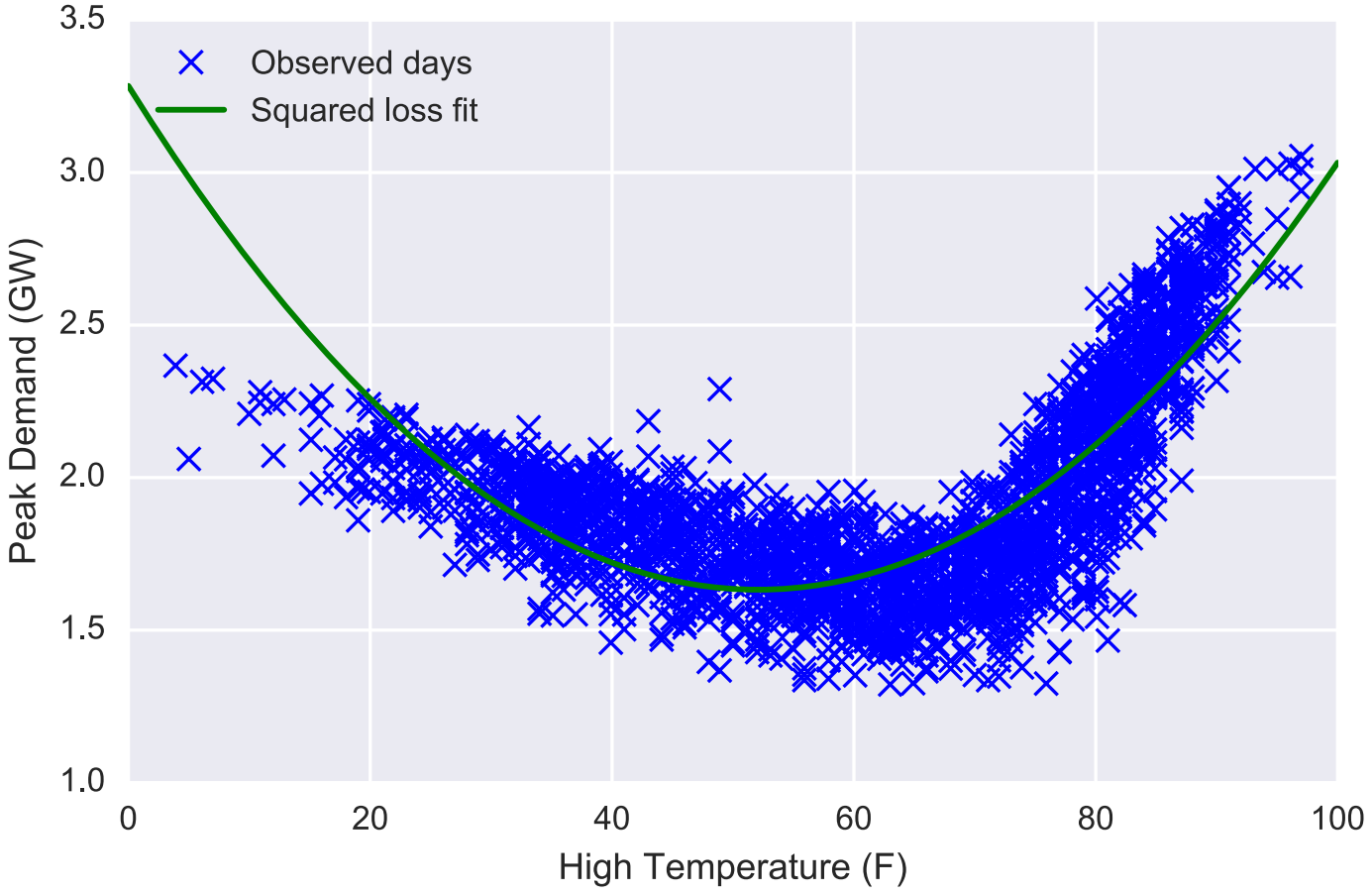
Though it may seem limited, linear regression algorithms are quite powerful when applied to *non-linear features* of the input data, e.g.

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

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

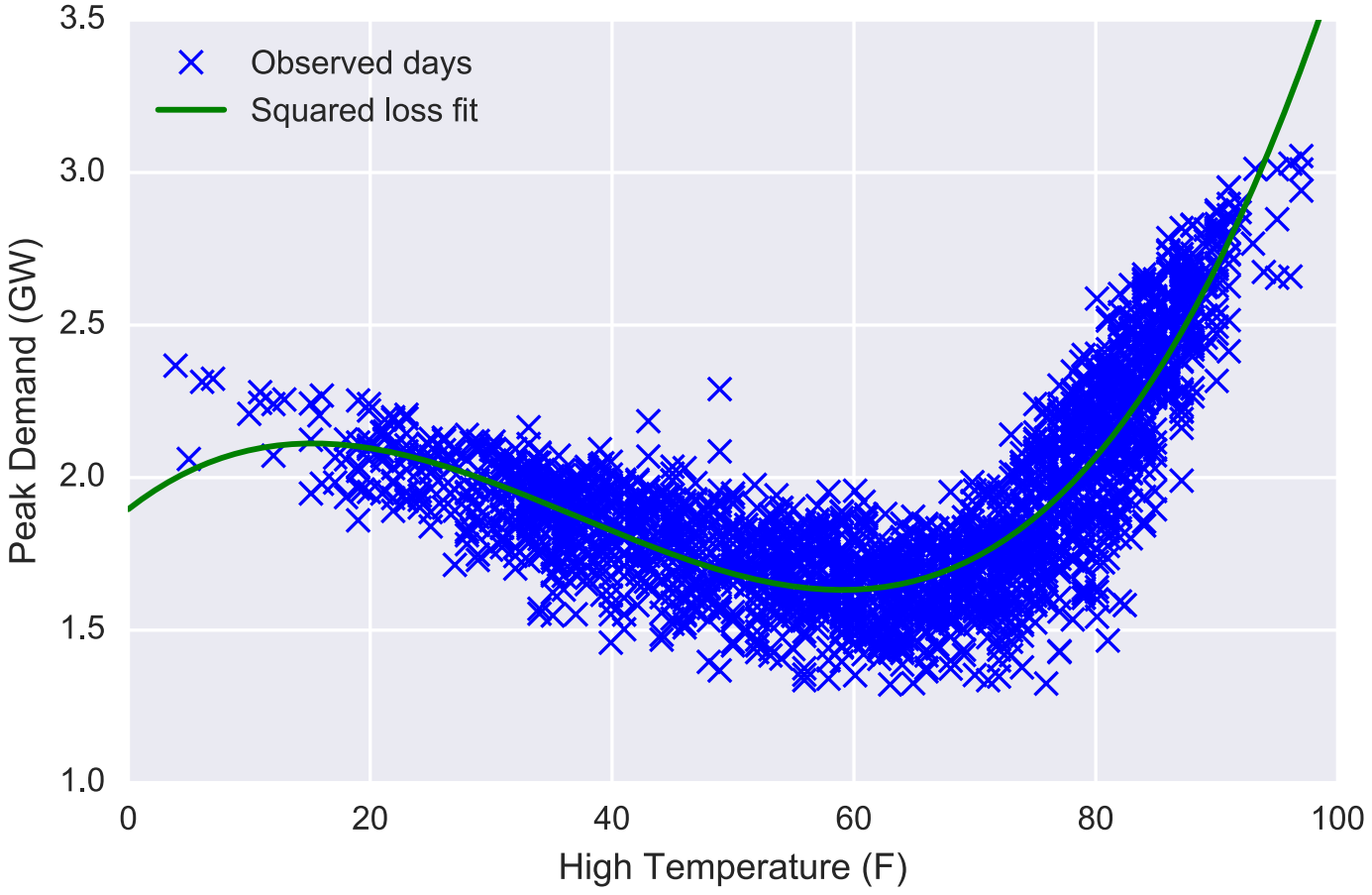
```
x = df_daily.loc[:,"Temperature"]
min_x, rng_x = (np.min(x), np.max(x) - np.min(x))
x = 2*(x - min_x)/rng_x - 1.0
y = df_daily.loc[:,"Load"]

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))
```

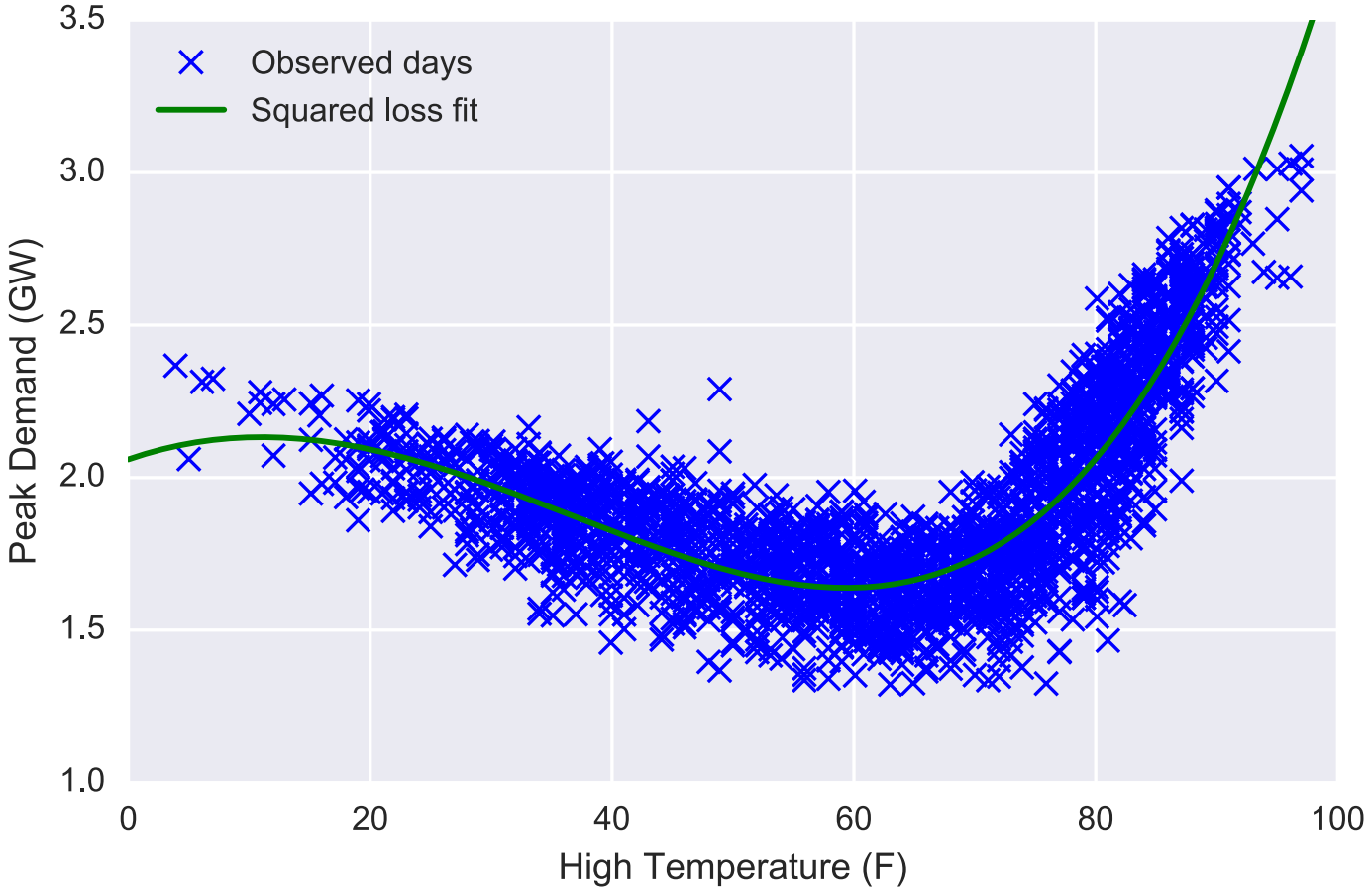
Output learned function:

```
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)
```

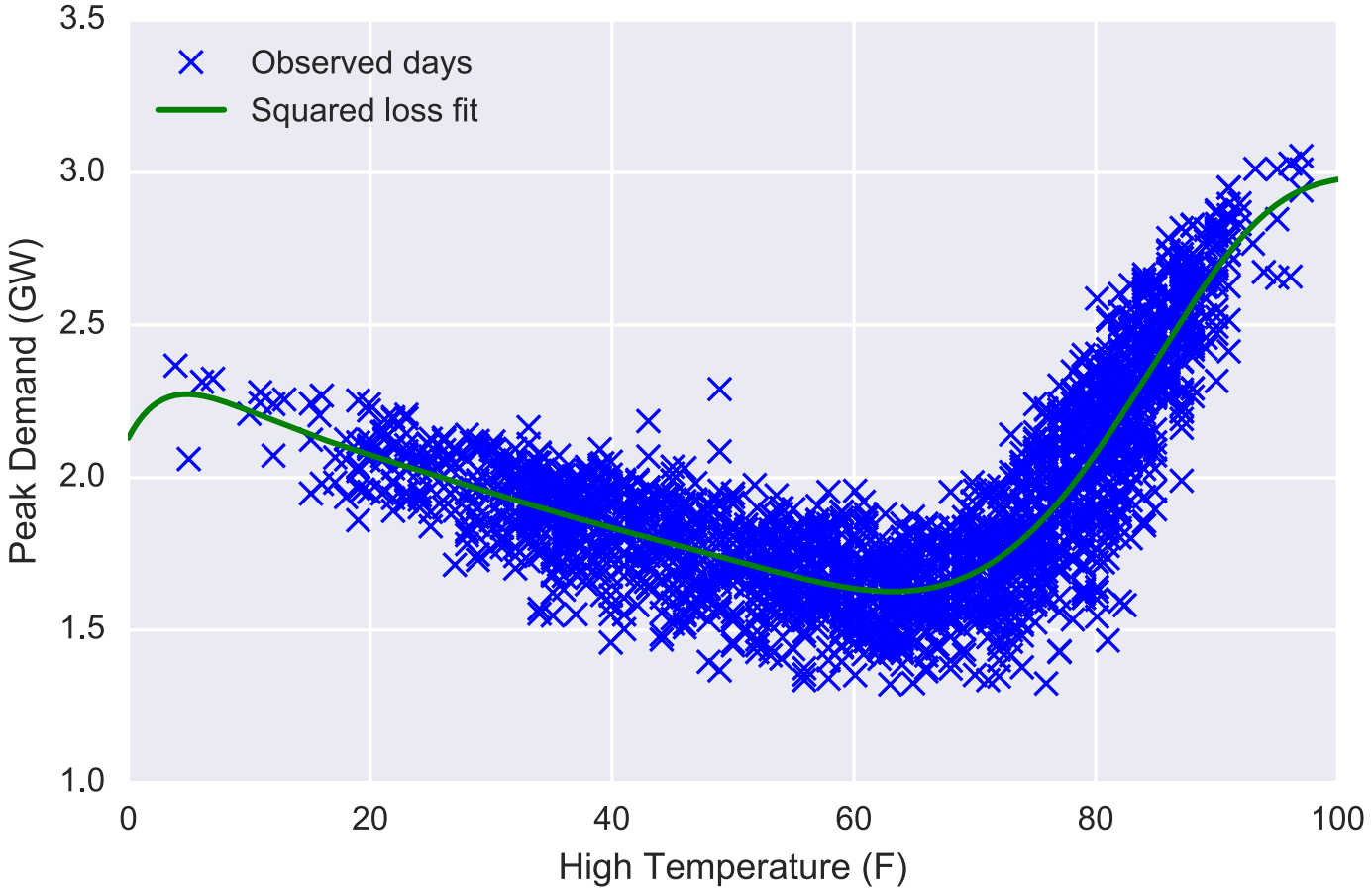
Polynomial features of degree 3



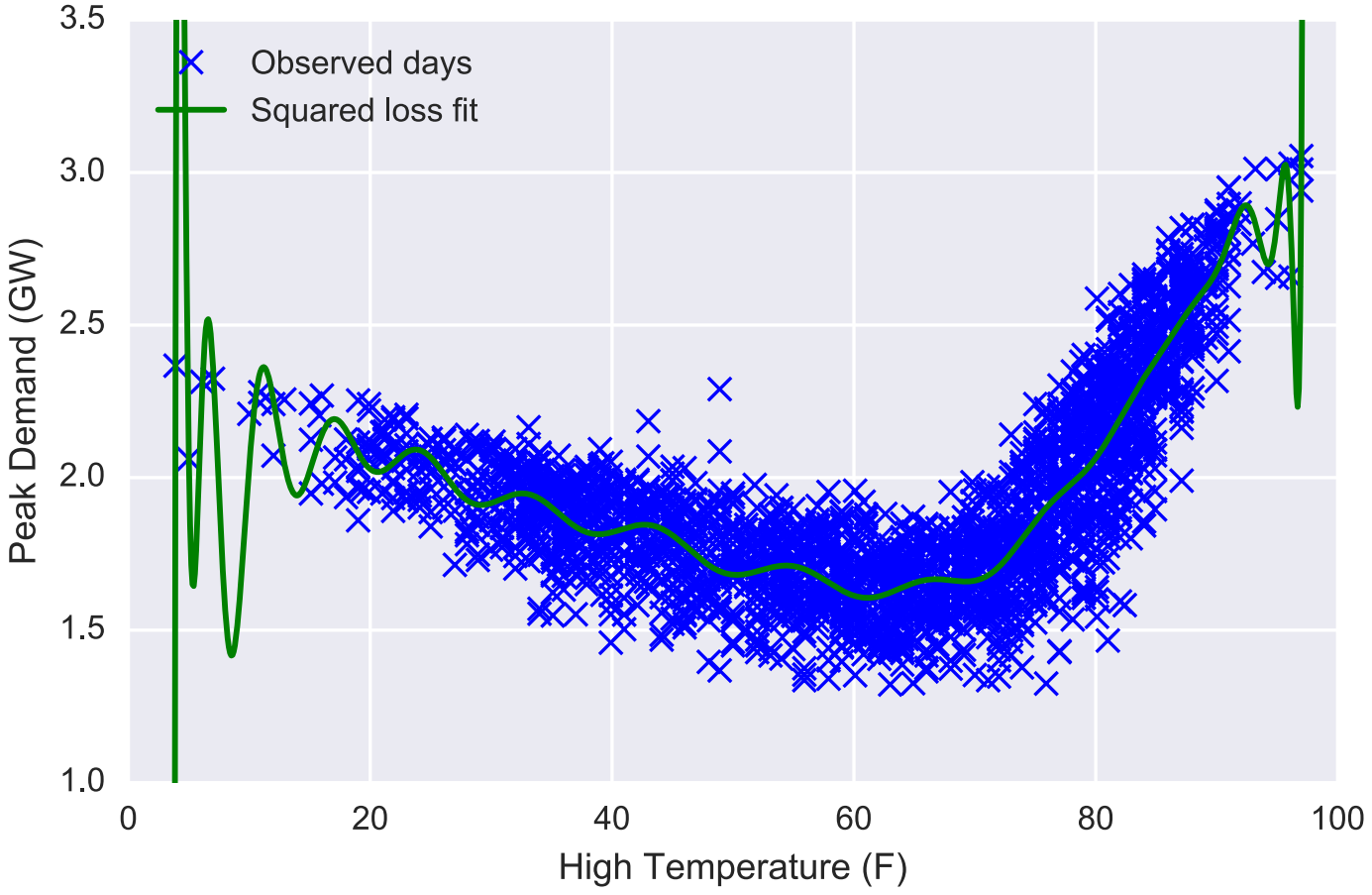
Polynomial features of degree 4



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

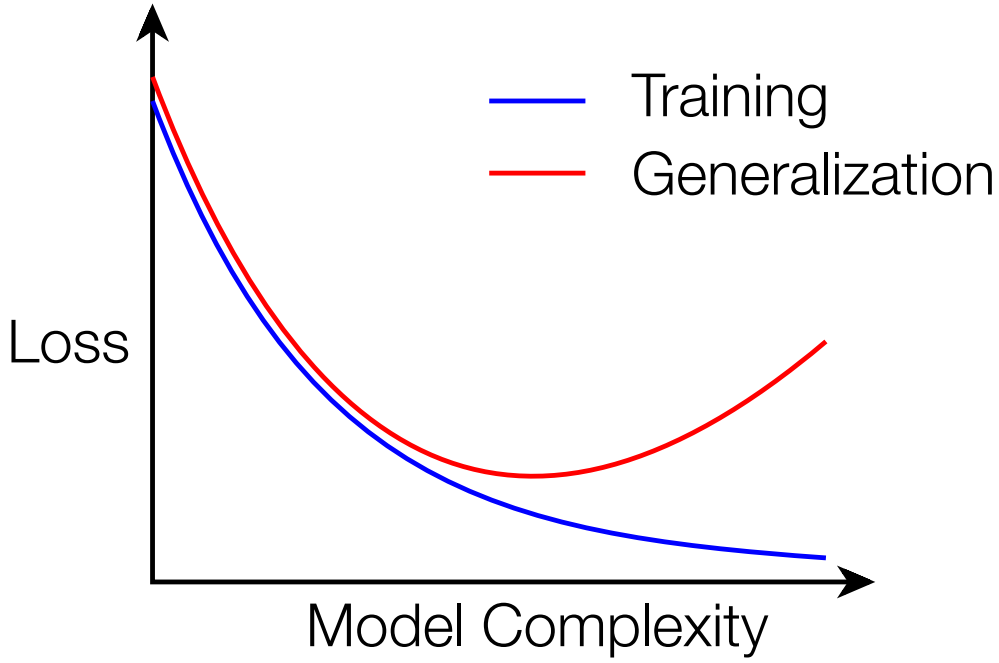
$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \ell\left(h_\theta\left(x^{(i)}\right), y^{(i)}\right)$$

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

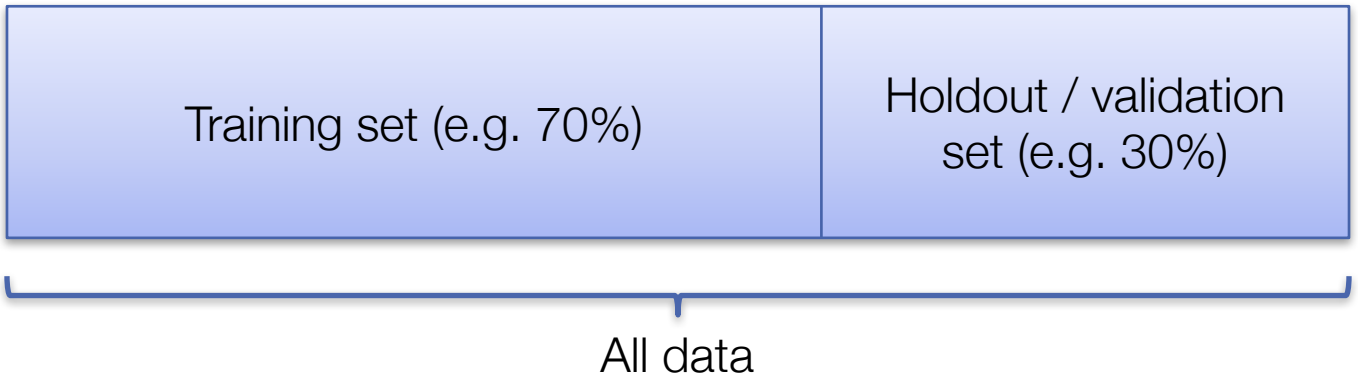
As model becomes more complex, training loss always decreases; generalization loss decreases to a point, then starts to increase



Cross-validation

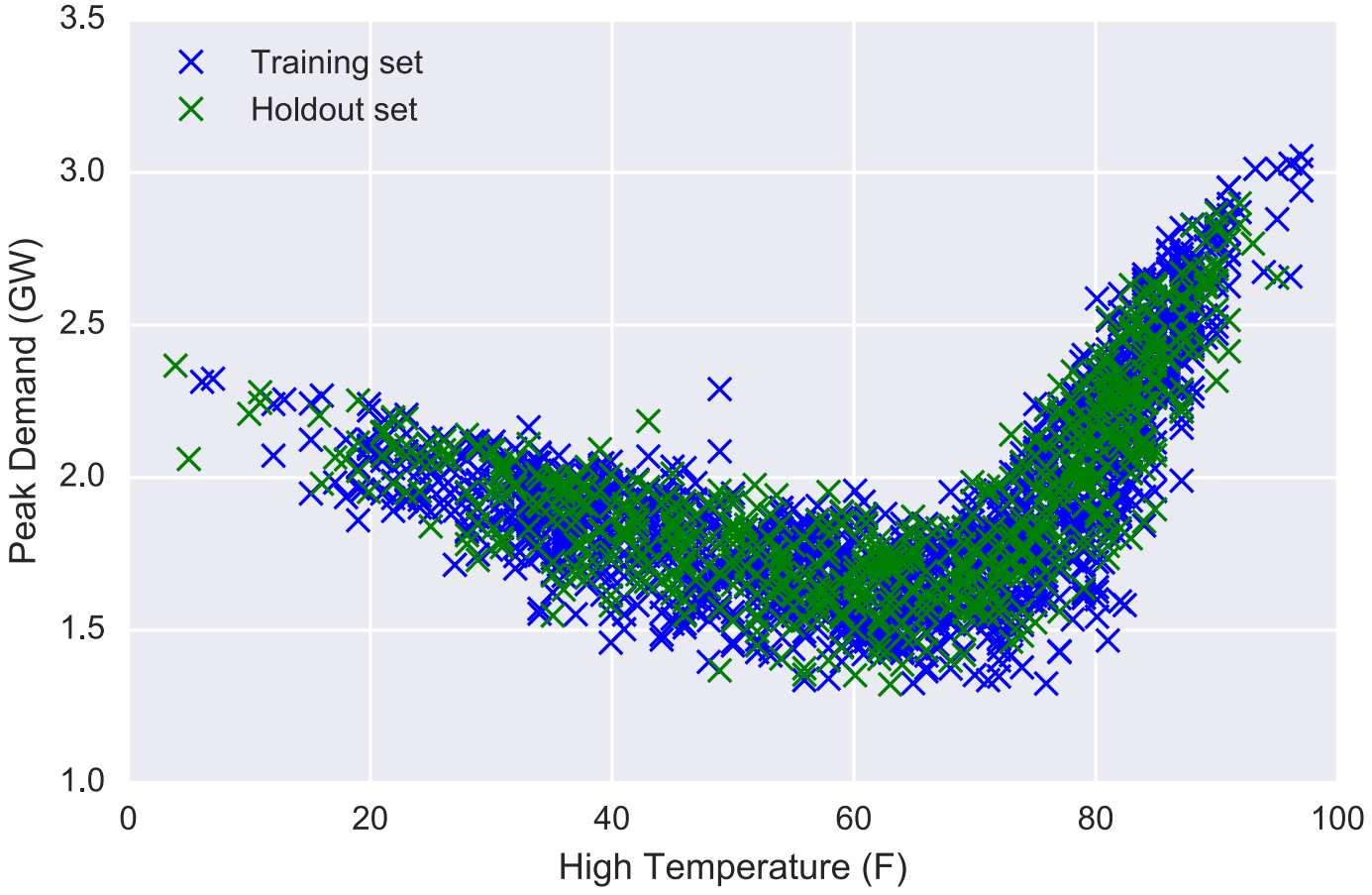
Although it is difficult to quantify the true generalization error (i.e., the error of these algorithms over the *complete* distribution of possible examples), we can approximate it by ***holdout cross-validation***

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

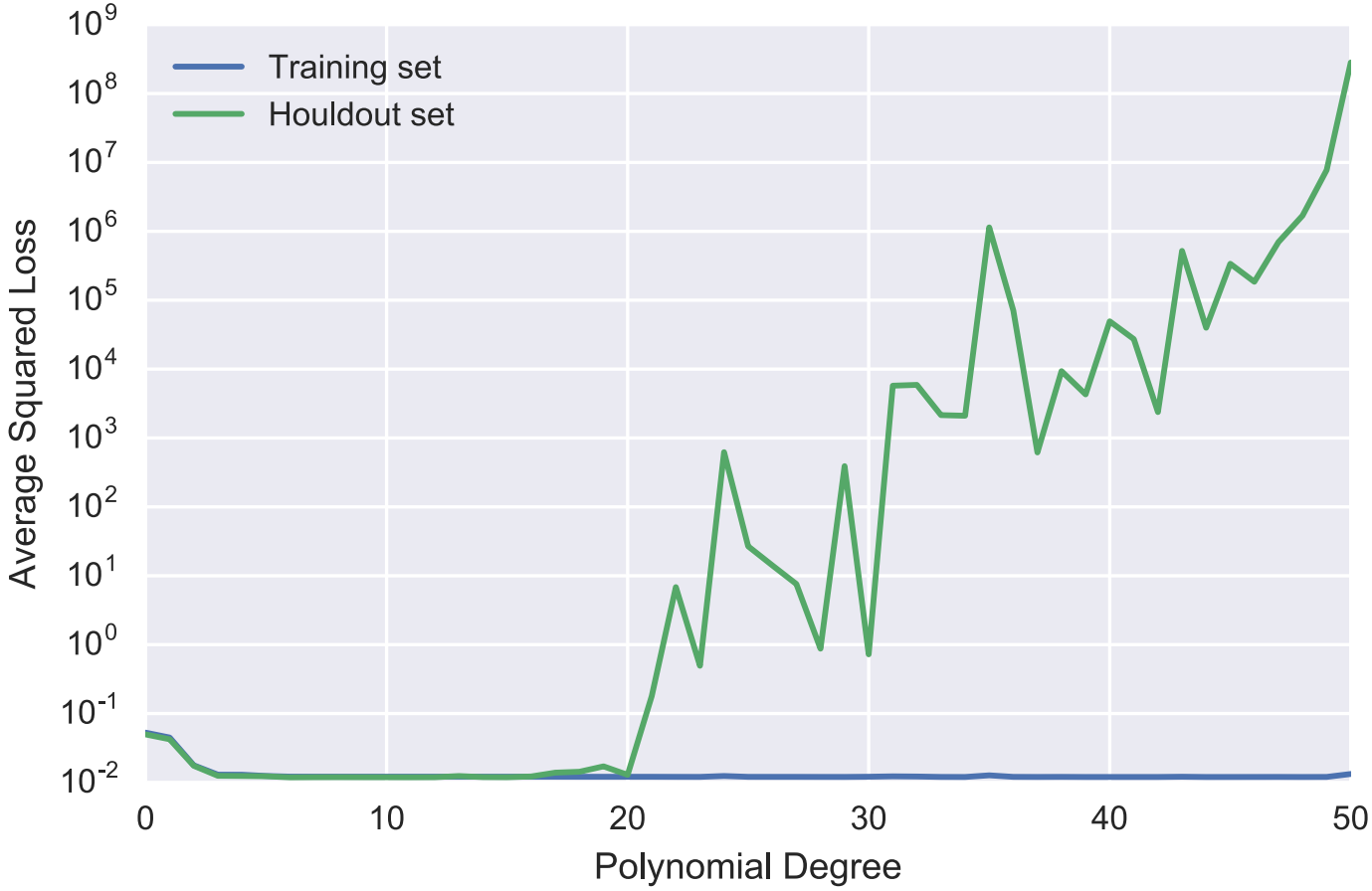


Train the algorithm on the training set and evaluate on the holdout set

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:

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

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:

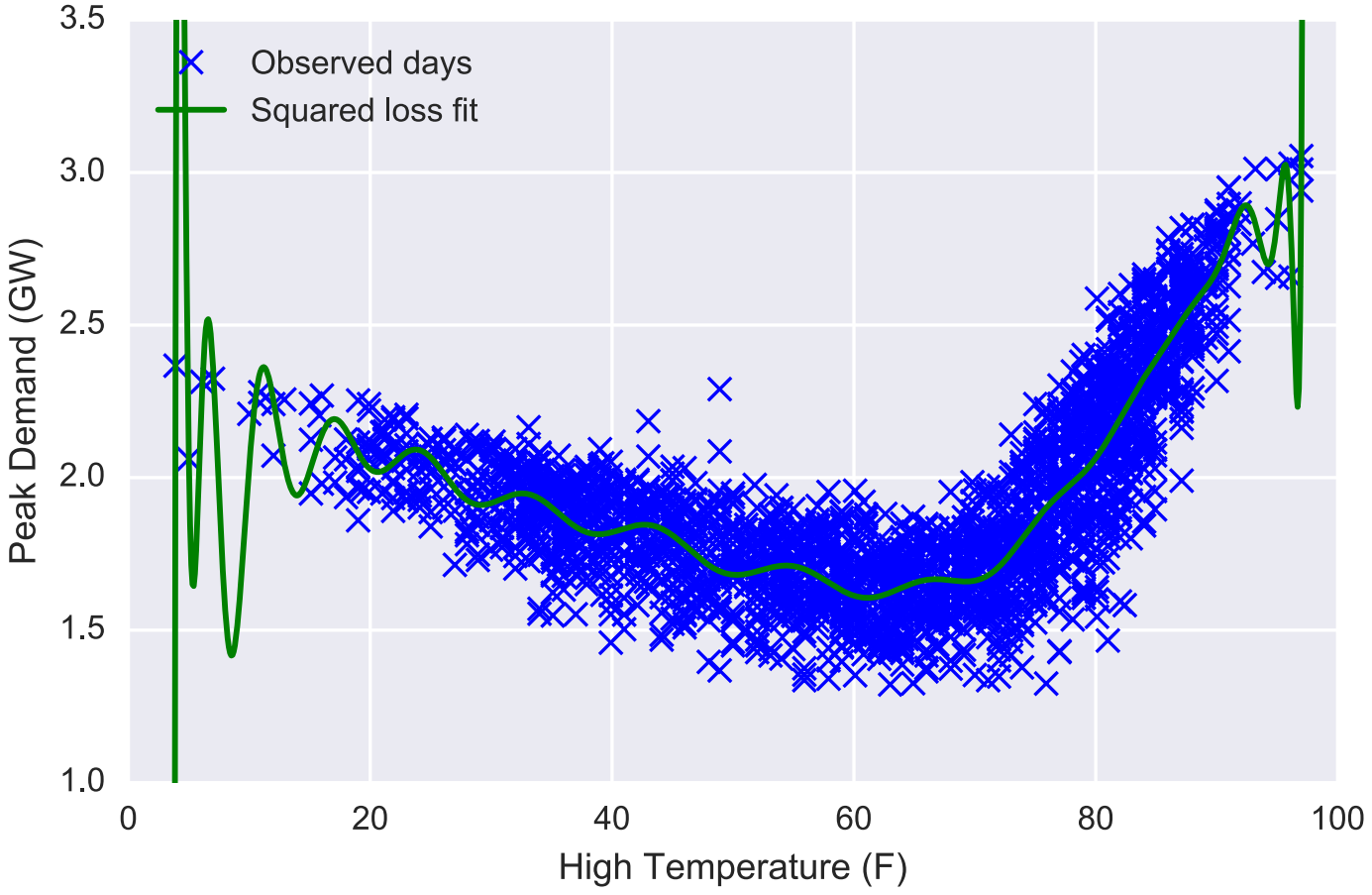
$$\nabla_\theta \left(\frac{1}{2}\|X\theta - y\|_2^2 + \frac{\lambda}{2}\|\theta\|_2^2\right) = X^T(X\theta - y) + \lambda\theta$$

Setting gradient equal to zero leads to the solution

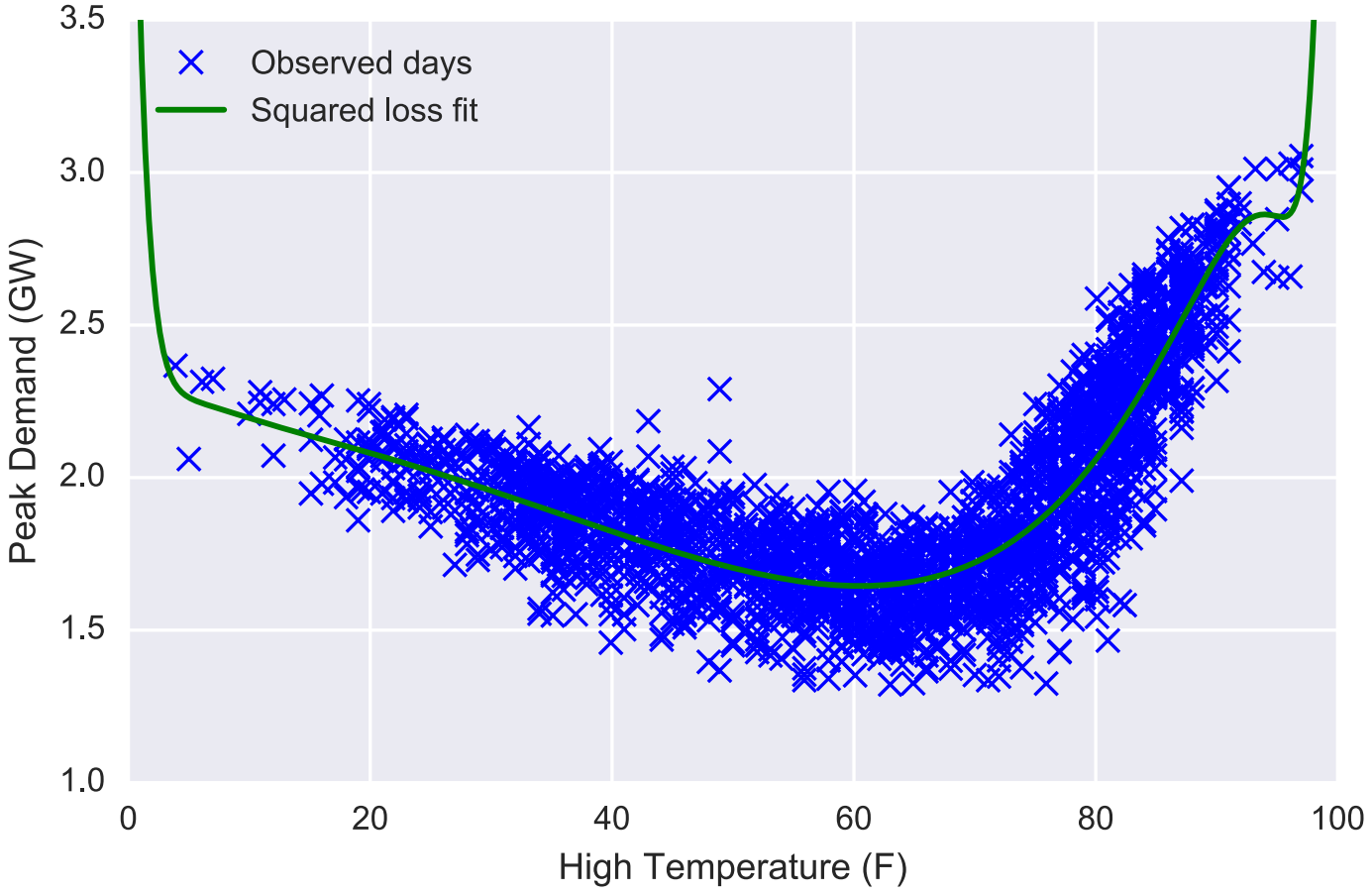
$$X^T X\theta + \lambda\theta = X^T y \implies \theta = (X^T X + \lambda I)^{-1} X^T y$$

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

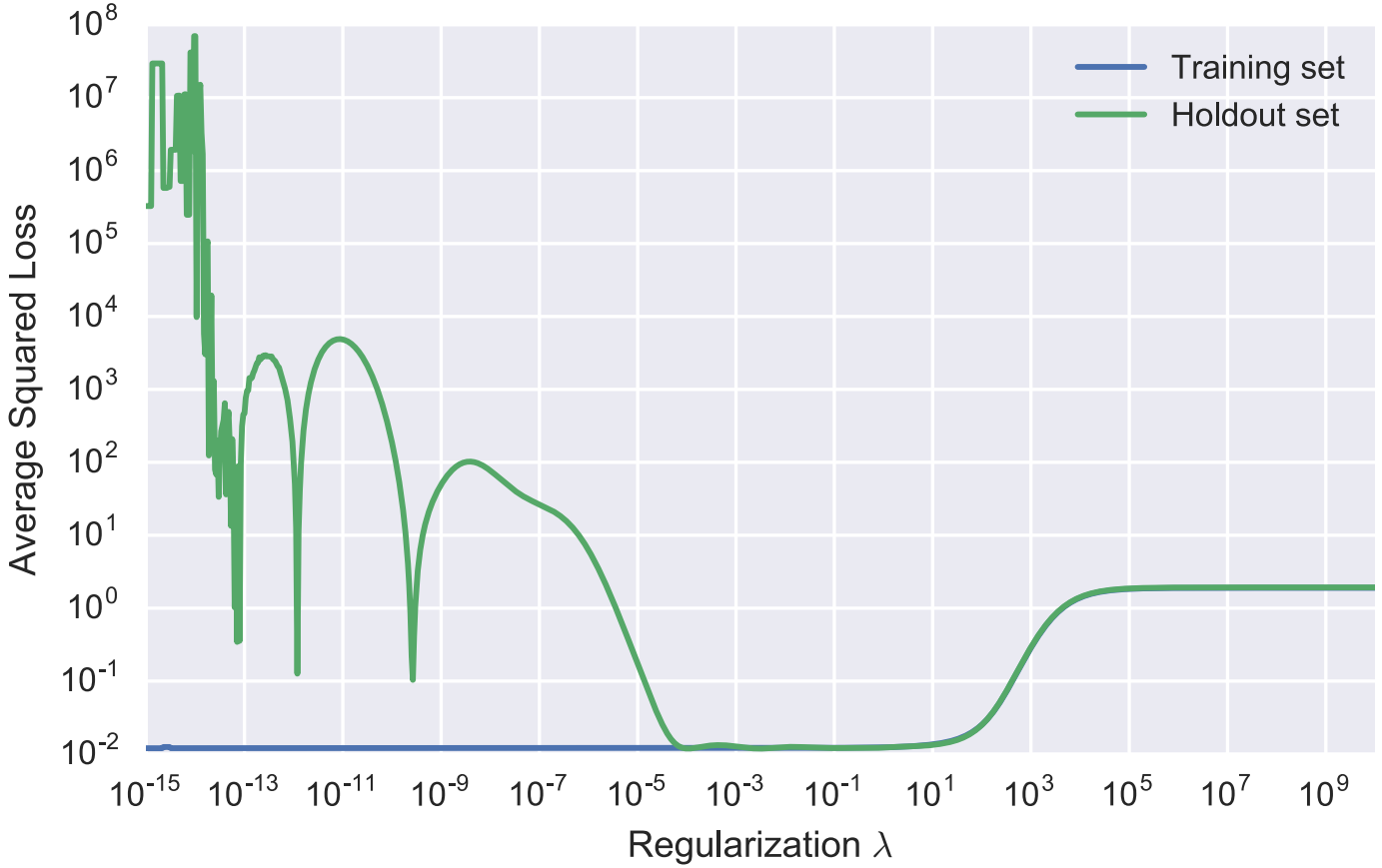
50 degree polynomial fit



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)

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

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

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

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)]
```

`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 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:

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

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

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 $X \in \mathbb{R}^{m \times k}$ 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 $\implies \binom{k+d}{k} = O(d^k)$ 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. $\mathrm{Temperature}^2 \cdot \mathrm{Weekday}$, 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)*

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

Nonlinear classification

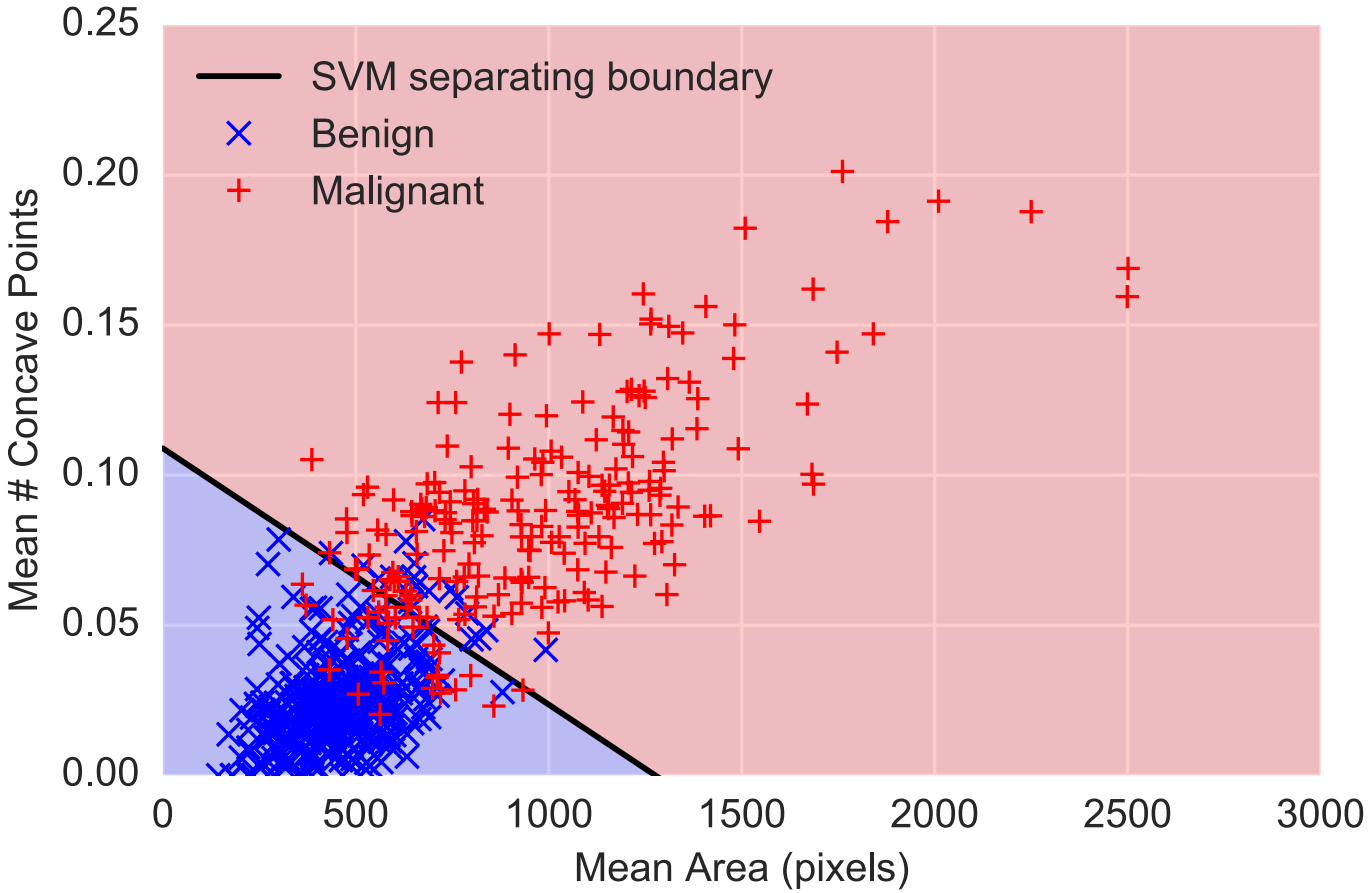
Just like linear regression, the nice thing about using nonlinear features for classification is that our algorithms remain exactly the same as before

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

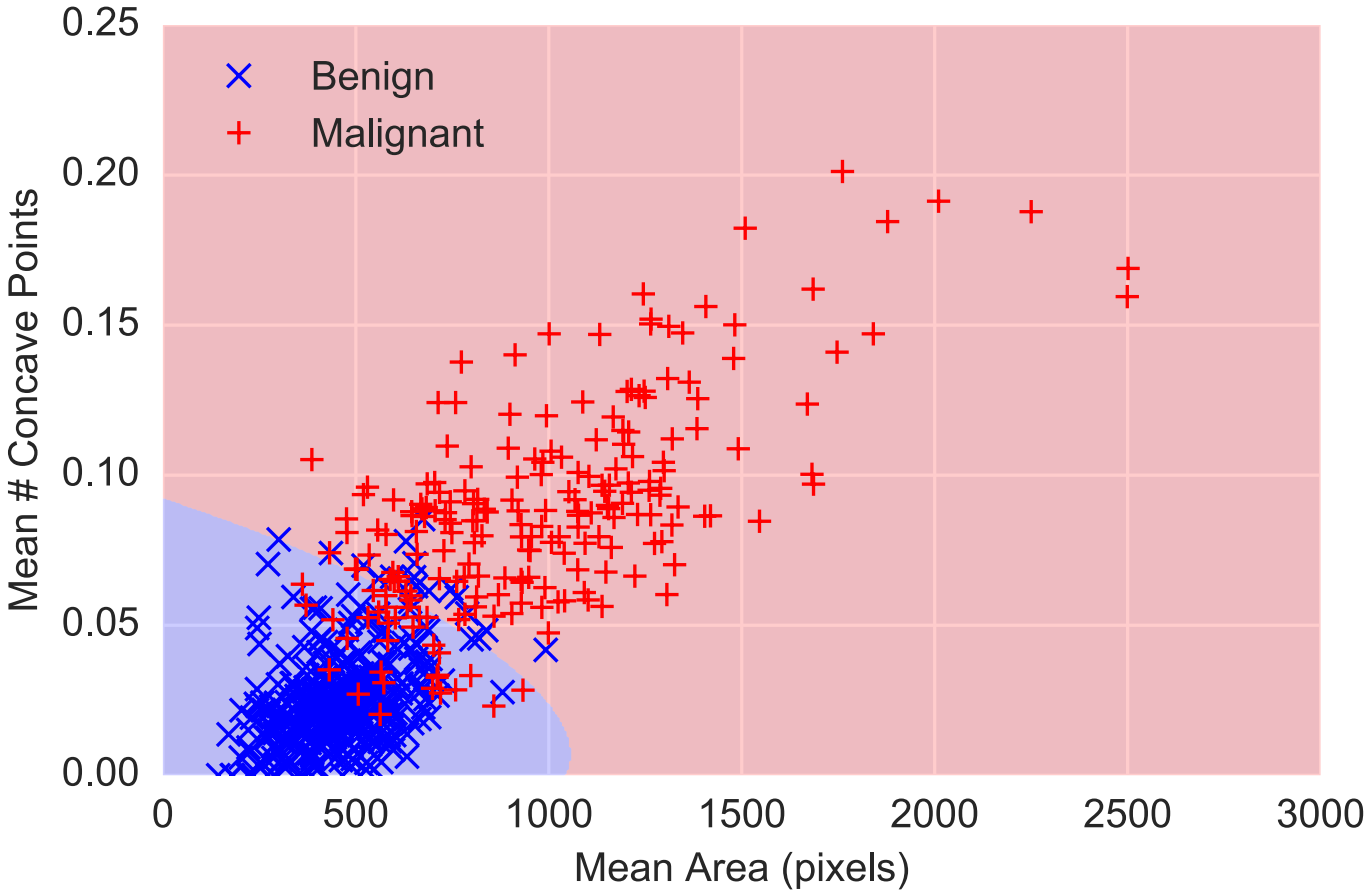
$$\text{minimize}_\theta \quad \sum_{i=1}^{m} \max\{1 - y^{(i)} \cdot \theta^T x^{(i)}, 0\} + \frac{\lambda}{2}\|\theta\|_2^2$$

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

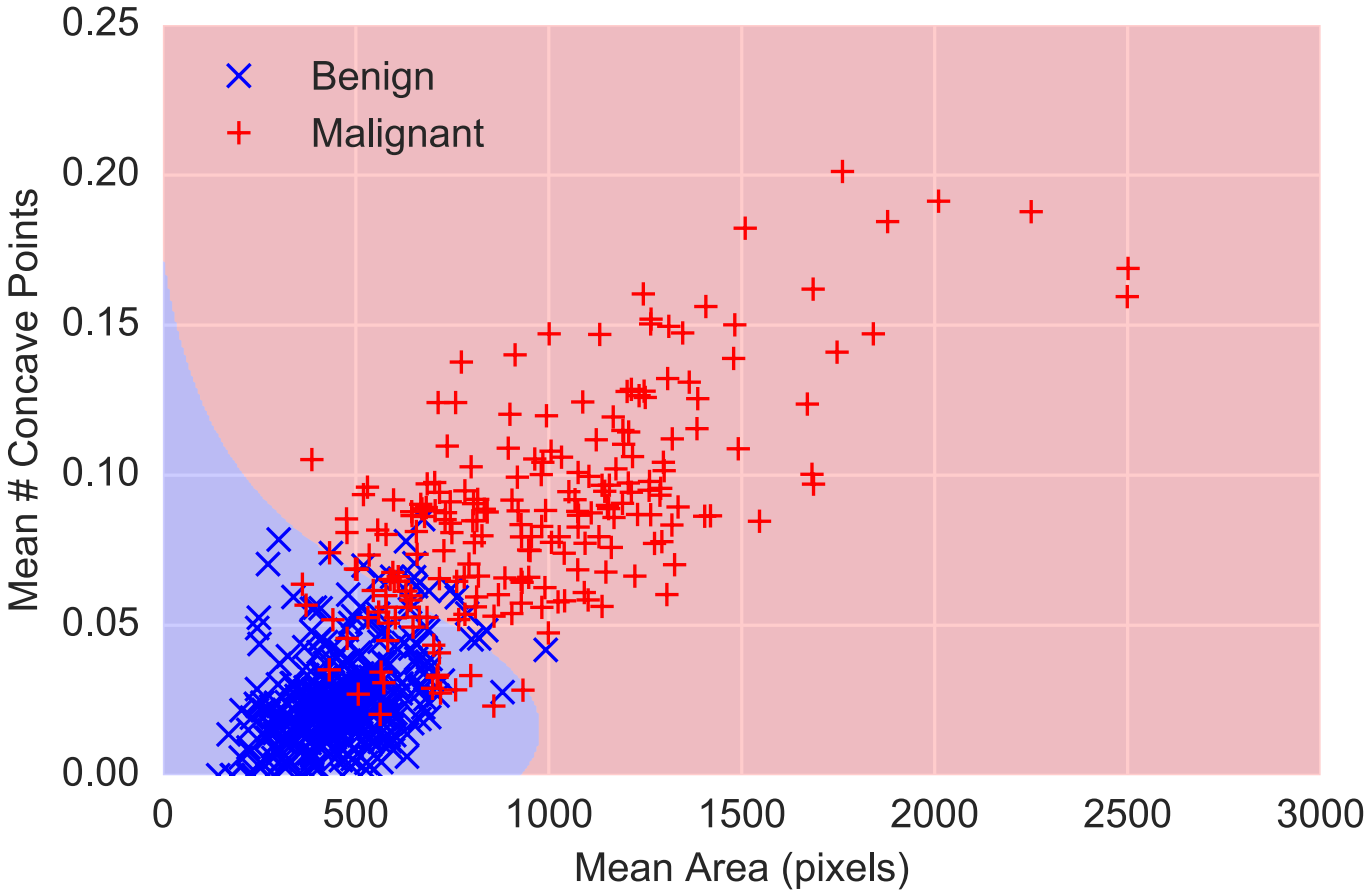
Linear SVM on cancer data set



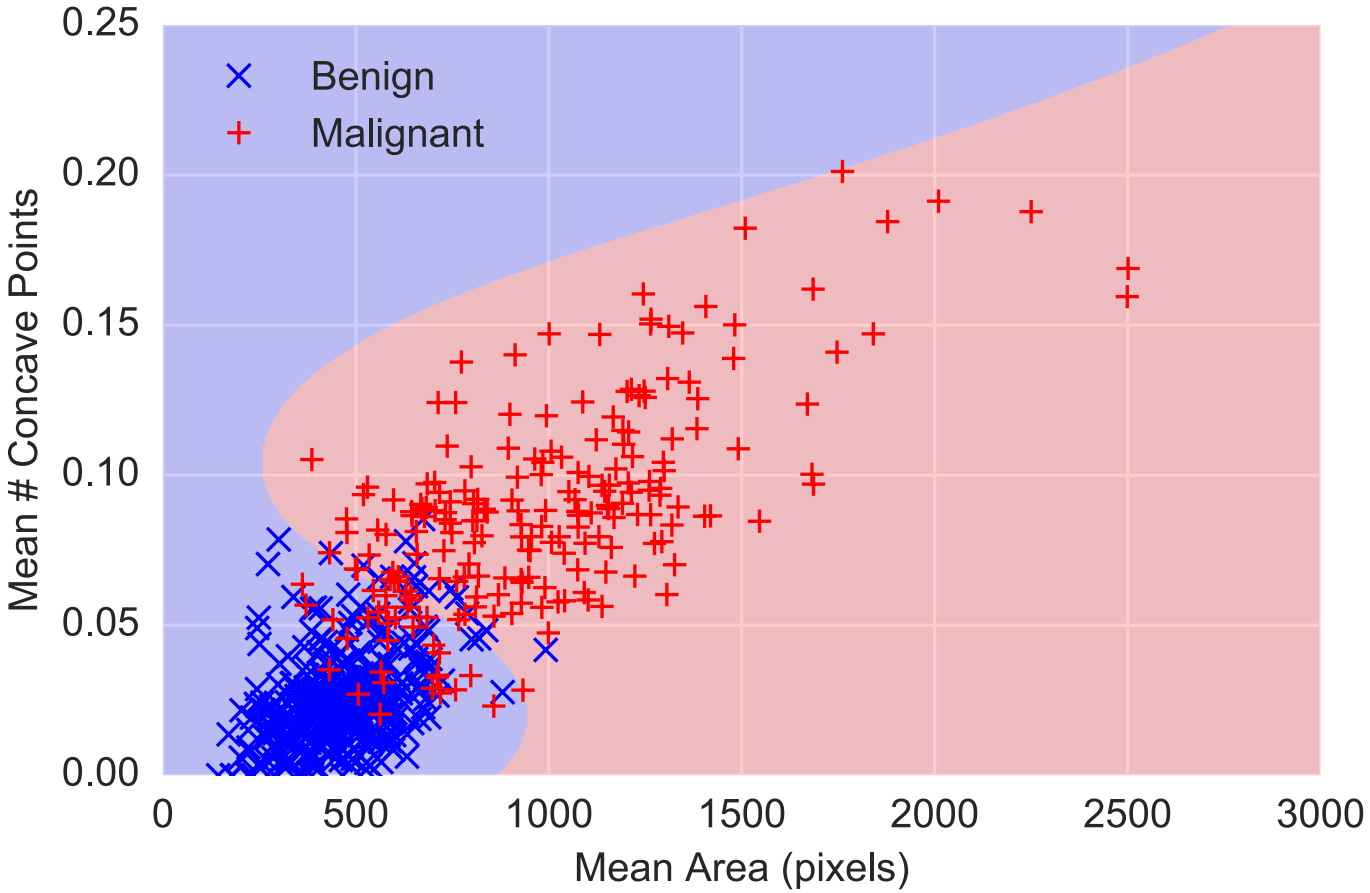
Polynomial features $d = 2$



Polynomial features $d = 3$



Polynomial features $d = 10$



LEARNING

Supervised learning paradigm

Training Data

$$\begin{matrix}(x^{(1)}, y^{(1)}) \\ (x^{(2)}, y^{(2)}) \\ (x^{(3)}, y^{(3)}) \\ \vdots\end{matrix}$$

Machine learning algorithm

Hypothesis function

$$y^{(i)} \approx h(x^{(i)})$$

Predictions

New example x

$$\hat{y} = h(x)$$

Unsupervised learning paradigm

Training Data

$$\left(x^{(1)}\right)$$
$$\left(x^{(2)}\right)$$
$$\left(x^{(3)}\right)$$
$$\vdots$$

Machine learning algorithm

Hypothesis function

$$? \approx h\left(x^{(i)}\right)$$

Predictions

New example x

$$? = h(x)$$

Three elements of unsupervised learning

It turns out the virtually all unsupervised learning algorithms can be considered in the same manner as supervised learning:

1. Define hypothesis function

2. Define loss function

3. Define how to optimize the loss function

But, what do a hypothesis function and loss function signify in the unsupervised setting?

UNSUPERVISED LEARNING

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

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

Hypothesis function: $h_\theta : \mathbb{R}^n \to \mathbb{R}^n$ **??????????????**

Want: approximate input given input, or $x^{(i)} \approx h_\theta(x^{(i)})$

Loss function: $\ell : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_+$

$$h_\theta(x) = \operatorname*{argmin}_{\mu \in \{\mu^{(1)}, \ldots \mu^{(k)}\}} \|\mu - x\|_2^2 \qquad \ell(h_\theta(x), x) = \|h_\theta(x) - x\|_2^2$$

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

Hypothesis and loss functions

The framework seems odd, what does it mean to have a hypothesis function approximate the input?

Can't we just pick $h_\theta(x) = x$?

The goal of unsupervised learning is to pick some *restricted* class of hypothesis functions that extract some kind of structure from the data (i.e., one that does not include the identity mapping above)

In this lecture, we'll consider two different algorithms that both fit the framework: k-means and principal component analysis

K-means graphically

The k-means algorithm is easy to visualize: given some collection of data points we want to find k centers such that all points are close to at least one center

K-means in unsupervised framework

Parameters of k-means are the choice of centers $\theta = \{\mu^{(1)}, \ldots \mu^{(k)}\}$, with $\mu^{(i)} \in \mathbb{R}^n$

Hypothesis function outputs the center closest to a point x

$$h_\theta(x) = \underset{\mu \in \{\mu^{(1)}, \ldots \mu^{(k)}\}}{\operatorname{argmin}} \|\mu - x\|_2^2$$

Loss function is squared error between input and hypothesis

$$\ell(h_\theta(x), x) = \|h_\theta(x) - x\|_2^2$$

Optimization problem is thus

$$\underset{\mu^{(1)}, \ldots \mu^{(k)}}{\operatorname{minimize}} \sum_{i=1}^{m} \left\| h_\theta\left(x^{(i)}\right) - x^{(i)} \right\|_2^2$$

K-MEANS

Non-convex optimization problem → locally good solutions

Given: dataset $x^{(i)}$, number of clusters k

Initialize k cluster centers:

$$\mu^{(j)} \leftarrow \text{Random}(x^{(i)}), \quad j = 1, \ldots, k$$

Repeat until convergence or bored:

1. **Compute cluster assignments:**

$$y^{(i)} = \underset{j}{\text{argmin}} \left\| \mu^{(j)} - x^{(i)} \right\|_2^2, \; i = 1, \ldots, m$$

2. **Re-compute the new cluster means:**

$$\mu^{(j)} \leftarrow \text{Mean}(\{x^{(i)} | y^{(i)} = j\}), j = 1, \ldots, k$$

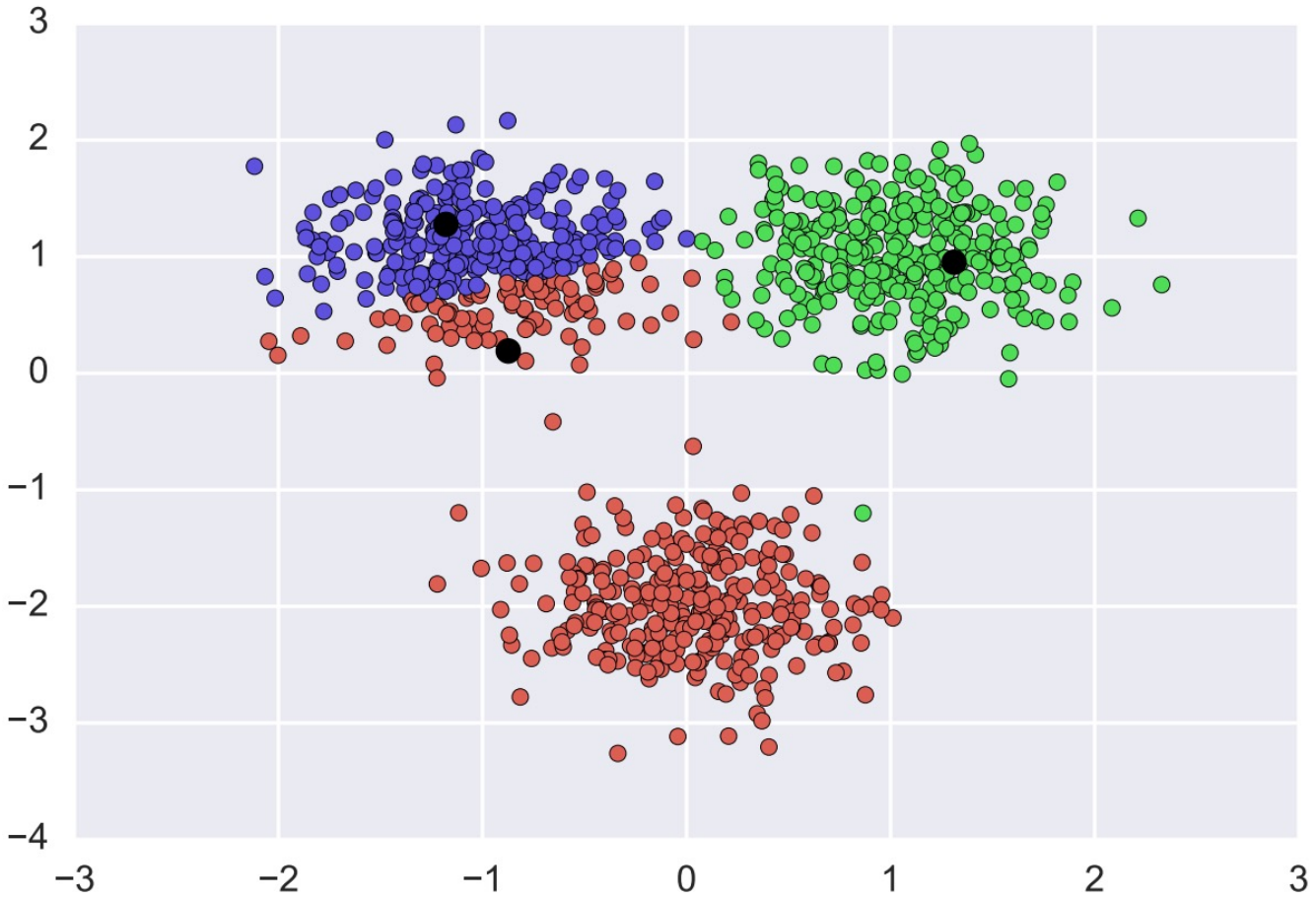
K-means in a few lines of code

Scikit-learn, etc, contains k-means implementations, but again these are pretty easy to write

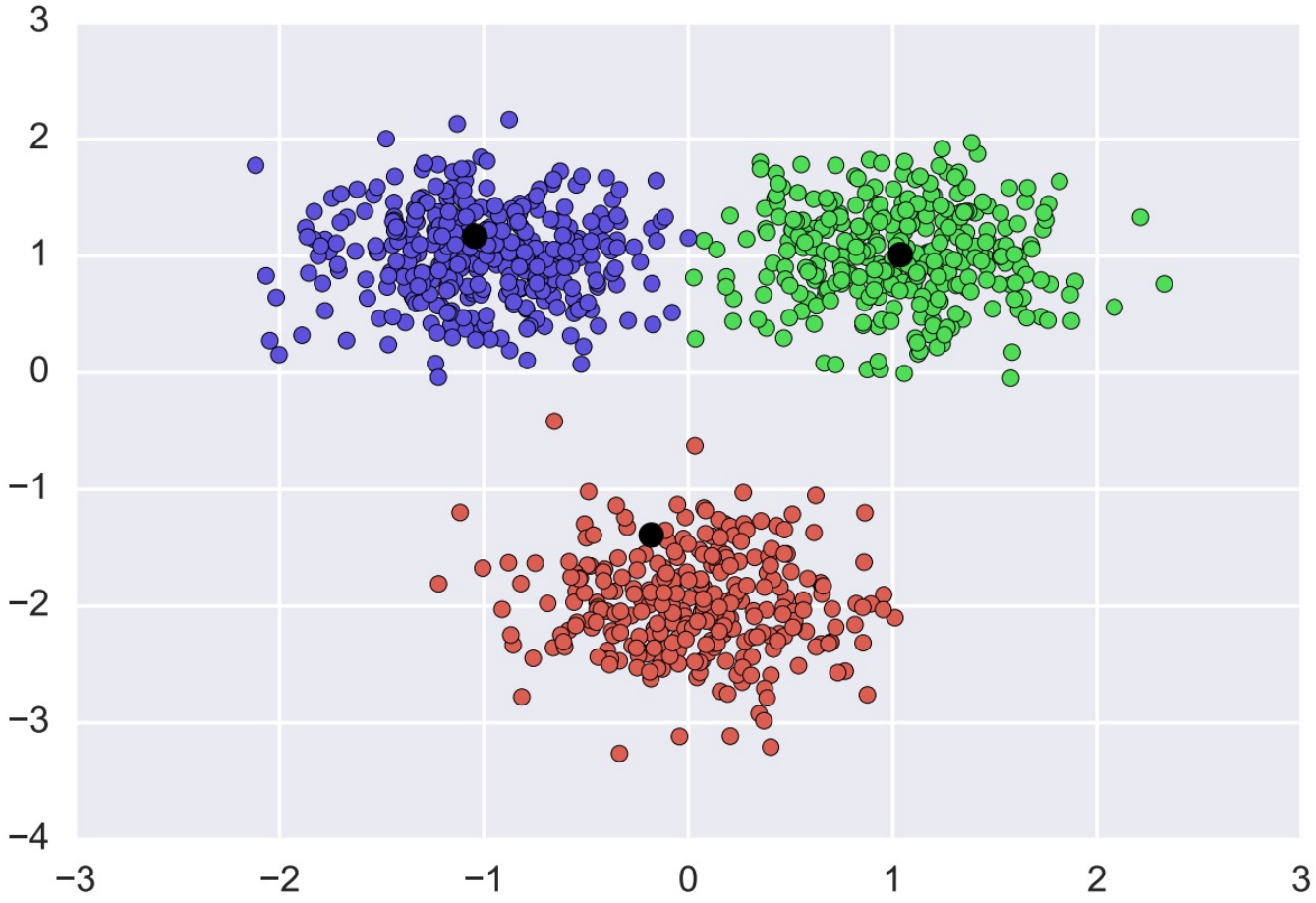
For better implementation, want to check for convergence as well as max number of iterations

```
def kmeans(X, k, max_iter=10):
    Mu = X[np.random.choice(X.shape[0],k),:]
    for i in range(max_iter):
        D = (-2*X.dot(Mu.T) + np.sum(X**2,axis=1)[:,None] +
             np.sum(Mu**2,axis=1))
        C = np.eye(k)[np.argmin(D,axis=1),:]
        Mu = C.T.dot(X)/np.sum(C,axis=0)[:,None]
    loss = np.linalg.norm(X - Mu[np.argmin(D,axis=1),:])**2
    return Mu, C, loss
```

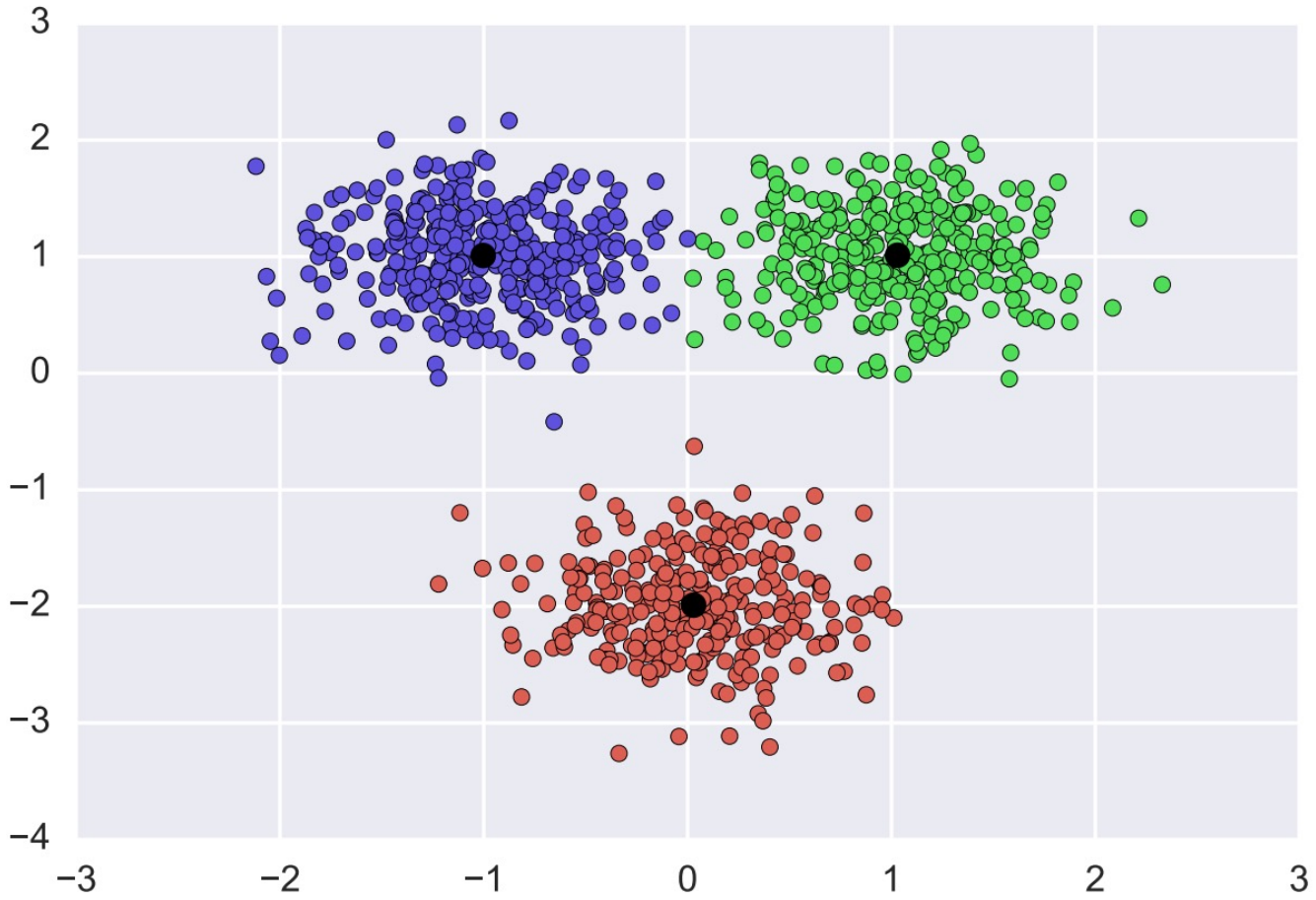
A (LUCKY) EXAMPLE



A (LUCKY) EXAMPLE



A (LUCKY) EXAMPLE



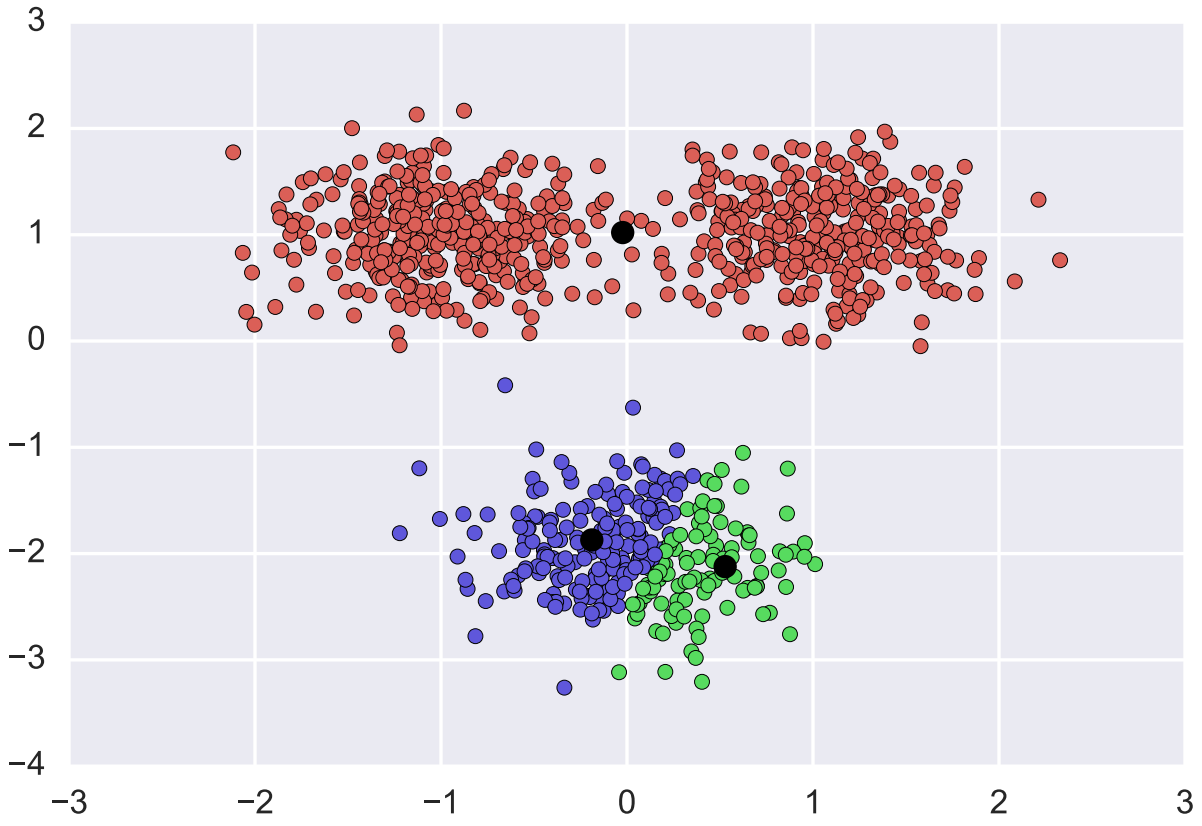
Possibility of local optima

Since the k-means objective function has local optima, there is the chance that we convert to a less-than-ideal local optima

Especially for large/high-dimensional datasets, this is not hypothetical: k-means will usually converge to a different local optima depending on its starting point

Convergence of k-means (bad)

Convergence of k-means (bad)



Convergence of k-means (bad)

Addressing poor clusters

Many approaches to address potential poor clustering: e.g. randomly initialize many times, take clustering with lowest loss

A common heuristic, k-means++: when initializing means, don't select $\mu^{(i)}$ randomly from all clusters, instead choose $\mu^{(i)}$ sequentially, sampled with probability proportion to the minimum squared distance to all other centroids

After these centers are initialized, run k-means as normal

K-means++

Given: Data set $\left(x^{(i)}\right)_{i=1,\ldots,m}$, # clusters k

Initialize:

$$\mu^{(1)} \leftarrow \mathrm{Random}\left(x^{(1:m)}\right)$$

For $j = 2, \ldots, k$:

Select new cluster:

$$\mu^{(j)} \leftarrow \mathrm{Random}\left(x^{(1:m)}, p^{(1:m)}\right)$$

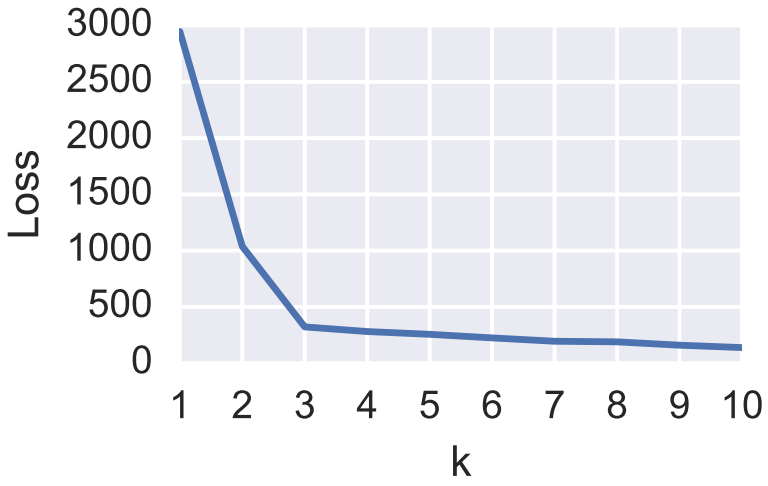
where probabilities $p^{(i)}$ given by

$$p^{(i)} \propto \min_{j'<j} \left\| \mu^{(j')} - x^{(i)} \right\|_2^2$$

How to select k?

There's no "right" way to select k (number of clusters): larger k virtually always will have lower loss than smaller k, even on a hold out set

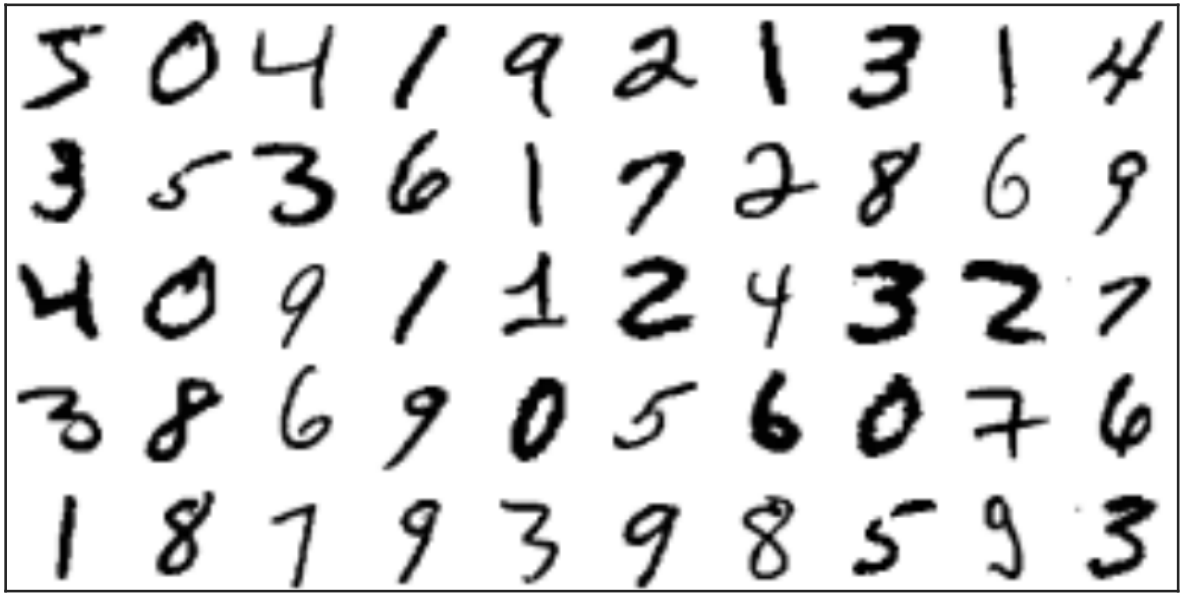
Instead, it's common to look at the loss function as a function of increasing k, and stop when things look "good" (lots of other heuristics, but they don't convincingly outperform this)



Example on real data

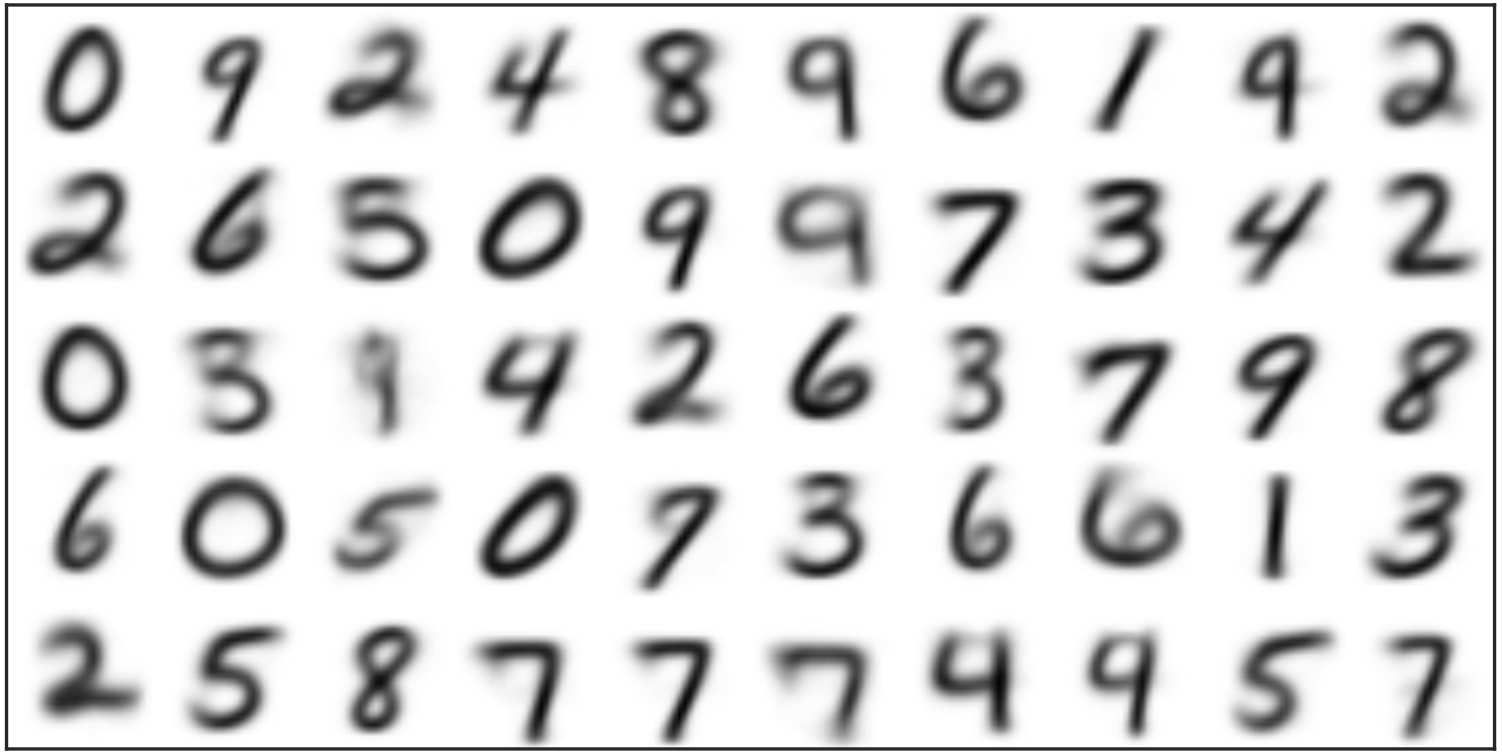
MNIST digit classification data set (used in question for 688 HW4)

60,000 images of digits, each 28x28

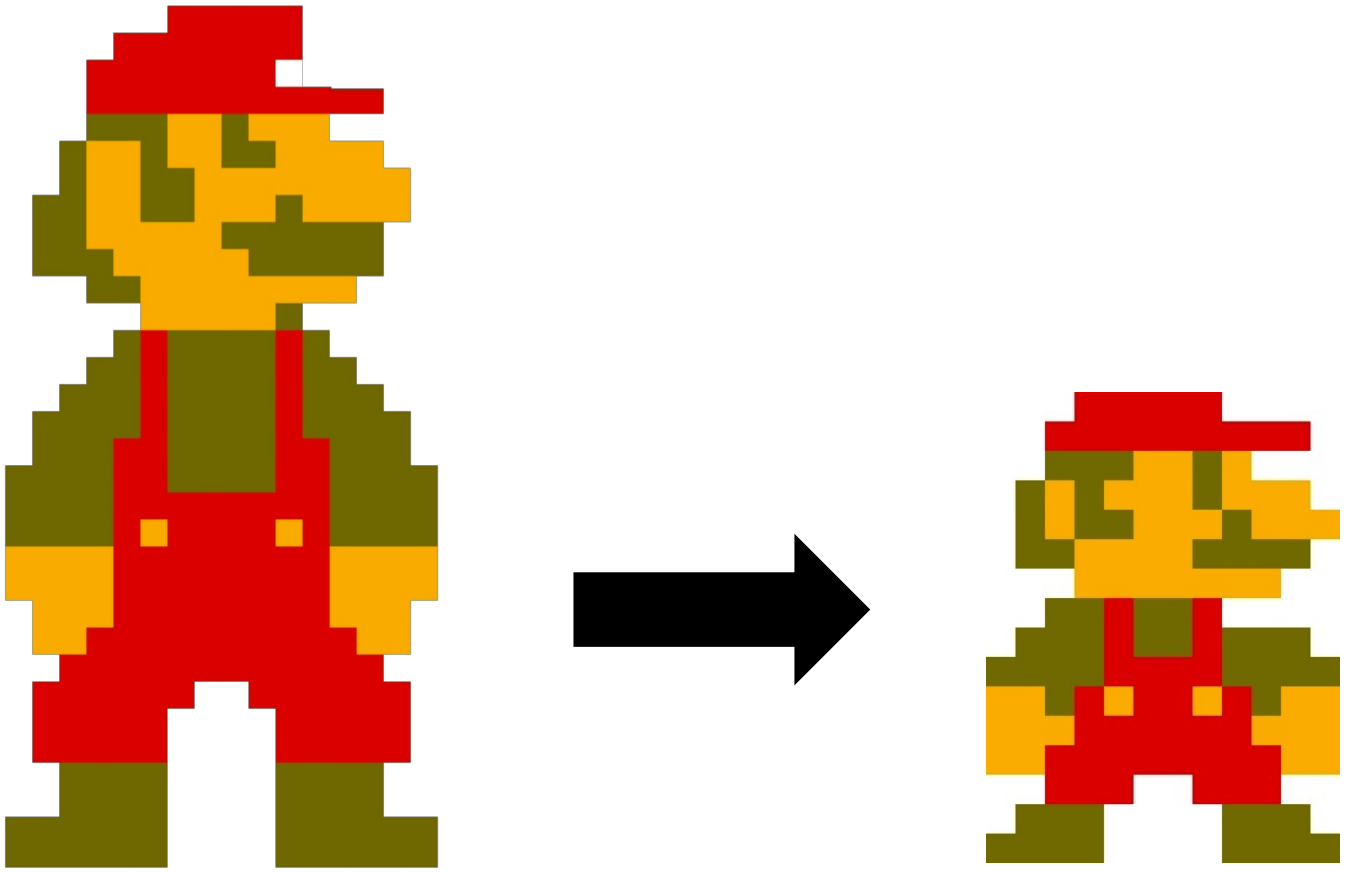


K-means run on MNIST

Means for k-means run with k=50 on MNIST data



DIMENSIONALITY REDUCTION



Thanks to: Zico Kolter

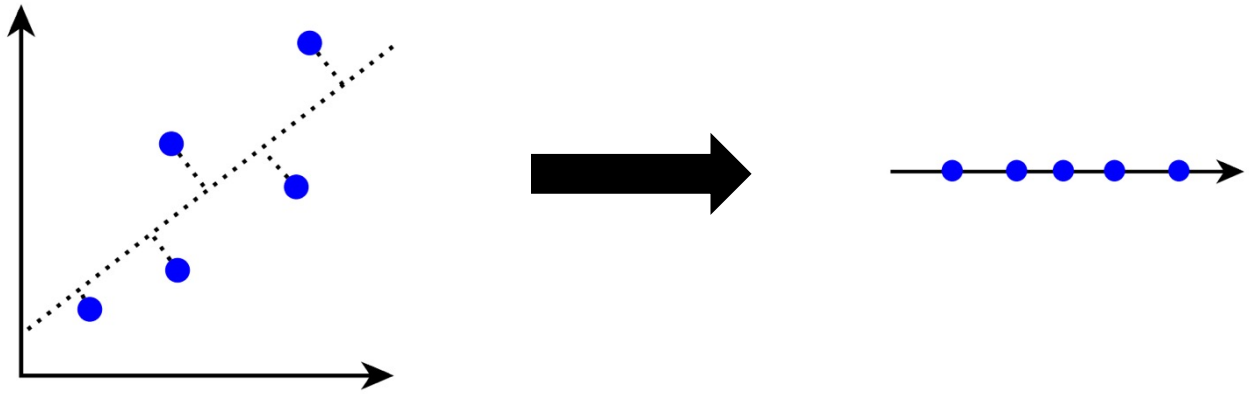
PRINCIPAL COMPONENT ANALYSIS (PCA)

So you've measured lots of features …

- Overfitting, interpretability issues, visualization, computation

Can we combine raw features into new features that yield a simpler description of the same system?

Principal component analysis (PCA) does this by preserving the axis of major variation in the data:



Images: Zico Kolter

PRINCIPAL COMPONENT ANALYSIS (PCA)

Assume: data is normalized ??????????

- Zero mean, unit (= 1) variance

Hypothesis function:

$$h_\theta(x) = UWx, \theta = \{U \in \mathbb{R}^{n \times k}, W \in \mathbb{R}^{k \times n}\}$$

- First multiply input by low rank matrix *W* ("compress" it), then map back into the initial space using *U*

Loss function: squared distance (like k-means)

$$\ell(h_\theta(x), x) = \|h_\theta(x) - x\|_2^2$$

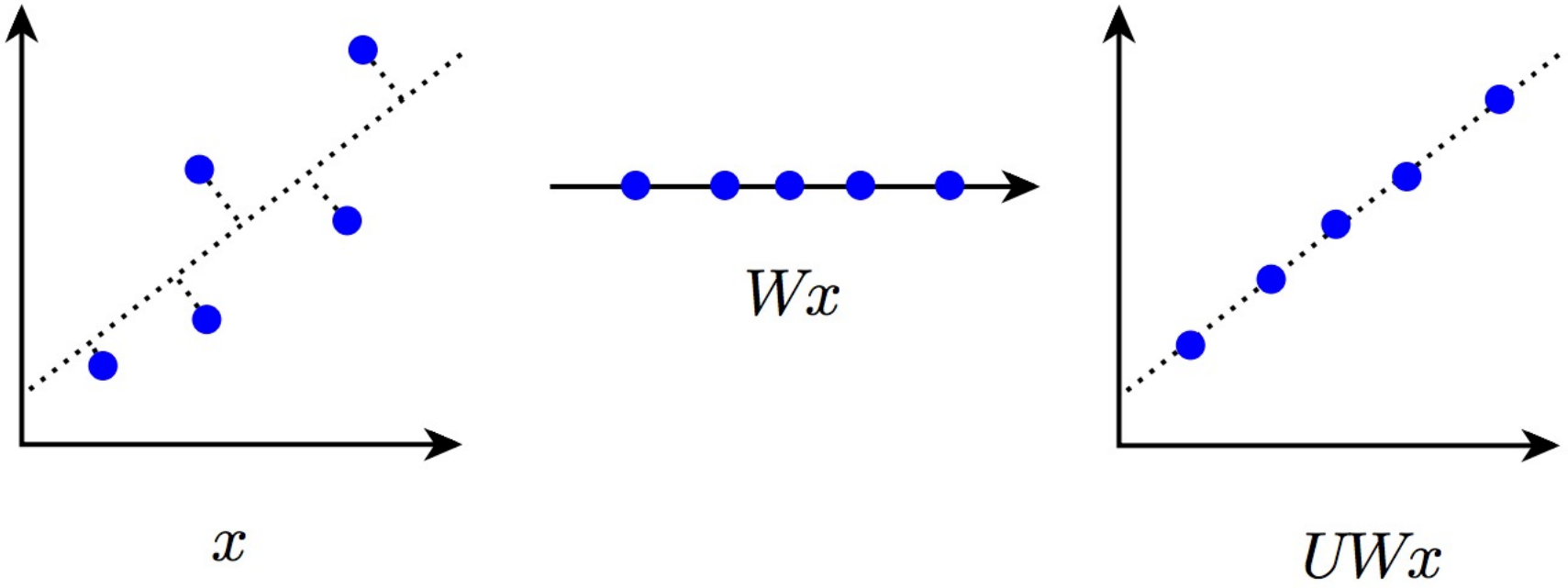
Optimization problem:

$$\underset{U,W}{\text{minimize}} \sum_{i=1}^{m} \left\| UWx^{(i)} - x^{(i)} \right\|_2^2$$

PRINCIPAL COMPONENT ANALYSIS (PCA)

Dimensionality reduction: main use of PCA for data science applications

If $h_\theta(x) = UWx$, then $Wx \in \mathbb{R}^k$ is a reduced (probably with some loss) representation of input features x



PRINCIPAL COMPONENT ANALYSIS (PCA)

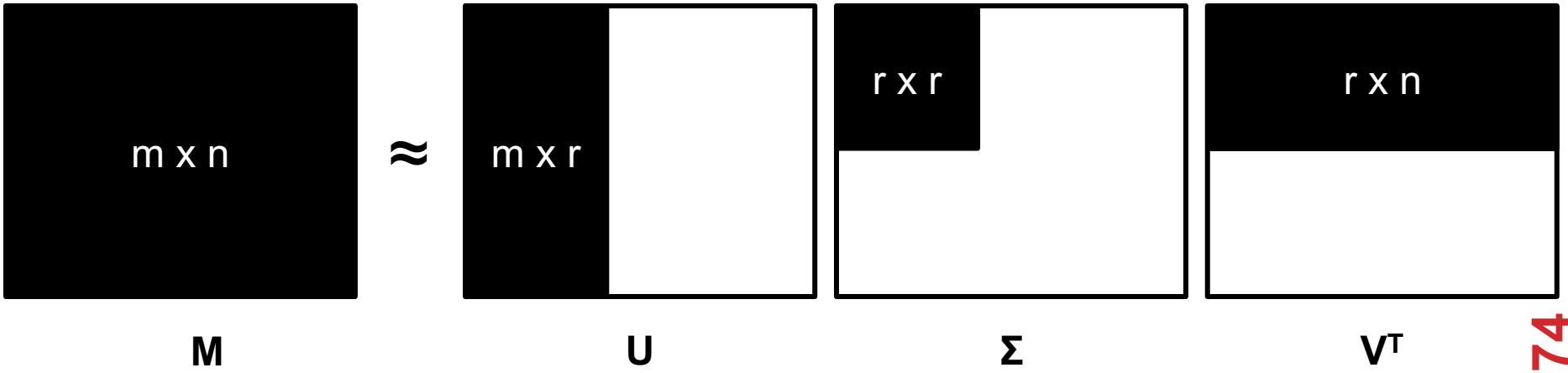
CMSC422
MATH240

$$\underset{U,W}{\text{minimize}} \sum_{i=1}^{m} \left\| UWx^{(i)} - x^{(i)} \right\|_2^2$$

PCA optimization problem is non-convex ??????????

We can solve the problem exactly using the singular value decomposition (SVD):

- Factorize matrix M = U Σ V^T (also used to approximate)



PRINCIPAL COMPONENT ANALYSIS (PCA)

Solving PCA exactly using the SVD:

1. **Normalize input data, pick #components k**

2. **Compute (exact) SVD of X = U Σ V^T**

3. **Return:** $h_\theta(x) = UWx$

- **U = $V_{:,1:k}$ $\Sigma^{-1}_{1:k,1:k}$**
- **W = $V^T_{:,1:k}$**

Loss is $\sum_{i=k+1}^{n} \Sigma_{ii}^2$

PCA IN PYTHON

Can roll your own PCA easily (assuming a call to SVD via SciPy or similar) …

… or just use Scikit-Learn:

```
from sklearn.decomposition import PCA

X=np.array([[-1,-1],[-2,-1],[-3,-2],[1,1],[2,1],[3,2]])

# Fit PCA with 2 components (i.e., two final features)
pca = PCA(n_components=2)
pca.fit(X)
print(pca.explained_variance_ratio_)
```

```
[ 0.99244... 0.00755...]
```

Looks like our data basically sit on a line

HOW TO USE PCA & FRIENDS IN PRACTICE

Unsupervised learning methods are useful for EDA

- Cluster or reduce to a few dimensions and visualize!

Also useful as data prep before supervised learning!

1. Run PCA, get W matrix
2. Transform $\tilde{x}^{(i)} = W x^{(i)}$ – (reduce colinearity, dimension)
3. Train and test your favorite supervised classifier

Or use k-means to set up radial basis functions (RBFs):

1. Get k centers $\mu^{(1)}, \ldots, \mu^{(k)}$
2. Create RBF features $\phi_j^{(i)} = \exp\left(-\frac{\|x^{(i)} - \mu^{(j)}\|_2^2}{2\sigma^2}\right)$

RECOMMENDER SYSTEMS & COLLABORATIVE FILTERING

NETFLIX PRIZE

Recommender systems: predict a user's rating of an item

	Twilight	Wall-E	Twilight II	TFotF
User 1	+1	-1	+1	?
User 2	+1	-1	?	?
User 3	-1	+1	-1	+1

Netflix Prize: $1MM to the first team that beats our in-house engine by 10%

- **Happened after about three years**
- **Model was never used by Netflix for a variety of reasons**
  - Out of date (DVDs vs streaming)
  - Too complicated / not interpretable

RECOMMENDER SYSTEMS

Recommender systems feel like:

- Supervised learning (we know the user watched some movies, so these are like labels)
- Unsupervised learning (we want to find latent structure, e.g., genres of movies)

They fall somewhere in between, in "Information Filtering" or Information Retrieval" …

- … but we can still just phrase the problem in terms of hypothesis classes, loss functions, and optimization problems

PREDICTION

Pure user information:

- Age
- Location
- Profession/Salary

Pure item information:

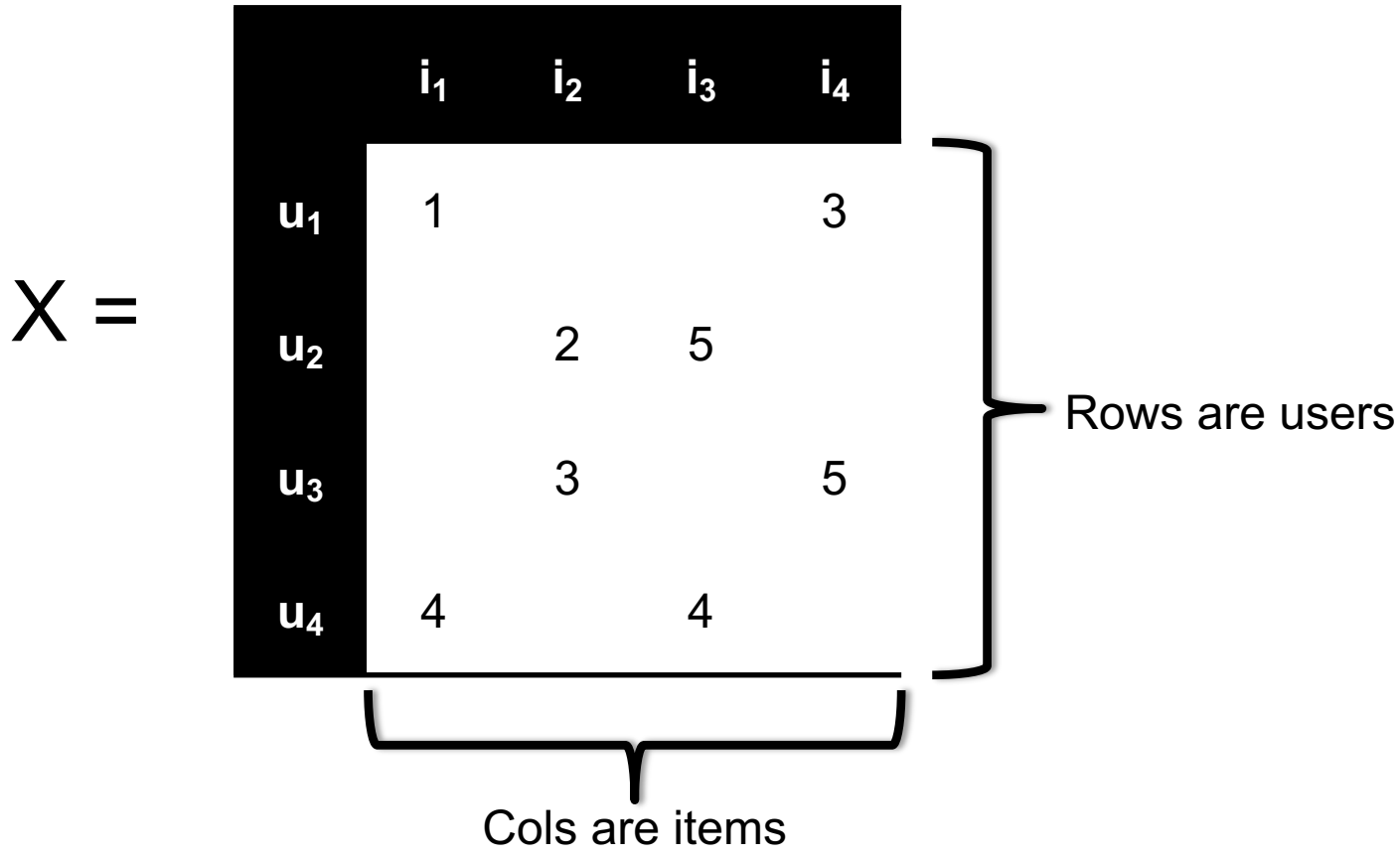
- Movie budget
- Main actors
- Is it a Netflix release?

User-item information:

- Which items are most similar to those I've watched before?
- Which users are most similar to me, and what did they watch?

COLLABORATIVE FILTERING

Collaborative filtering (CF): recommender systems that predict based only on the expressed preferences of other users for an item



MATRIX VIEW

Goal: “fill in” the matrix

	i_1	i_2	i_3	i_4
u_1	1	?	?	3
u_2	?	2	5	?
u_3	?	3	?	5
u_4	4	?	4	?

The matrix is sparse, but the empty cells are not (necessarily) zero!

APPROACHES TO CF

User-user:

- Find users who look like me – based on items that we've both rated
- Predict scores for my unrated items as average of those users

Item-item:

- Find similar items (based on scores from all users who have rated), predict scores for other users based off this

Matrix factorization:

- Find a low-rank decomposition of X that agrees (exactly, approximately) at the observed values

APPROACH #1: ITEM-BASED CF EX: INFER (USER 1, ITEM 3)

	Item 1	Item 2	Item 3	Item 4	Item 5
User 1	8	1	?	2	7
User 2	2	?	5	7	5
User 3	5	4	7	4	7
User 4	7	1	7	3	8
User 5	1	7	4	6	?
User 6	8	3	8	3	7

Example due to Qiang Yang [HKUST]

HOW TO CALCULATE SIMILARITY (ITEMS 3 AND 5)?

	Item 1	Item 2	Item 3	Item 4	Item 5
User 1	8	1	?	2	7
User 2	2	?	5	7	5
User 3	5	4	7	4	7
User 4	7	1	7	3	8
User 5	1	7	4	6	?
User 6	8	3	8	3	7

Example due to Qiang Yang [HKUST]

SIMILARITY BETWEEN ITEMS

Item 3	Item 4	Item 5
?	2	7
5	7	5
7	4	7
7	3	8
4	6	?
8	3	7

How should we calculate the similarity between two items (e.g., items 3 and 5)?

We've done this before in a different context!

SIMILARITY BETWEEN ITEMS

Item 3	Item 5
?	7
5	5
7	7
7	8
4	?
8	7

Only consider users (i.e., rows) who have rated both items (i.e., non-empty)

One approach: For each user:

- Calculate difference in ratings for the two items
- Take the average of this difference over the users

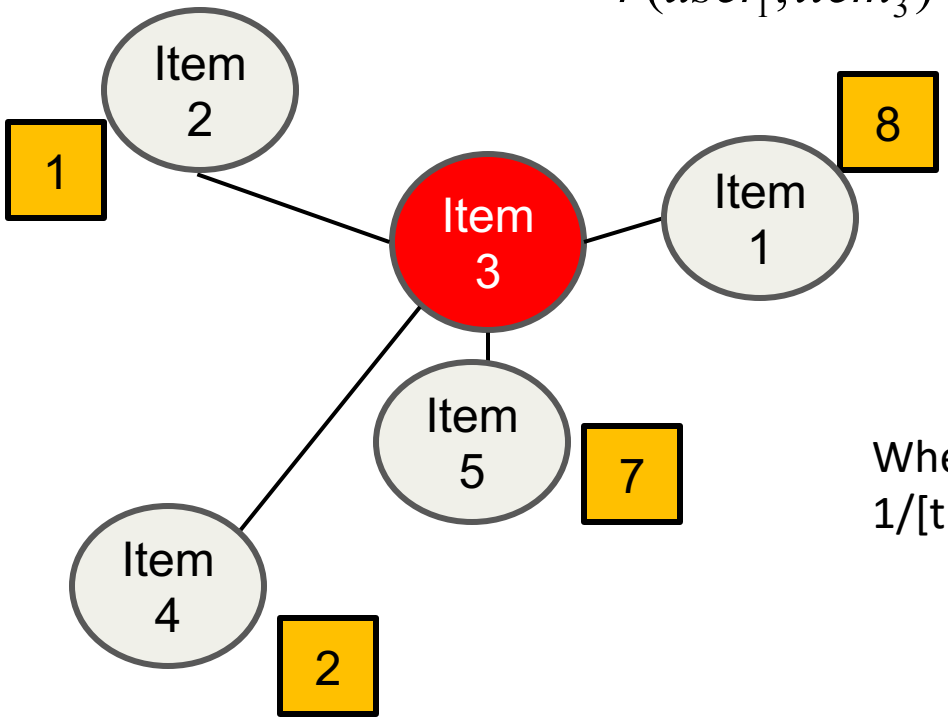
Another approach: cosine similarity!

$$\text{sim(item 3, item 5)} = \text{cosine}((5, 7, 7, 8), (5, 7, 8, 7))$$

$$= (5*5 + 7*7 + 7*8 + 8*7)/(\text{sqrt}(5^2+7^2+7^2+8^2)* \text{sqrt}(5^2+7^2+8^2+7^2))$$

Can also use Pearson Correlation Coefficients (also in user-user approach)

PREDICTION: CALCULATING RANKING R(USER1,ITEM3)



$$r(user_1, item_3) = \alpha * \{r(user_1, item_1) sim(item_1, item_3) + r(user_1, item_2) sim(item_2, item_3) + r(user_1, item_4) sim(item_4, item_3) + r(user_1, item_5) sim(item_5, item_3)\}$$

Where α is a normalization factor, which is 1/[the sum of all sim($item_i$,$item_3$)].

APPROACH #2: CF VIA MATRIX FACTORIZATION

Want: all entries i, j of ratings matrix $X \in \mathbb{R}^{m \times n}$

Idea: approximate as $u_i^T v_j$

- User-specific weights $u_i \in \mathbb{R}^k$
- Item-specific weights $v_j \in \mathbb{R}^k$

Hypothesis function: ??????????

$$h_\theta(i,j) = u_i^T v_j, \qquad \theta = \{u_{1:m}, v_{1:n}\}$$

Loss function: least squares on observed entries

$$\ell(h_\theta(i,j), X_{ij}) = (h_\theta(i,j) - X_{ij})^2$$

Optimization problem: (S is observed entries)

$$\underset{\theta}{\text{minimize}} \sum_{i,j \in S} \ell\left(h_\theta(i,j), X_{ij}\right)$$

OPTIMIZATION FOR CF

Matrix factorization is non-convex; we won't go for perfect

Consider the objective w.r.t. a single term u_i:

$$\underset{u_i}{\text{minimize}} \sum_{j:(i,j)\in S} (v_j^T u_i - X_{ij})^2$$

Just a least squares problem! Analytic solution (from earlier):

$$u_i = \left(\sum_{j:(i,j)\in S} v_j v_j^T \right)^{-1} \left(\sum_{j:(i,j)\in S} v_j X_{ij} \right)$$

Idea (not optimal): repeatedly solve for all u_i, v_j until converged or bored

WHAT ARE WE DOING?

Factorizing the ratings matrix X as

$$X \approx UV, \qquad U \in \mathbb{R}^{m \times k}, V \in \mathbb{R}^{k \times n}$$

with

$$U = \begin{bmatrix} - & u_1^T & - \\ & \vdots & \\ - & u_m^T & - \end{bmatrix}, \qquad V = \begin{bmatrix} | & & | \\ v_1 & \cdots & v_n \\ | & & | \end{bmatrix}$$

But we are only penalizing mismatches between UV and X at the observed entries in X

ISN'T THIS JUST PCA?

PCA also performs a factorization $X \approx UV$

- **Or more precisely** $X^T = UV = U(Wx)$

In PCA, all entries are observed (or imputed beforehand)

Simplifies the solution to PCA (which we can solve exactly via the SVD) versus CF via matrix factorization (which we cannot)

(SOME MORE) RECOMMENDER SYSTEMS (ISH)

ASSOCIATION RULES

Last time: CF systems give predictions based on other users' scores of the same item

Complementary idea: Find rules that associate the presence of one set of items with that of another set of items

Customers who bought this item also bought

ThinkGeek Plush Unicorn Slippers, One Size, White
395
$7.77

Adult New Purple Unicorn Onesie Pajamas Kigurumi Cosplay Costumes Animal Outfit
168
$23.99 - $28.99

EOS ~ Holiday 2015 Limited Edition Decorative Lip Balm Collection
156
$5.24 - $22.99

FORMAT OF ASSOCIATION RULES

Typical Rule form:

- **Body → Head**
- **Body and Head can be represented as sets of items (in transaction data) or as conjunction of predicates (in relational data)**
- **Support and Confidence**
  - Usually reported along with the rules
  - Metrics that indicate the strength of the item associations

Examples:

- {diaper, milk} → {beer} [support: 0.5%, confidence: 78%]
- buys(x, "bread") /\ buys(x, “eggs") → buys(x, "milk") [sup: 0.6%, conf: 65%]
- major(x, "CS") /\ takes(x, "DB") → grade(x, "A") [1%, 75%]
- age(x,30-45) /\ income(x, 50K-75K) → owns(x, SUV)
- age=“30-45”, income=“50K-75K” → car=“SUV”

Thanks to: Bamshad Mobasher

ASSOCIATION RULES: BASIC CONCEPTS

Let D be database of transactions

Transaction ID	Items
1000	A, B, C
2000	A, B
3000	A, D
4000	B, E, F

Let I be the set of items that appear in the database:

- **e.g., I = {A, B, C, D, E, F}**

Each transaction t is a subset of I

A rule is an implication among itemsets X and Y, of the form by X → Y, where $X \subset I$, $Y \subset I$, and $X \cap Y = \varnothing$

- **e.g.: {B,C} → {A}**

Thanks to: Bamshad Mobasher

ASSOCIATION RULES: BASIC CONCEPTS

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Itemset

- **A set of one or more items**
  - E.g.: {Milk, Bread, Diaper}
- **k-itemset**
  - An itemset that contains k items

Support count (σ)

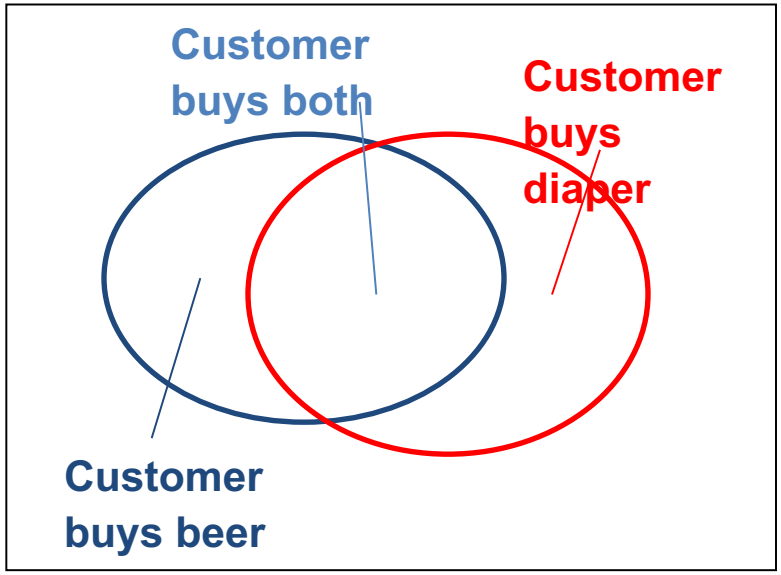
- Frequency of occurrence of an itemset (number of transactions in which it appears)
- E.g. σ({Milk, Bread,Diaper}) = 2

Support

- Fraction of the transactions in which an itemset appears
- E.g. s({Milk, Bread, Diaper}) = 2/5

Frequent Itemset

- An itemset whose support is greater than or equal to a *minsup* threshold



ASSOCIATION RULES: BASIC CONCEPTS

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Association Rule

- X → Y, where X and Y are non-overlapping itemsets
- {Milk, Diaper} → {Beer}

Rule Evaluation Metrics

- **Support (s)**
  - Fraction of transactions that contain both **X** and **Y**
  - **i.e., support of the itemset $X \cup Y$**
- **Confidence (c)**
  - Measures how often items in **Y** appear in transactions that contain **X**

Example:

$$\{\text{Milk}, \text{Diaper}\} \rightarrow \text{Beer}$$

$$s = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{|\text{D}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk}, \text{Diaper}, \text{Beer})}{\sigma(\text{Milk}, \text{Diaper})} = \frac{2}{3} = 0.67$$

ASSOCIATION RULES: INTERESTINGNESS

Another interpretation of support and confidence for *X → Y*

- **Support is the probability that a transaction contains {X ∪ Y} or Pr(X /\ Y)**

$$\text{support}(X \rightarrow Y) = \text{support}(X \cup Y) = \sigma(X \cup Y) / |D|$$

- **Confidence is the conditional probability that a transaction will contains Y given that it contains X or Pr(Y | X)**

$$\text{confidence}(X \rightarrow Y) = \sigma(X \cup Y) / \sigma(X)$$
$$= \text{support}(X \cup Y) / \text{support}(X)$$

ASSOCIATION RULES: INTERESTINGNESS

Other considerations of how interesting a rule is:

$$\text{lift}(X \Rightarrow Y) = \frac{\text{supp}(X \cup Y)}{\text{supp}(X) \times \text{supp}(Y)}$$

If lift is equal to 1 ??????????

- Body X and Head Y are independent

If lift is greater than 1 ?????????

- Body X and Head Y are in some sense dependent

Conviction measures frequency of X and Y occurring together, vs. how frequently X occurs but not Y

Many others …

ASSOCIATION RULES IN PRACTICE

Orange3 is a {GUI, Python API, …} that:

- Enumerates frequent itemsets
- Performs association rule mining
- (Wrapper calls to, shared functionality with, Scikit-Learn)

```
conda install –c ales-erjavec orange3
```

More information:
https://blog.biolab.si/2016/04/25/association-rules-in-orange/

In general:

- **Can be useful for interpretable, fast data mining**
- **Typically doesn't consider order, scalability issues …**