Tufts

Class #04: Linear Methods

Machine Learning (COMP 135): M. Allen, 16 Sept. 19

The General Learning Problem

- We want to learn functions from inputs to outputs, where each input has n features:

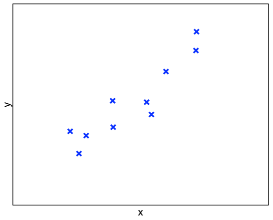
Inputs $\langle x_1, x_2, \ldots, x_n \rangle$, with each feature x_i from domain X_i.
Outputs y from domain Y.

Function to learn: $f : X_1 \times X_2 \times \cdots \times X_n \rightarrow Y$

- The type of learning problem we are solving really depends upon the type of the output domain, Y

1. If output $Y \in R$ (a real number), this is regression
2. If output Y is a finite discrete set, this is classification

Linear Regression



- In general, we want to learn a hypothesis function h that minimizes our error relative to the actual output function f
- Often we will assume that this function h is linear, so the problem becomes finding a set of weights that minimize the error between f and our function:

$$h(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

An Error Function: Least Squared Error

- For a chosen set of weights, $\mathbf{w}$, we can define an error function as the ***squared residual*** between what the hypothesis function predicts and the actual output, summed over all N test-cases:

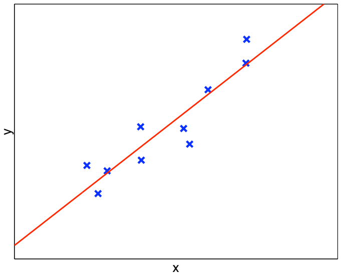
$$Loss(\mathbf{w}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2$$

- Learning is then the process of finding a weight-sequence that ***minimizes*** this loss:

$$\mathbf{w}^{\star} = \arg\min_{w} Loss(\mathbf{w})$$

- ***Note***: Other loss-functions are commonly used (but the basic learning problem remains the same)

An Example



- For the data given, the best fit for a simple linear function of x is as follows:

$$h(x) \longleftarrow \quad y = 1.05 + 1.60x$$

Finding Minimal-Error Weights

$$\mathbf{w}^\star = \arg\min_w Loss(\mathbf{w})$$

- We can ***in principle*** solve for the weight with least error ***analytically***

1. Create data matrix with one training input example per row, one feature per column, and output vector of all training outputs

$$\mathbf{X} = \begin{bmatrix} f_{11} & f_{12} & \cdots & f_{1n} \\ f_{21} & f_{22} & \cdots & f_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{N1} & f_{N2} & \cdots & f_{Nn} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

2. ***Solve*** for the minimal weights using linear algebra (for large data, requires optimized routines for finding matrix inverses, doing multiplications, etc., as well as for certain matrix properties to hold, which are not universal):

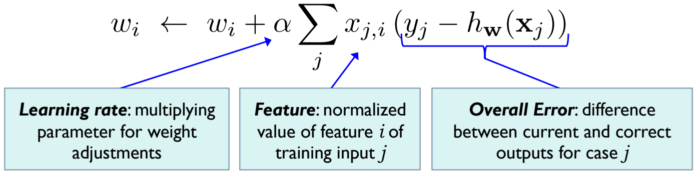
$$\mathbf{w}^\star = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

Finding Minimal-Error Weights

$$\mathbf{w}^\star = \arg\min_w Loss(\mathbf{w})$$

- Weights that minimize error can instead be found (or at least approximated) using gradient descent:

1. ***Loop repeatedly*** over all weights w_i, updating them based on their "contribution" to the overall error:



$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i} \underbrace{(y_j - h_{\mathbf{w}}(\mathbf{x}_j))}$$

Learning rate: multiplying parameter for weight adjustments

Feature: normalized value of feature *i* of training input *j*

Overall Error: difference between current and correct outputs for case *j*

2. ***Stop on convergence***, when maximum update on any weight (Δ) drops below some threshold (Θ); alternatively, stop when change in error/loss grows small enough

Updating Weights

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i} \, (y_j - h_{\mathbf{w}}(\mathbf{x}_j))$$

- For each value i, the update equation takes into account:

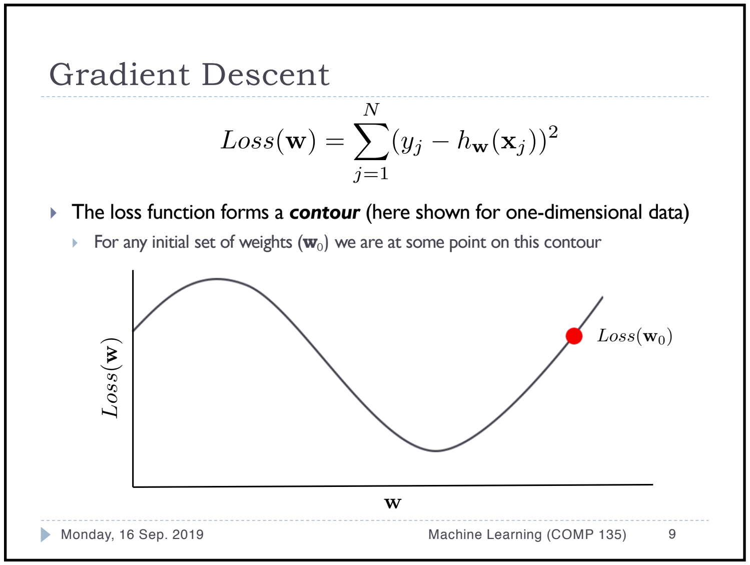
1. The ***current*** weight-value, w_i
2. The ***difference*** (positive or negative) between the current hypothesis for input *j* and the known output: $(y_j - h_{\mathbf{w}}(\mathbf{x}_j))$
3. The i-th feature of the data, $x_{j,i}$

- When doing this update, we must remember that for n data features, we have $(n + 1)$ weights, including the bias, w_0

$$h(x_1, x_2, \ldots, x_n) = \boxed{w_0} + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

- It is presumed that the related "feature" $x_{j,0} = 1$ in every case, and so the update for the bias weight becomes:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j))$$

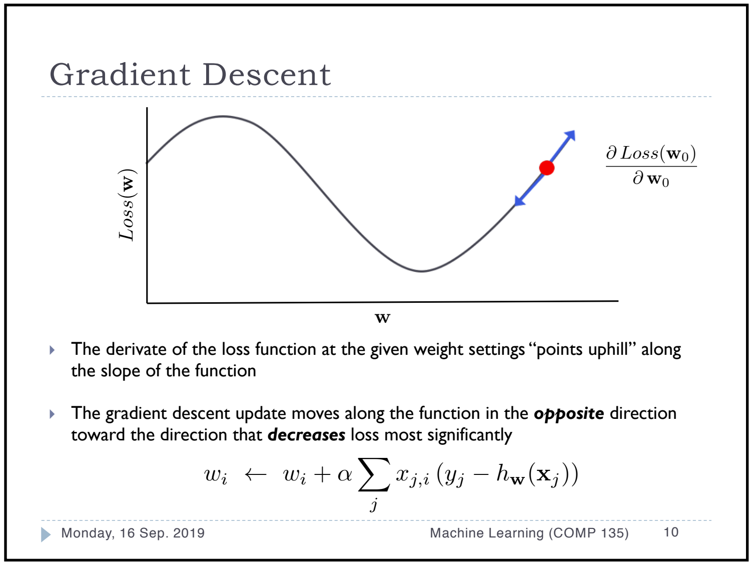


Gradient Descent

$$Loss(\mathbf{w}) = \sum_{j=1}^{N} (y_j - h_{\mathbf{w}}(\mathbf{x}_j))^2$$

- The loss function forms a ***contour*** (here shown for one-dimensional data)
  - For any initial set of weights ($\mathbf{w}_0$) we are at some point on this contour

$Loss(\mathbf{w})$
$Loss(\mathbf{w}_0)$
$\mathbf{w}$

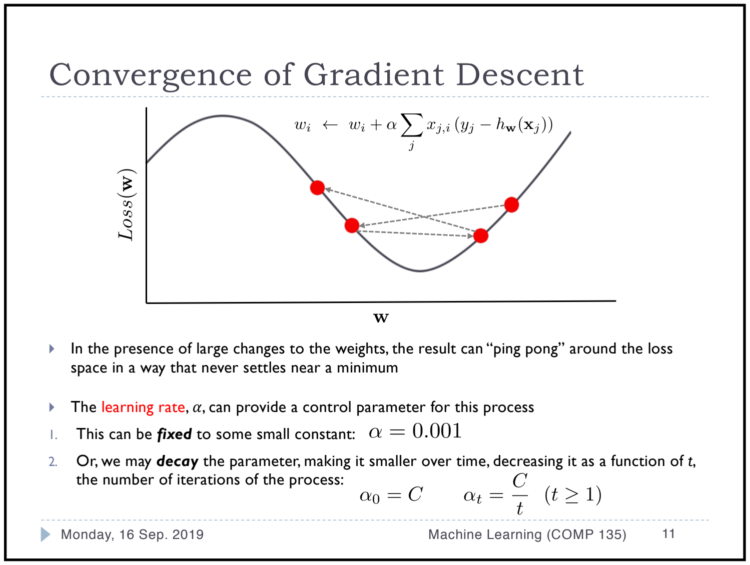


Gradient Descent

$Loss(\mathbf{w})$
$\frac{\partial\, Loss(\mathbf{w}_0)}{\partial\, \mathbf{w}_0}$
$\mathbf{w}$

- The derivate of the loss function at the given weight settings "points uphill" along the slope of the function
- The gradient descent update moves along the function in the ***opposite*** direction toward the direction that ***decreases*** loss most significantly

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}\,(y_j - h_{\mathbf{w}}(\mathbf{x}_j))$$

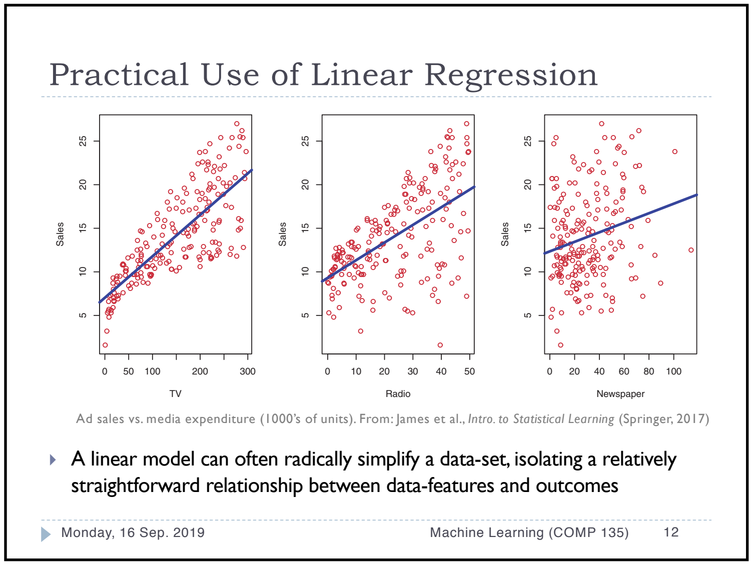


Convergence of Gradient Descent

$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}\,(y_j - h_{\mathbf{w}}(\mathbf{x}_j))$
$Loss(\mathbf{w})$
$\mathbf{w}$

- In the presence of large changes to the weights, the result can "ping pong" around the loss space in a way that never settles near a minimum
- The learning rate, α, can provide a control parameter for this process

1. This can be ***fixed*** to some small constant: $\alpha = 0.001$
2. Or, we may ***decay*** the parameter, making it smaller over time, decreasing it as a function of t, the number of iterations of the process: $\alpha_0 = C \qquad \alpha_t = \frac{C}{t} \quad (t \geq 1)$



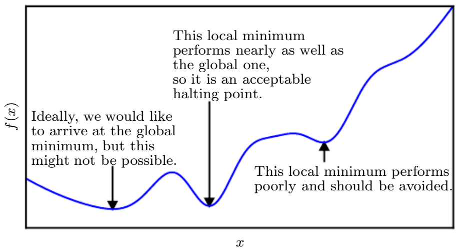
Practical Use of Linear Regression

Sales
5
10
15
20
25
0
50
100
200
300
TV
Radio
0
10
20
30
40
50
Newspaper
0
20
40
60
80
100

Ad sales vs. media expenditure (1000's of units). From: James et al., *Intro. to Statistical Learning* (Springer, 2017)

- A linear model can often radically simplify a data-set, isolating a relatively straightforward relationship between data-features and outcomes

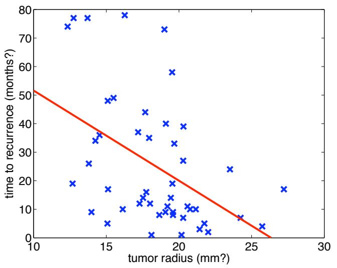
Potential Issues in Gradient Descent



Minima in the loss function. From: Goodfellow, Bengio & Courville., *Deep Learning* (MIT, 2016)

- When loss functions are complex, descending the gradient ***does not*** guarantee optimality
  - Local minima in the loss function are possible
  - Can be dealt with by a variety of techniques, e.g. randomly repeating initial conditions
  - Can often be tolerated, so long as a ***reasonable*** minimum is found

Accuracy of the Hypothesis Function



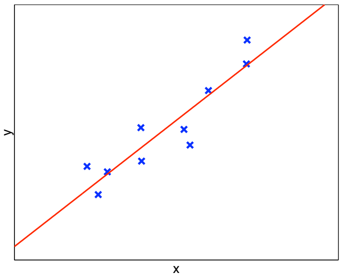
- Although we can generally find the best set of weights efficiently, the exact form of the equation, in terms of the ***degree*** of the polynomial used in that equation, can limit our accuracy
- ***Example***: if we try to predict time to tumor recurrence based on a simple linear function of its radius, this is likely to be very inaccurate

Higher Order Polynomial Regression

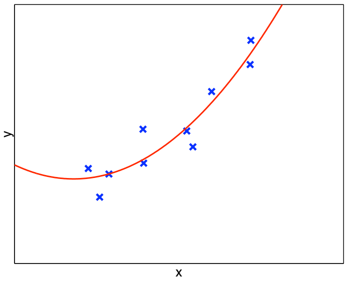
- Since not every data-set is best represented as a simple linear function, we will in general want to explore ***higher-order*** hypothesis functions
- We can still keep these functions quasi-linear, in terms of a sum of weights over terms, but we will allow those terms to take more complex polynomial forms, like:

$$h(x) \longleftarrow y = w_0 + w_1 x + w_2 x^2$$

Higher-Order Regression Solutions

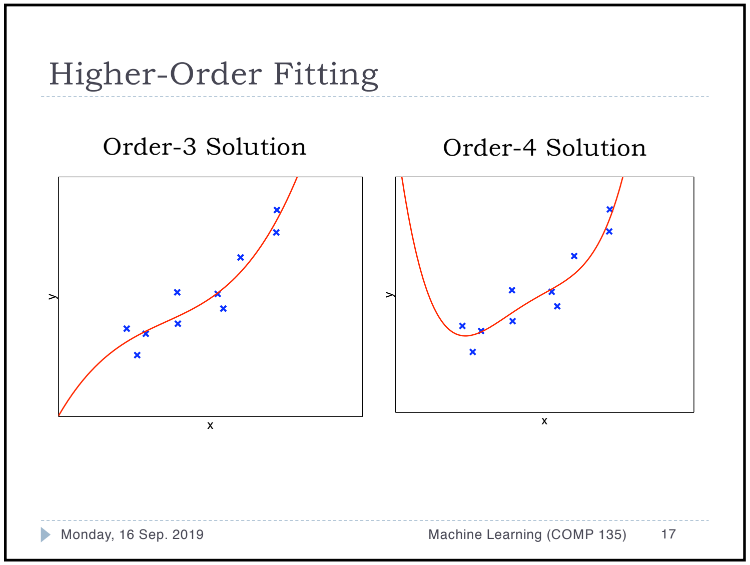


$h(x) \longleftarrow y = 1.05 + 1.60x$



$h(x) \longleftarrow y = 0.73 + 1.74x + 0.68x^2$

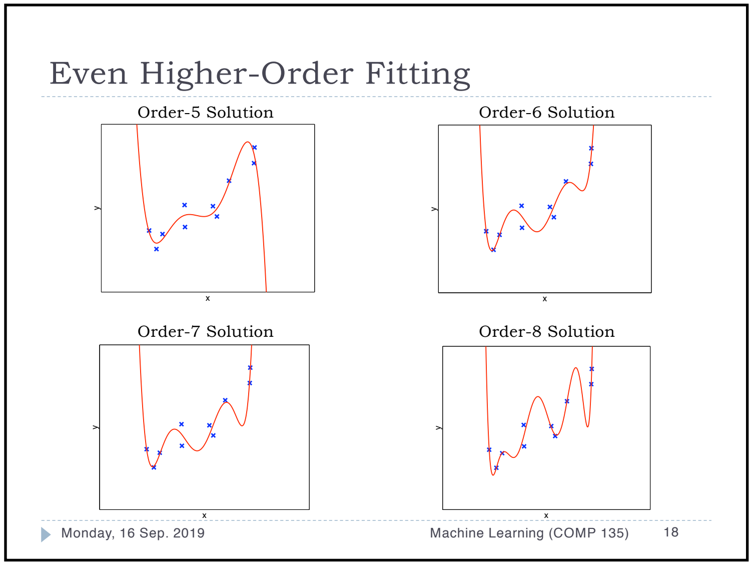
- With an order-2 function, we can fit our data somewhat better than with the original, order-1 version



Higher-Order Fitting

Order-3 Solution

Order-4 Solution



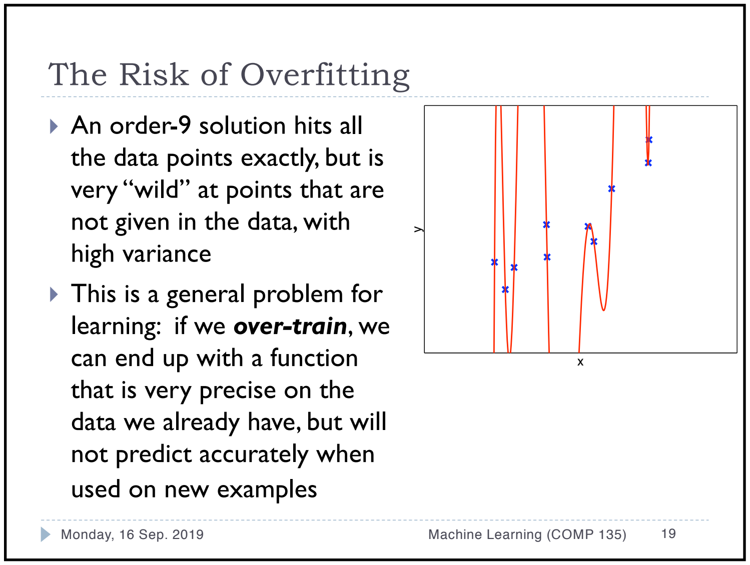
Even Higher-Order Fitting

Order-5 Solution

Order-6 Solution

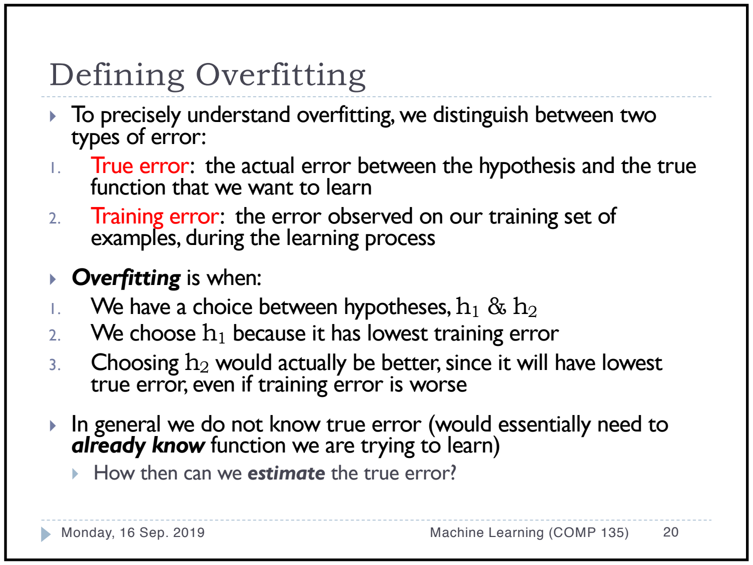
Order-7 Solution

Order-8 Solution



The Risk of Overfitting

- An order-9 solution hits all the data points exactly, but is very “wild” at points that are not given in the data, with high variance
- This is a general problem for learning: if we ***over-train***, we can end up with a function that is very precise on the data we already have, but will not predict accurately when used on new examples



Defining Overfitting

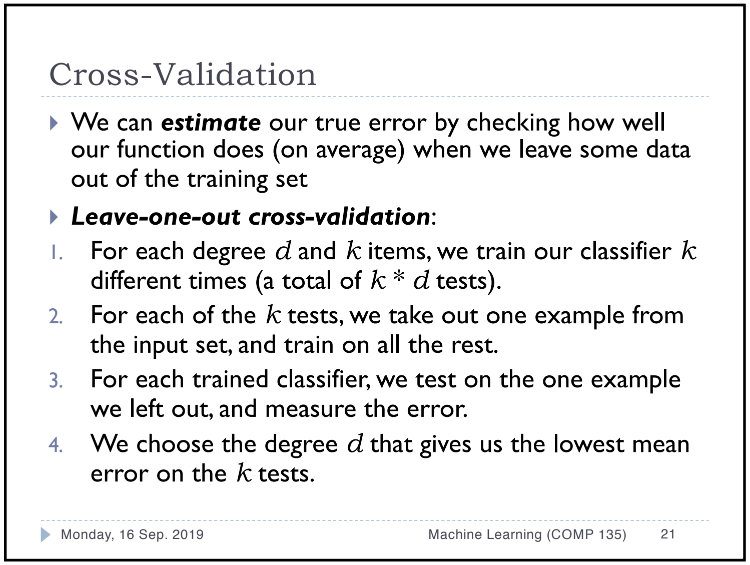
- To precisely understand overfitting, we distinguish between two types of error:

1. True error: the actual error between the hypothesis and the true function that we want to learn
2. Training error: the error observed on our training set of examples, during the learning process

- ***Overfitting*** is when:

1. We have a choice between hypotheses, h_1 & h_2
2. We choose h_1 because it has lowest training error
3. Choosing h_2 would actually be better, since it will have lowest true error, even if training error is worse

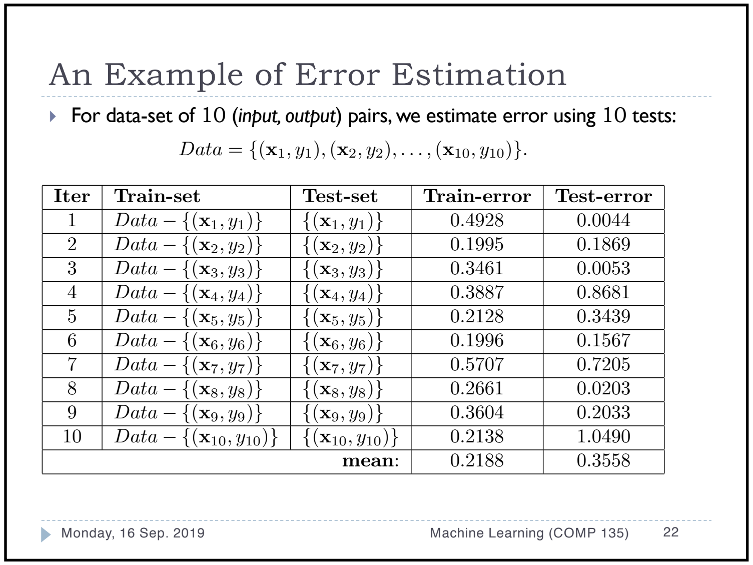
- In general we do not know true error (would essentially need to ***already know*** function we are trying to learn)
  - How then can we ***estimate*** the true error?



Cross-Validation

- We can ***estimate*** our true error by checking how well our function does (on average) when we leave some data out of the training set
- ***Leave-one-out cross-validation***:

1. For each degree d and k items, we train our classifier k different times (a total of $k * d$ tests).
2. For each of the k tests, we take out one example from the input set, and train on all the rest.
3. For each trained classifier, we test on the one example we left out, and measure the error.
4. We choose the degree d that gives us the lowest mean error on the k tests.

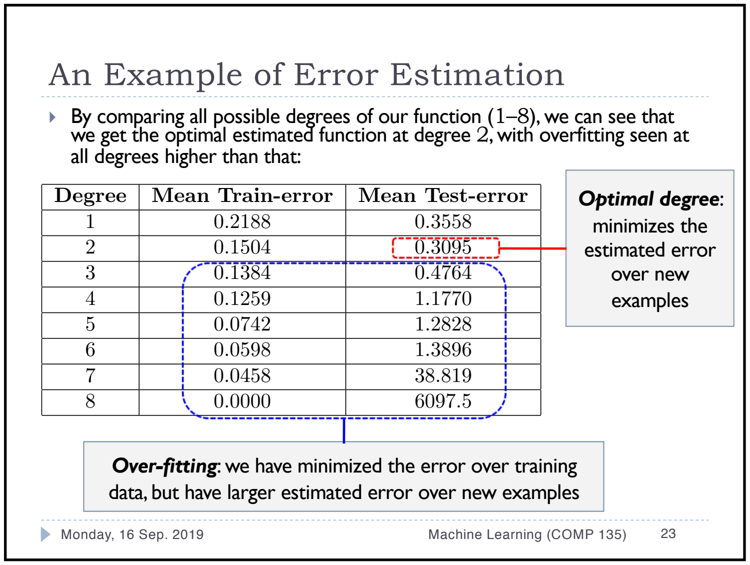


An Example of Error Estimation

- For data-set of 10 (*input, output*) pairs, we estimate error using 10 tests:

$$Data = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_{10}, y_{10})\}.$$

Iter	Train-set	Test-set	Train-error	Test-error
1	$Data - \{(\mathbf{x}_1, y_1)\}$	$\{(\mathbf{x}_1, y_1)\}$	0.4928	0.0044
2	$Data - \{(\mathbf{x}_2, y_2)\}$	$\{(\mathbf{x}_2, y_2)\}$	0.1995	0.1869
3	$Data - \{(\mathbf{x}_3, y_3)\}$	$\{(\mathbf{x}_3, y_3)\}$	0.3461	0.0053
4	$Data - \{(\mathbf{x}_4, y_4)\}$	$\{(\mathbf{x}_4, y_4)\}$	0.3887	0.8681
5	$Data - \{(\mathbf{x}_5, y_5)\}$	$\{(\mathbf{x}_5, y_5)\}$	0.2128	0.3439
6	$Data - \{(\mathbf{x}_6, y_6)\}$	$\{(\mathbf{x}_6, y_6)\}$	0.1996	0.1567
7	$Data - \{(\mathbf{x}_7, y_7)\}$	$\{(\mathbf{x}_7, y_7)\}$	0.5707	0.7205
8	$Data - \{(\mathbf{x}_8, y_8)\}$	$\{(\mathbf{x}_8, y_8)\}$	0.2661	0.0203
9	$Data - \{(\mathbf{x}_9, y_9)\}$	$\{(\mathbf{x}_9, y_9)\}$	0.3604	0.2033
10	$Data - \{(\mathbf{x}_{10}, y_{10})\}$	$\{(\mathbf{x}_{10}, y_{10})\}$	0.2138	1.0490
		mean:	0.2188	0.3558



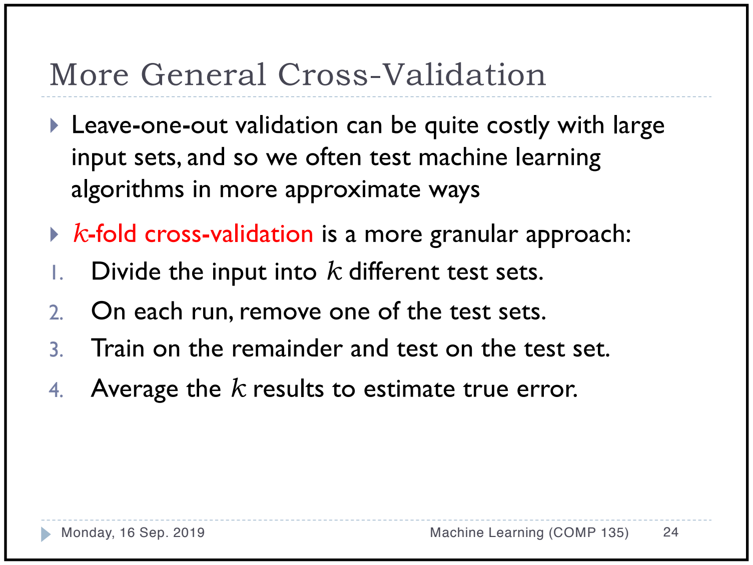
An Example of Error Estimation

- By comparing all possible degrees of our function (1–8), we can see that we get the optimal estimated function at degree 2, with overfitting seen at all degrees higher than that:

Degree	Mean Train-error	Mean Test-error
1	0.2188	0.3558
2	0.1504	0.3095
3	0.1384	0.4764
4	0.1259	1.1770
5	0.0742	1.2828
6	0.0598	1.3896
7	0.0458	38.819
8	0.0000	6097.5

Optimal degree: minimizes the estimated error over new examples

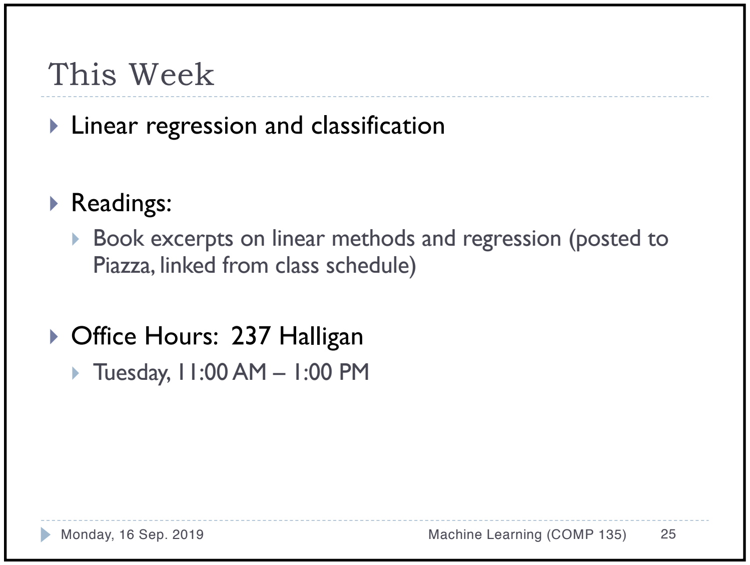
Over-fitting: we have minimized the error over training data, but have larger estimated error over new examples



More General Cross-Validation

- Leave-one-out validation can be quite costly with large input sets, and so we often test machine learning algorithms in more approximate ways
- *k*-fold cross-validation is a more granular approach:

1. Divide the input into k different test sets.
2. On each run, remove one of the test sets.
3. Train on the remainder and test on the test set.
4. Average the k results to estimate true error.



This Week

- Linear regression and classification

- Readings:
  - Book excerpts on linear methods and regression (posted to Piazza, linked from class schedule)

- Office Hours: 237 Halligan
  - Tuesday, 11:00 AM – 1:00 PM