CSE/STAT 416

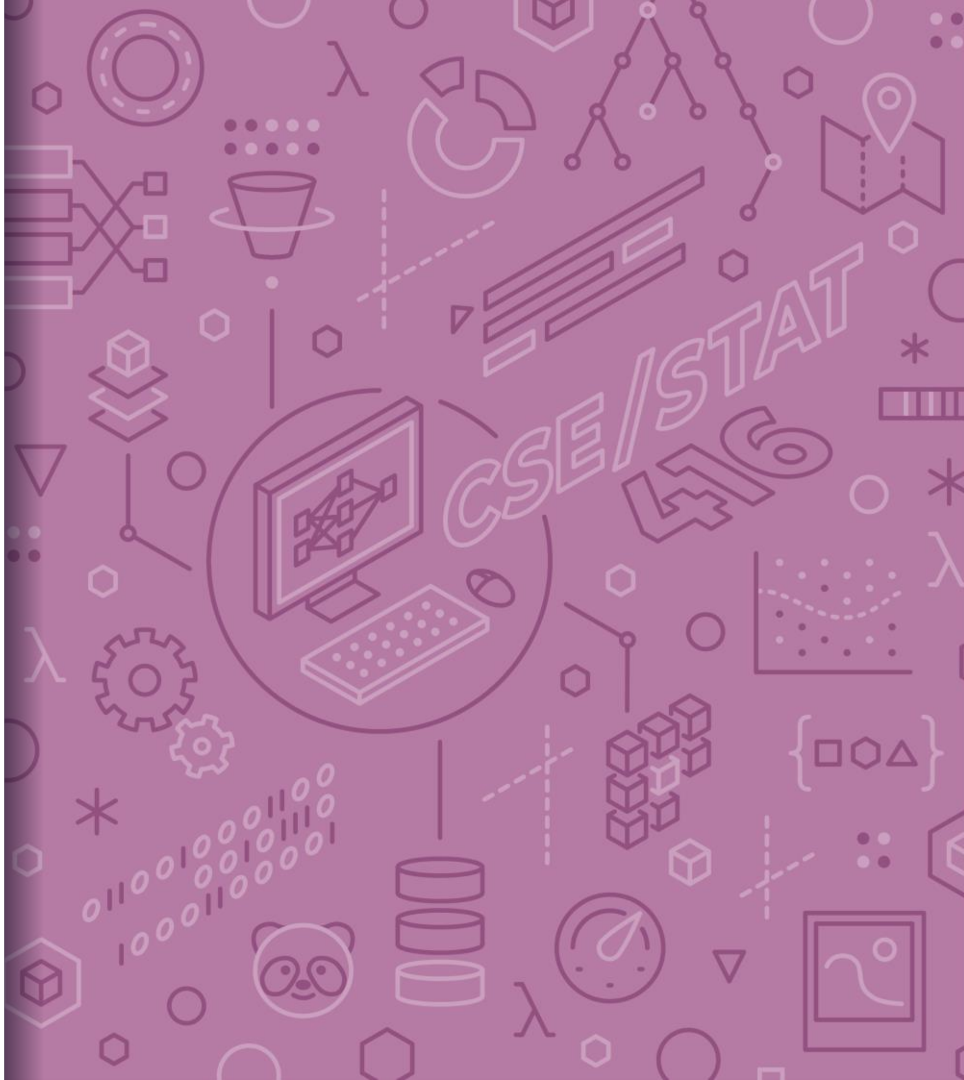
Assessing Performance

Tanmay Shah
University of Washington
March 29, 2024

Questions? Raise hand or **sli.do #cs416**

Before Class: Did you get to do anything fun during the (short) spring break?

Listening to: [grentperez]



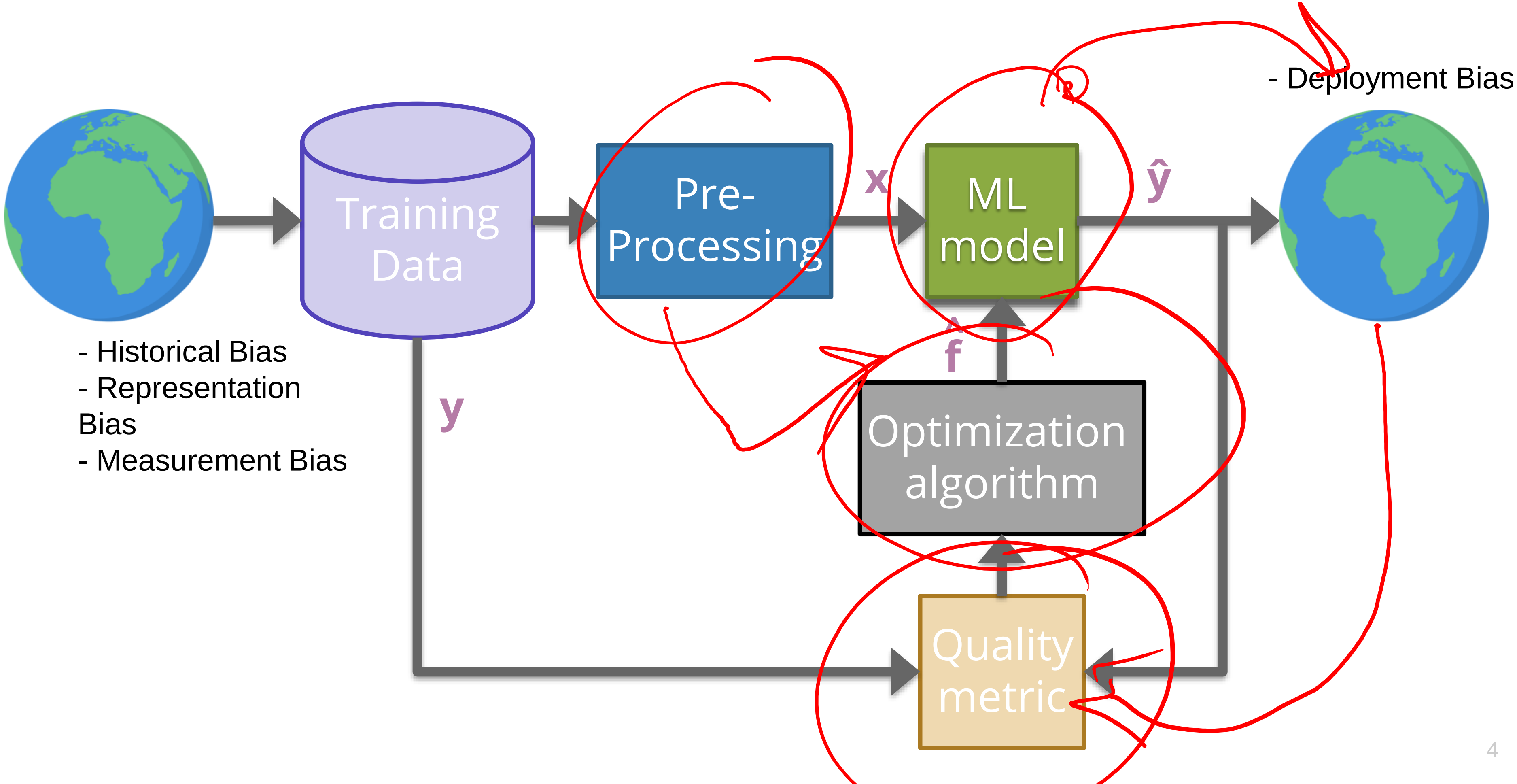
Logistics

- Check EdStem for announcements and clarifications
- Section tomorrow will focus on writing code for HW1
- Module 0 assignments out!
  - Learning Reflection due Monday
  - HW0 released and due Thursday night

Pre-Class Video 1

Feature Extraction

ML Pipeline



- Historical Bias
- Representation Bias
- Measurement Bias

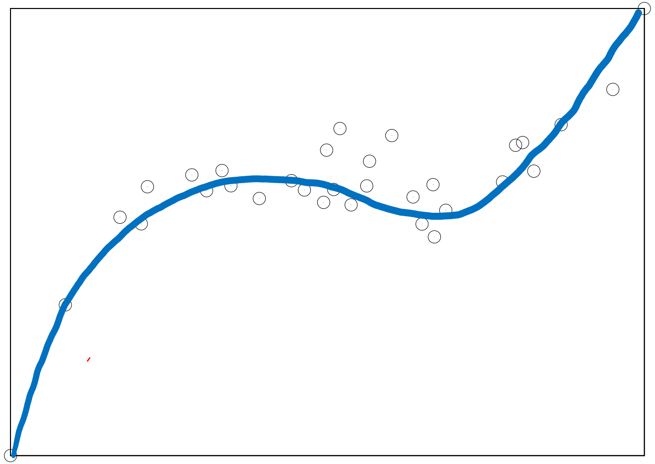
- Deployment Bias

Higher Order Features

This data doesn't look exactly linear, why are we fitting a line instead of some higher-degree polynomial?

We can! We just have to use a slightly different model!

$$y_i = \underline{w_0 + w_1 x_i + w_2 x_i^2 + w_3 x_i^3 + \epsilon_i}$$



Polynomial Regression

Model

$$y_i = w_0 + w_1 x_i + w_2 x_i + \ldots + w_p x_i^p + \epsilon_i$$

Just like linear regression, but uses more features!

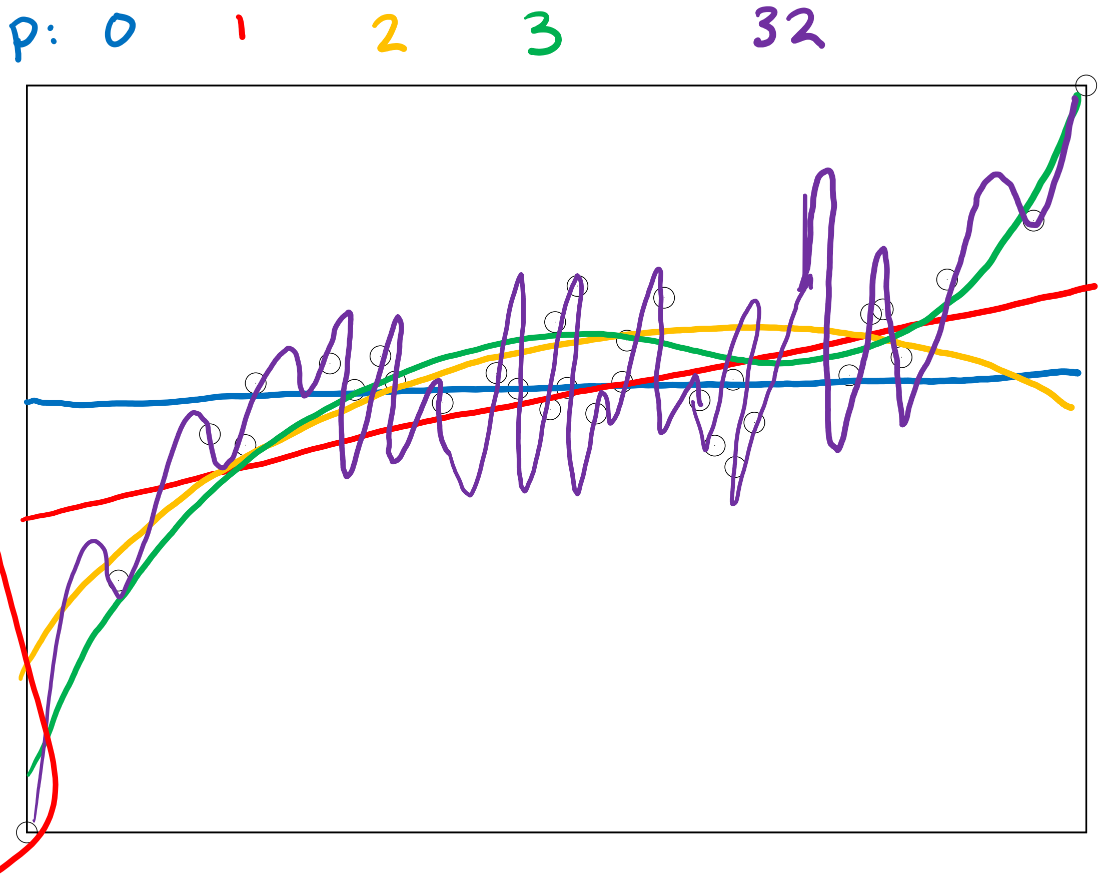
	Feature	Value	Parameter
1	0	1 (constant)	w_0
p	1	x	w_1
	2	x^2	w_2
	...	…	…
	p	x^p	w_p

How do you train it? Gradient descent (with more parameters)

Polynomial Regression

$$y_1 = w_1 x_1 + w_2 x_2^2 + \dots + w_d x_d^d$$

$$1 \, w_0$$



How to decide what the right degree? Come back Wednesday!

Features

$h_j(x_i)$

$h_j(x_i)$ = feature j for example :

Features are the values we select or compute from the data inputs to put into our model. **Feature extraction** is the process of turning the data into features.

Model

$$y_i = w_0 h_0(x_i) + w_1 h_1(x_i) + \ldots + w_D h_D(x_i) + \epsilon_i$$

$$= \sum_{j=0}^{D} w_j h_j(x_i) + \epsilon_i$$

Feature	Value	Parameter
0	$h_0(x)$ often 1 (constant)	w_0
1	$h_1(x) = x_i$	w_1
2	$h_2(x) = x_i^2$	w_2
…	h_j … $= x_i^3 \cdot \log(x_i)$	…
D	$h_D(x) = e^{x_i}$	w_D

Adding Other Inputs

log(sqft)

$h(x_i)$

Generally we are given a data table of values we might look at that include more than one value per house.

- Each row is a single house.
- Each column (except Value) is a data input.

w_0

sq. ft.	# bathrooms	owner's age	…	value
1400	3	47	…	70,800
700	3	19	…	65,000
…	…	…	…	…
1250	2	36	…	100,000

log(x_0)

$h_1(x_i) = x_0$

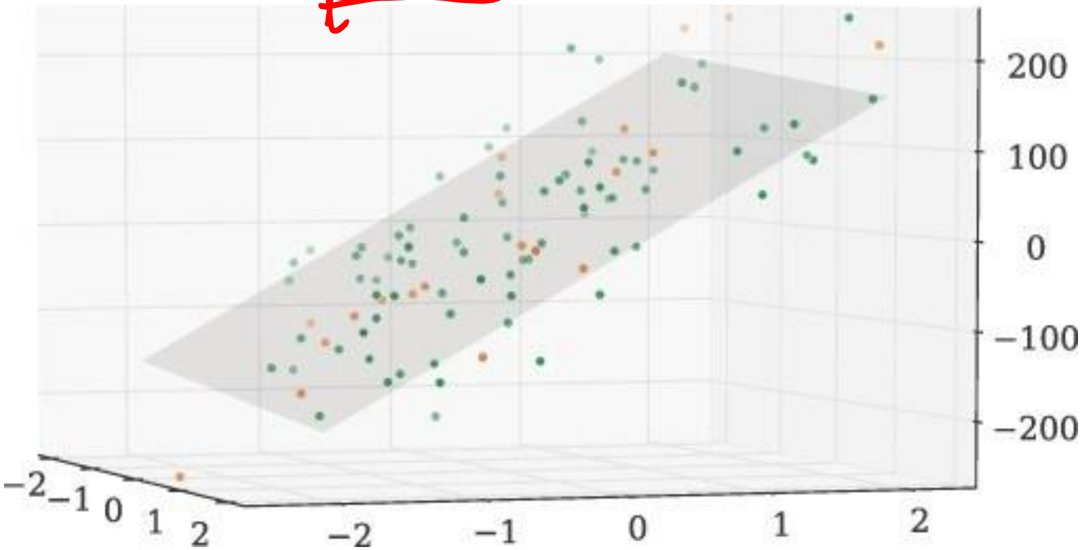
More Inputs - Visually

$w_1 \times 2 \leftarrow w_1$

sq·2

Adding more features to the model allows for more complex relationships to be learned

$$y_i = \underline{w_0} + \boxed{w_1(\underline{sq.ft.})} + w_2(\underline{\# bathrooms}) + \epsilon_i$$



Coefficients tell us the rate of change **if all other features are constant**

Notation

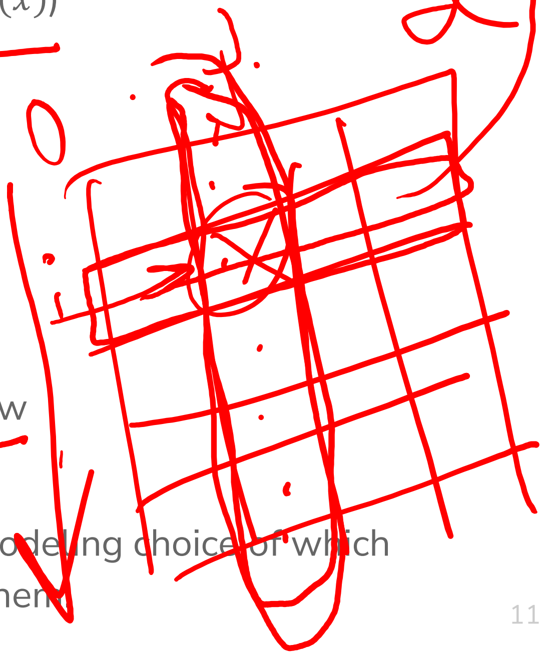
Important: Distinction is the difference between a *data input* and a *feature.*

- Data inputs are columns of the raw data
- Features are the values (possibly transformed) for the model (done after our feature extraction $h(x)$)

Data Input: $x_i = (x_i[1], x_i[2], \dots, x_i[d])$

Output: y_i

- x_i is the i^{th} row
- $x_i[j]$ is the i^{th} row's j^{th} data input
- $h_j(x_i)$ is the j^{th} feature of the i^{th} row



This makes explicit, an often-implicit modeling choice of which features to use and how to transform them.

Features

You can use anything you want as features and include as many of them as you want!

Generally, more features means a more complex model. This might not always be a good thing!

Choosing good features is a bit of an art.

Feature	Value	Parameter
0	1 (constant)	w_0
1	$h_1(x)$ … $x[1]$ = sq. ft.	w_1
2	$h_2(x)$ … $x[2]$ = # bath	w_2
…	…	…
D	$h_D(x)$ … like $\log(x[7]) * x[2]$	w_{D}

Linear Regression Recap

Notation: $\hat{w}^T h(x) = \sum_{j=0}^{D} \hat{w}_j h_j(x) = \hat{y}$

Shorthand

Dataset

$\{(x_i, y_i)\}_{i=1}^{n}$ where $x \in \mathbb{R}^d$, $y \in \mathbb{R}$

Feature Extraction

$h(x): \mathbb{R}^d \to \mathbb{R}^D$

$h(x) = (h_0(x), h_1(x), \ldots, h_D(x))$

Regression Model

$$y = f(x) + \epsilon$$
$$= \sum_{j=0}^{D} w_j h_j(x) + \epsilon$$
$$= w^T h(x) + \epsilon$$

Quality Metric

$$MSE(w) = \frac{1}{n} \sum_{i=1}^{n} (y_i - w^T x_i)^2$$

Predictor

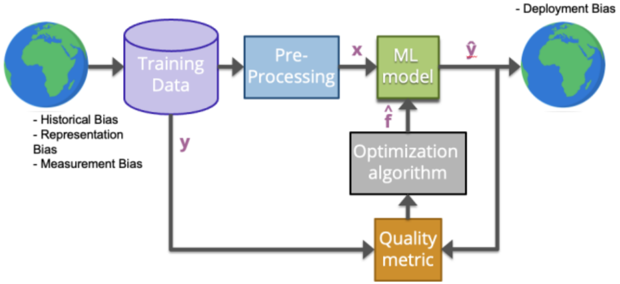
$$\hat{w} = \underset{w}{\operatorname{argmin}}\, MSE(w)$$

ML Algorithm

Optimized using Gradient Descent

Prediction

$$\hat{y} = \hat{w}^T h(x)$$

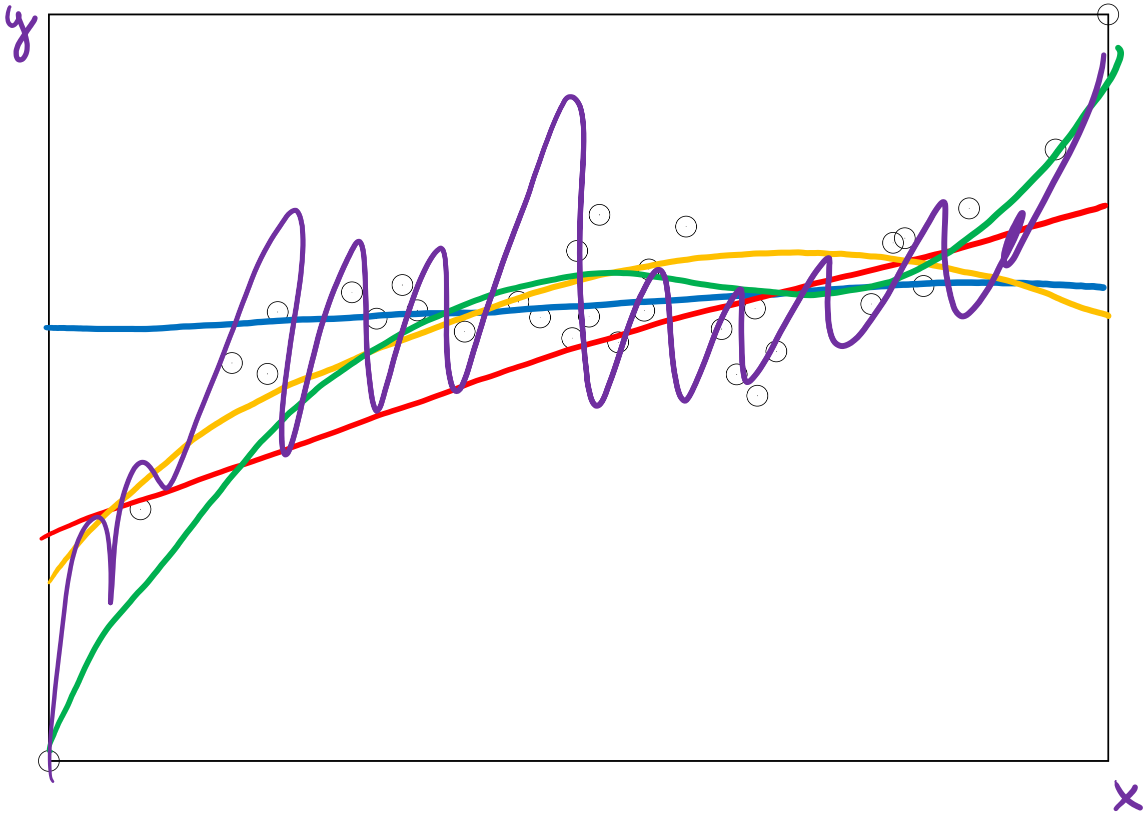


Brain Break

Pre-Class Video 2

Assessing Performance

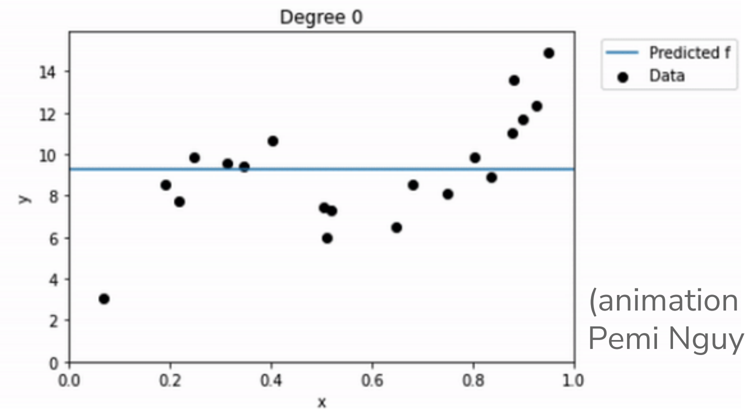
Polynomial Regression



How do we decide what the right choice of p is?

Polynomial Regression

Consider using different degree polynomials on the same training set.



(animation by Pemi Nguyen)

From estimating with your eyes, which one seems to have the lowest MSE on this dataset?

It seems like minimizing the MSE on the training set is not the whole story here …

Performance

Why do we train ML models?

We generally want them to do well on **unseen** data.

If we choose the model that minimizes MSE on the data it learned from, we are just choosing the model that can **memorize**, not the one that **generalizes** well.

- **Analogy:** Just because you can get 100% on a practice exam you've studied for hours, it doesn't mean you will also get 100% on the real test that you haven't seen before.

Key Idea: Assessing yourself based on something you *learned from* generally overestimates how well you will do in the future!

$\hat{f} = f_{\hat{w}}$

Future Performance

$(y_i - \hat{y}_i)^2$

Generalized Loss Function

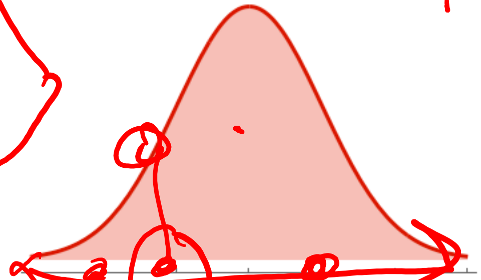
$L(y, \boxed{f_{\hat{w}}(x)})$ ← MSE, MAE

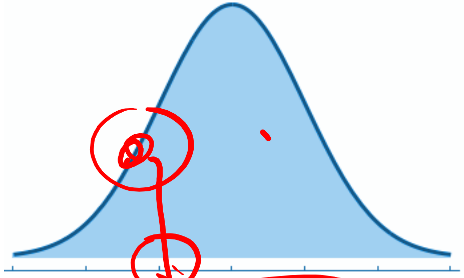
What we care about is how well the model will do on unseen data.

How do we measure this? **True error**

To do this, we need to understand uncertainty in the world

Reg. Model: $y_i = f(x_i) + \varepsilon_i$


Sq. Ft.


Price | Sq. Ft.

PDF

True Error

$$\mathbb{E}_{(x,y)}\left[L(y, f_{\hat{w}}(x))\right] = \sum_{x} \sum_{y} L(y, f_{\hat{w}}(x))\, p(x,y)$$

↑ all possible (x,y) pairs

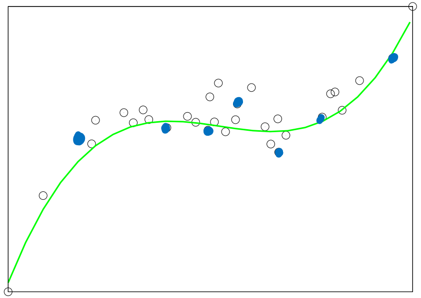
Model Assessment

How can we figure out how well a model will do on future data if we don't have any future data?

- Estimate it! We can hide data from the model to test it later as an estimate how it will do on future data

We will randomly split our dataset into a **train set** and a **test set**

- The train set is to train the model
- The test set is to estimate the performance in the future



Test Error

What we really care about is the **true error,** or how well a model perform on unseen data in the wild, but we can't know that without having an infinite amount of data!

We will use the **test set** to estimate the true error.

↑ never trained on

Note: The train and test set need to be **randomly split** in order for the test set to be truly reflective of data in the real world.

Call the error on the test set the **test error** for a model $\hat{f}$:

$$MSE_{test} = \frac{1}{n}\sum_{i \in Test}\left(y^{(i)} - \hat{f}\left(x^{(i)}\right)\right)^2$$

n of test set

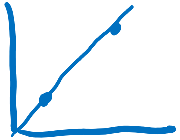
If the test set is large enough, this can approximate the true error.

Train/Test Split

If we use the test set to estimate future, how big should it be?

More test data => better estimate of true error

This comes at a cost of reducing the size of the training set though (in the absence of being able to just get more data)



Small train set => poor model

In practice people generally do train:test as either

Tr Test

- 80:20
- 90:10

Important: Never train your model on data in the test set!

Regression Recap

Linear Regression Model

Assume we have a simple model with ***one feature***, where we establish a linear relationship between ***the area of a house*** i and ***its price***:

$$y_i = w_0 + w_1 x_i + \epsilon_i$$

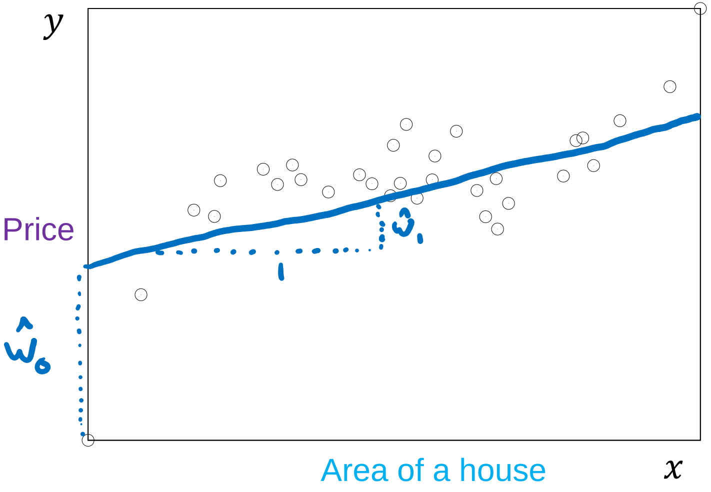
w_0, w_1 are the **parameters** of our model that need to be learned

- w_0 is the intercept / **bias**, representing the starting price of a house
- w_1 is the slope / **weight** associated with **feature** ”area of a house”

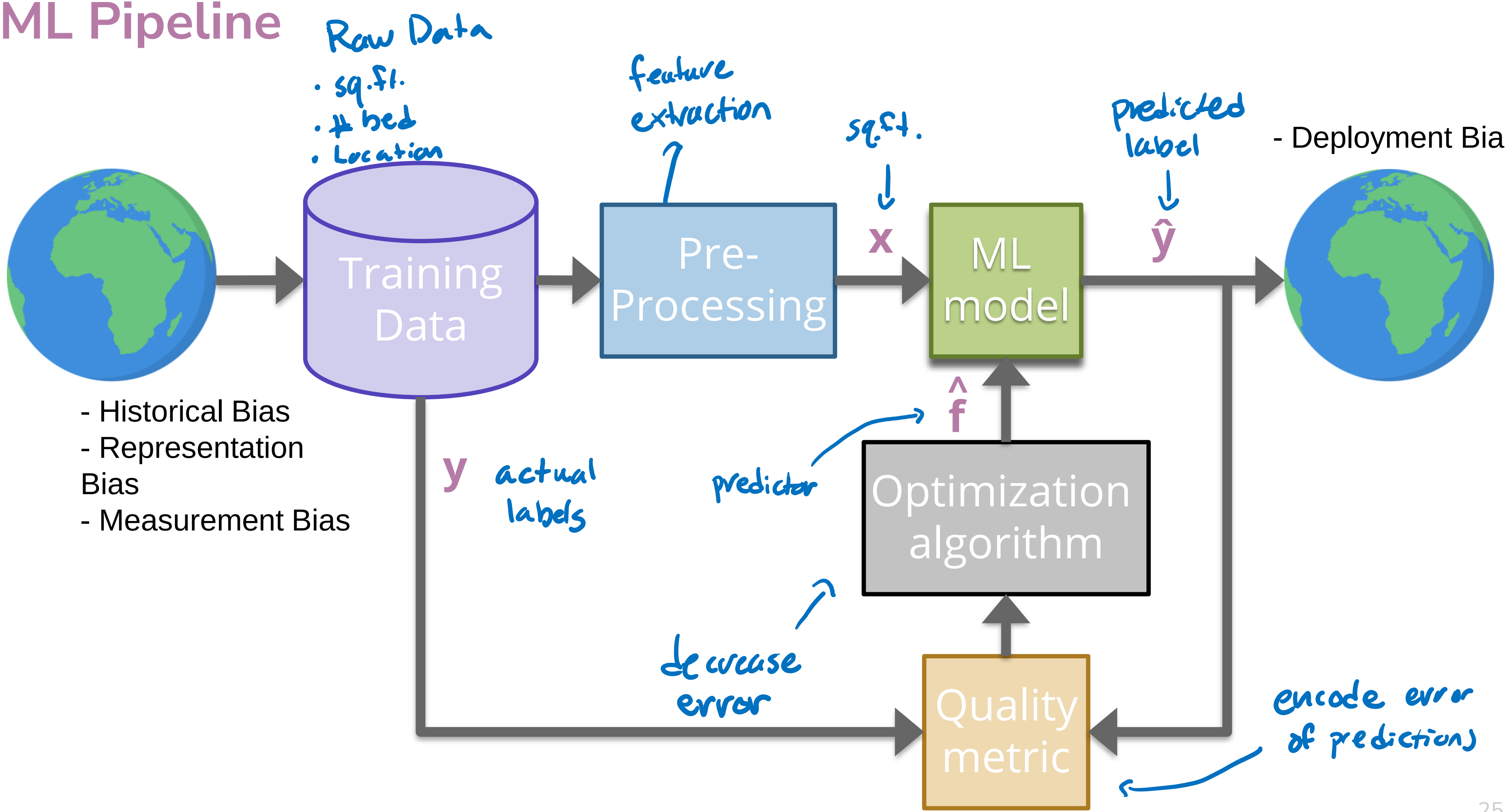
Learn estimates of these parameters $\hat{w}_1$, $\hat{w}_0$ and use them to predict new value for any input x!

$$\hat{y} = \hat{w}_1 x + \hat{w}_0$$

Why don’t we add ϵ?



ML Pipeline

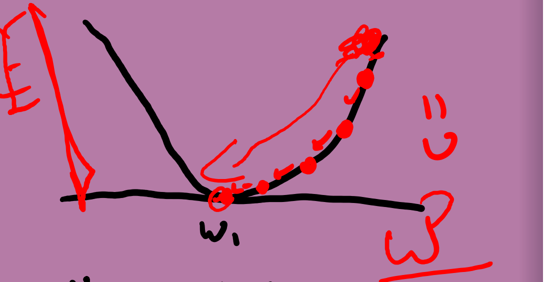


- Deployment Bias

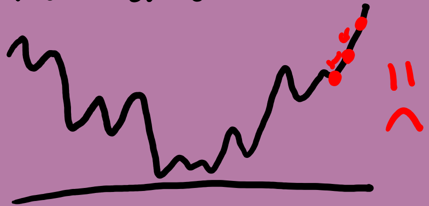
- Historical Bias
- Representation Bias
- Measurement Bias

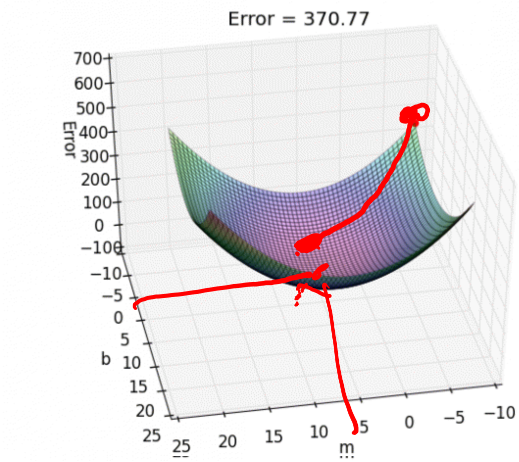
Gradient Descent

Convex

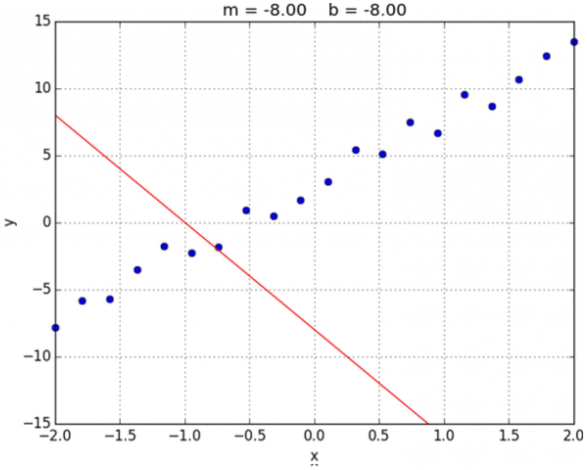


Non-convex





Only works if convex!



Instead of computing all possible points to find the minimum, just start at one point and “roll” down the hill. Use the gradient (slope) to determine which direction is down.

Start at some (random) weights w

While we haven’t converged:

$$w^{(t+1)} = w^{(t)} - \alpha \boxed{\nabla L(w)}$$

- α: learning rate

gradient = fancy word for “slope”

- - $\nabla L(w)$: the gradients of loss function L on a set of weights w

Pre-Class 1 Recap: Features

You can use anything you want as features and include as many of them as you want!

Generally, more features means a more complex model. This might not always be a good thing!

Choosing good features is a bit of an art.

Feature	Value	Parameter
0	1 (constant)	w_0
1	$h_1(x)$ … $x[1]$ = sq. ft.	w_1
2	$h_2(x)$ … $x[2]$ = # bath	w_2
…	…	…
D	$h_D(x)$ … like $\log(x[7]) * x[2]$	w_D

Linear Regression Recap

Dataset

$\{(x_i, y_i)\}_{i=1}^{n}$ where $x \in \mathbb{R}^d$, $y \in \mathbb{R}$

Feature Extraction

$h(x)\colon\ \mathbb{R}^d \to \mathbb{R}^D$

$h(x) = (h_0(x), h_1(x), \dots, h_D(x))$

Regression Model

$$\begin{aligned} y &= f(x) + \epsilon \\ &= \sum_{j=0}^{D} w_j h_j(x) + \epsilon \\ &= w^T h(x) + \epsilon \end{aligned}$$

Quality Metric

$$MSE(w) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - w^T x_i\right)^2$$

Predictor

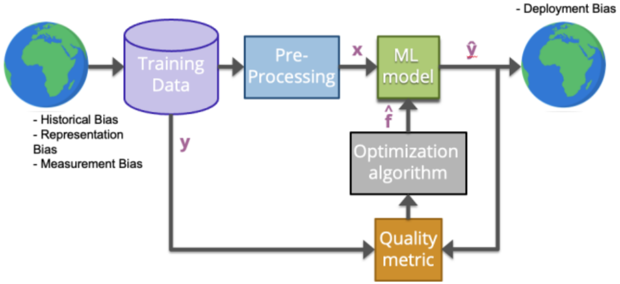
$$\hat{w} = \underset{w}{\operatorname{argmin}}\, MSE(w)$$

ML Algorithm

Optimized using Gradient Descent

Prediction

$$\hat{y} = \hat{w}^T h(x)$$



Term recap

- **Supervised learning**: The machine learning task of learning a function that maps an input to an output based on example input-output pairs.
- **Regression**: A supervised learning task where the outputs are continuous values.
- **Feature**:
  - An attribute that we're selecting for our model
  - Can come from the original dataset, or through some transformations (***feature extraction***)
- **Parameter**: The weight or bias associated with a feature. The goal of machine learning is to adjust the weights to optimize the loss functions on training data.
- **Loss function**: A function that computes the distance between the predicted output from a machine learning model and the actual output.
- **Machine learning model**: An algorithm that combs through an amount of data to find patterns, make predictions, or generate insights
- **Optimization algorithm**: An algorithm used to minimize the loss during training. The most common one is ***Gradient Descent***.

Pre-Class 2 Recap: Model Evaluation

- Low training error != a good model
- To avoid memorizing, need to test on data we didn’t train on
- **Training set** to train on and a **test set** for evaluation
  - Test set is a stand-in for all future data

slido

sli.do #cs416

- **Goal**: Get you actively participating in your learning
- Typical Activity
  - Question is posed
  - **Think** (1 min): Think about the question on your own
  - **Pair** (2 min): Talk with your neighbor to discuss question
    - If you arrive at different conclusions, discuss your logic and figure out why you differ!
    - If you arrived at the same conclusion, discuss why the other answers might be wrong!
  - **Share** (1 min): We discuss the conclusions as a class
- During each of the **Think** and **Pair** stages, you will respond to the question via a sli.do poll
  - Not worth any points, just here to help you learn!

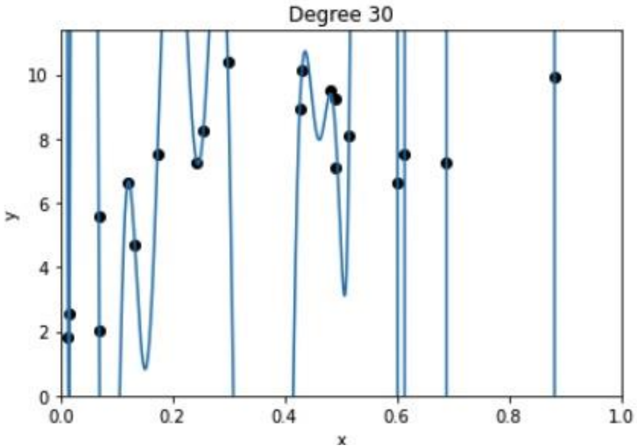
slido

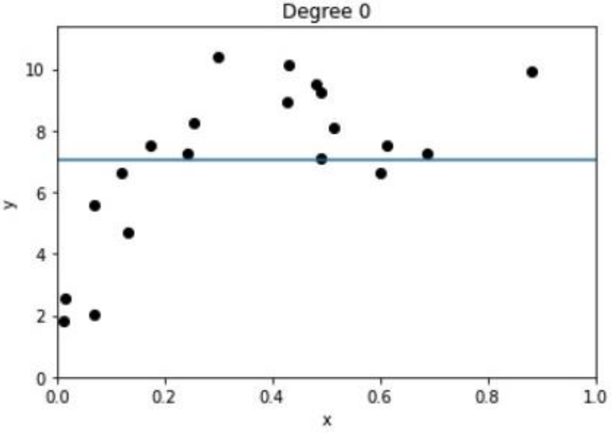
Think

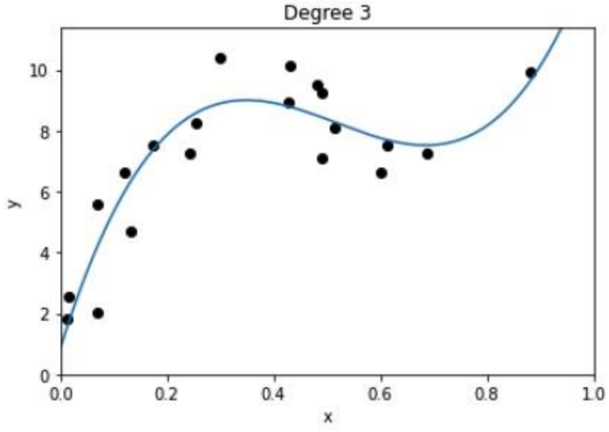
1 minute

Which of the models do you expect to have the:

- Highest Train Error
- Highest Test Error
- Lowest Train Error
- Lowest Test Error







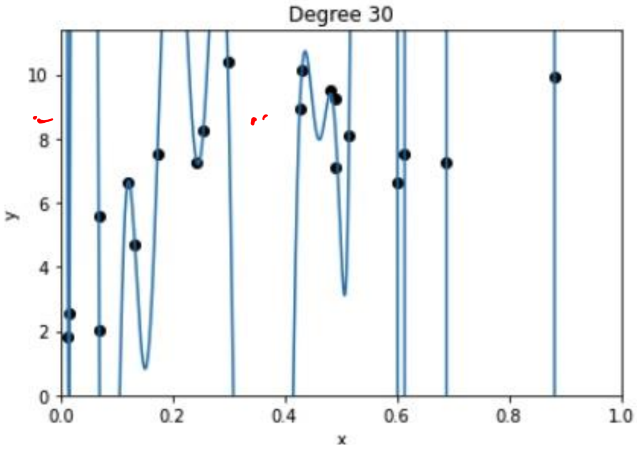
slido

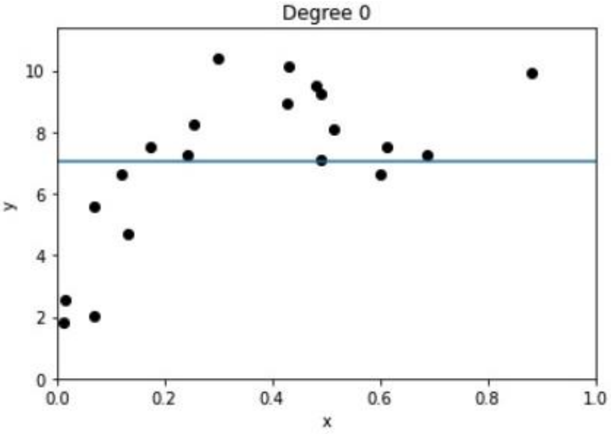
Group

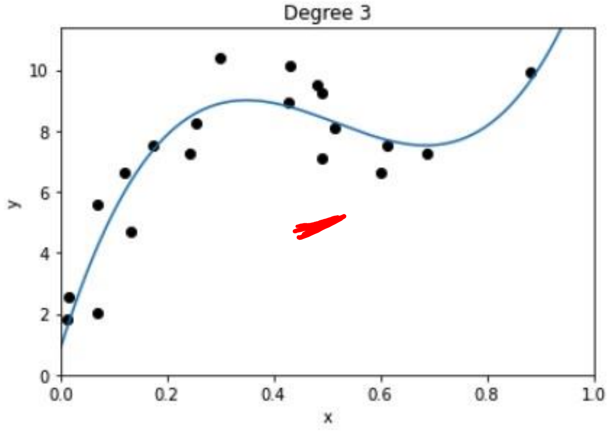
2 minute

Which of the models do you expect to have the:

- Highest Train Error
- Highest Test Error
- Lowest Train Error
- Lowest Test Error







Model Complexity

Model Complexity

- There is not a well-defined way to measure the complexity of a model. It depends on the nature of the models.

- We usually associate it with the number of parameters. A model with more parameters is usually more complex.

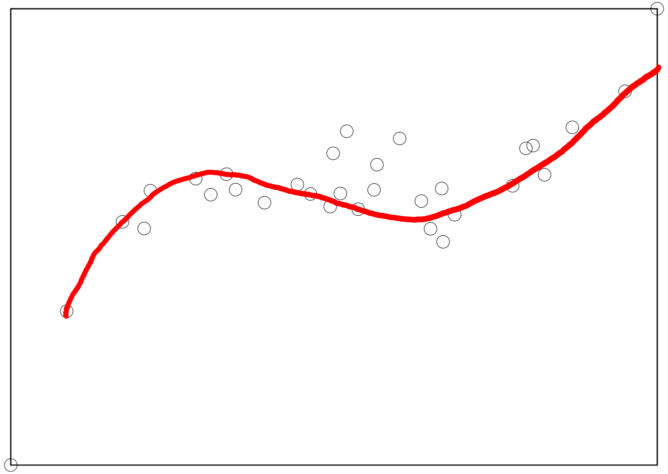
- Example with polynomial regression:
  - Model 1: (2 parameters)
    - $y = \underline{w_0} + \underline{w_1}x$
  - Model 2: (4 parameters)
    - $y = w_0 + w_1x + \underline{w_2}x^2 + \underline{w_3}x^3$

  We say that model 2 is more complex than model 1.

Training Error

What happens to **training error** as we increase model complexity?

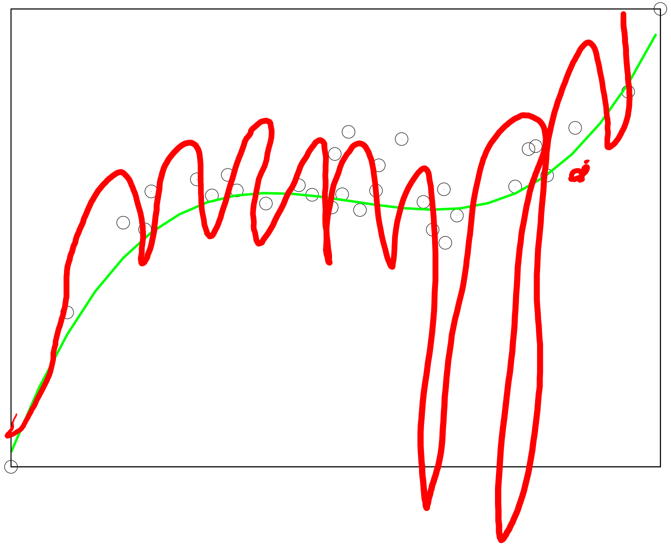
- Start with the simplest model (a constant function)
- End with a very high degree polynomial



True Error

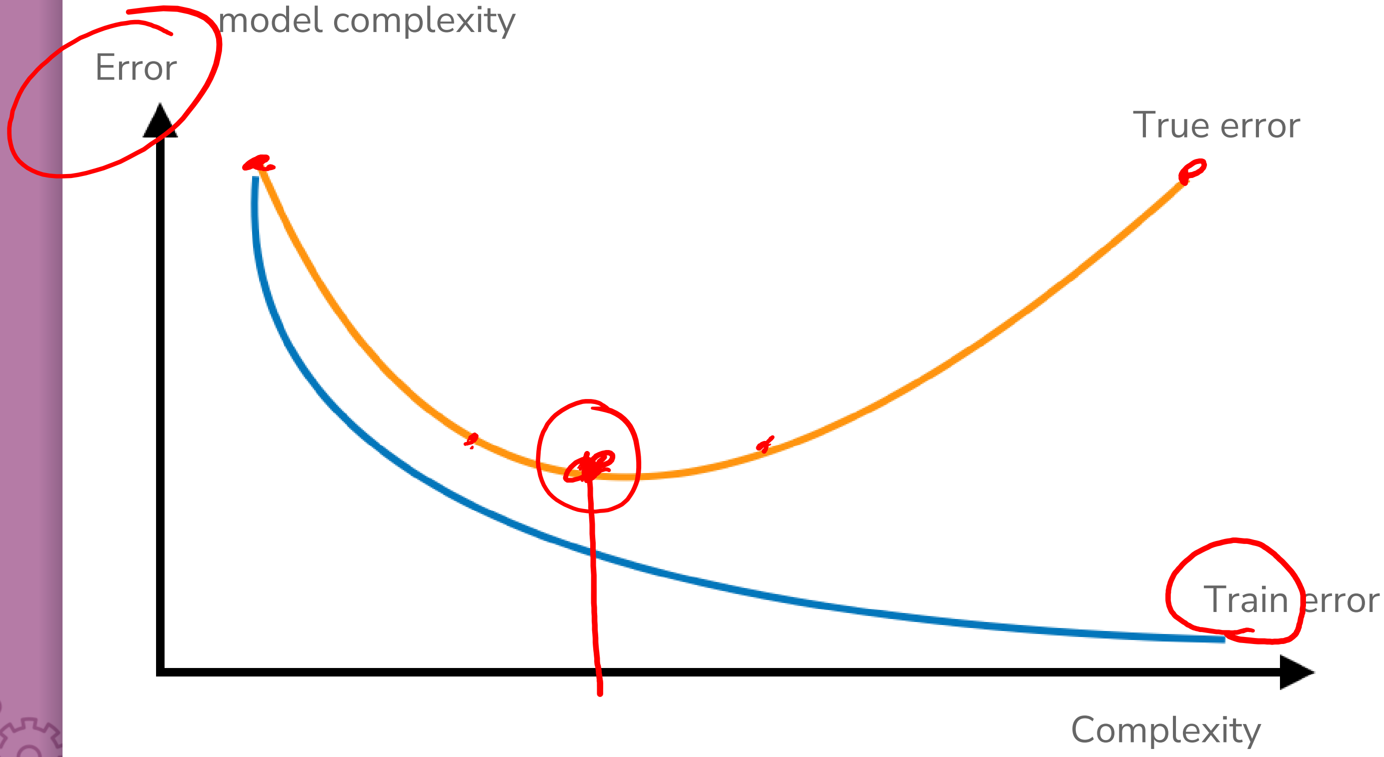
What happens to **true error** as we increase model complexity?

- Start with the simplest model (a constant function)
- End with a very high degree polynomial



Train/True Error

Compare what happens to train and true error as a function of model complexity

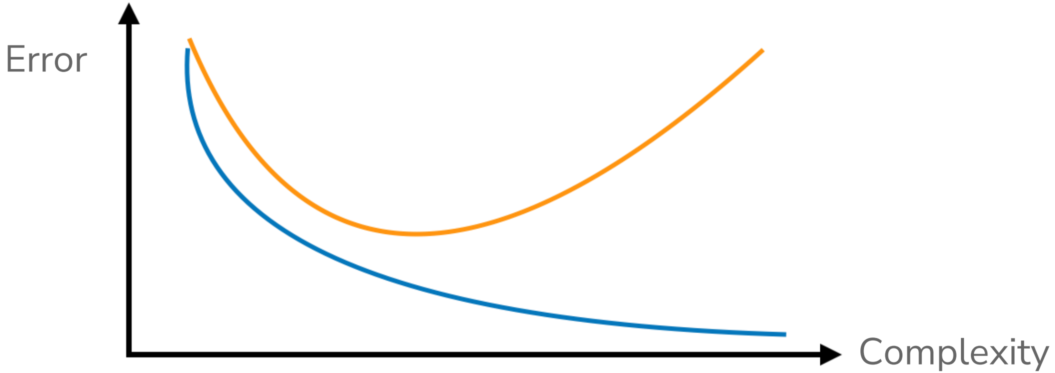


Overfitting

Overfitting happens when we too closely match the training data and fail to generalize.

Overfitting occurs when you train a predictor $\hat{w}$ but there exists another predictor w' from the same model class such that:

- $error_{true}(w') < error_{true}(\hat{w})$
- $error_{train}(w') > error_{train}(\hat{w})$



slido

Think

1 min

sli.do #cs416

Consider the learning task of predicting the price of a house based on its features. **Evaluate the statement: “To make the model more accurate, we should include as many features as possible (e.g., square footage, # bathrooms, location, etc.).**

- True
- False
- Unsure

slido

Group

1 min

sli.do #cs416

Consider the learning task of predicting the price of a house based on its features. **Evaluate the statement: “To make the model more accurate, we should include as many features as possible (e.g., square footage, # bathrooms, location, etc.).**

- True
- False
- Unsure

Brain Break

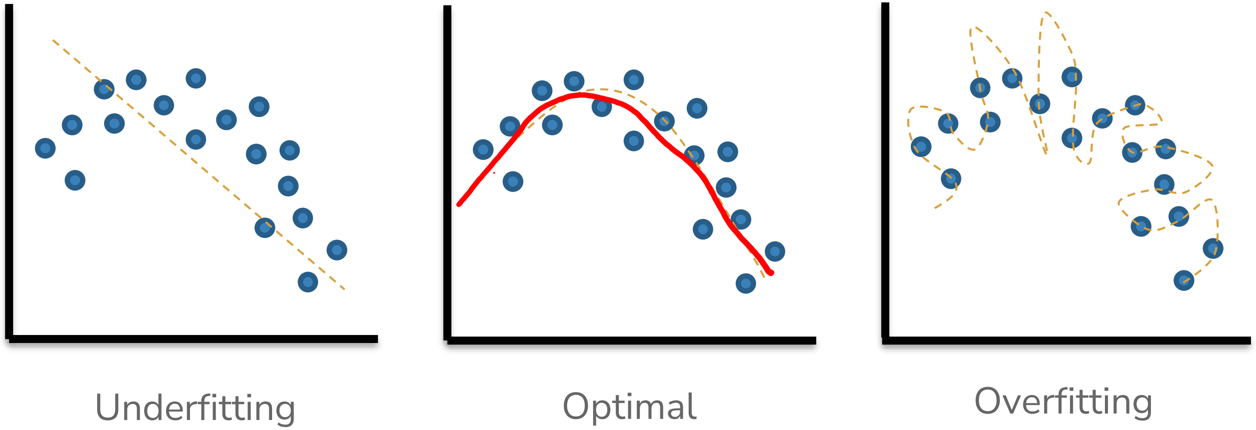
Bias-Variance Tradeoff

Underfitting / Overfitting

The ability to overfit/underfit is a knob we can turn based on the model complexity.

- More complex => easier to overfit
- Less complex => easier to underfit

In a bit, we will talk about how to chose the “just right”, but now we want to look at this phenomena of overfitting/underfitting from another perspective.



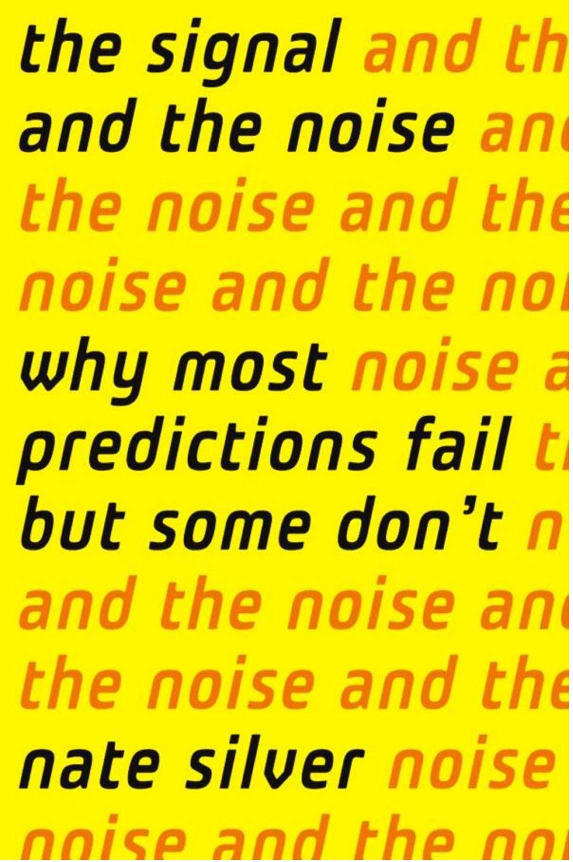
Signal vs. Noise

Learning from data relies on balancing two aspects of our data

- **Signal**
- **Noise**

Complex models make it easier to fit too closely to the noise

Simple models have trouble picking up the signal



the signal and th
and the noise an
the noise and the
noise and the no
why most noise a
predictions fail t
but some don't n
and the noise an
the noise and the
nate silver noise
noise and the no

Source of errors in a model

Total errors for a machine learning model comes from 3 types:

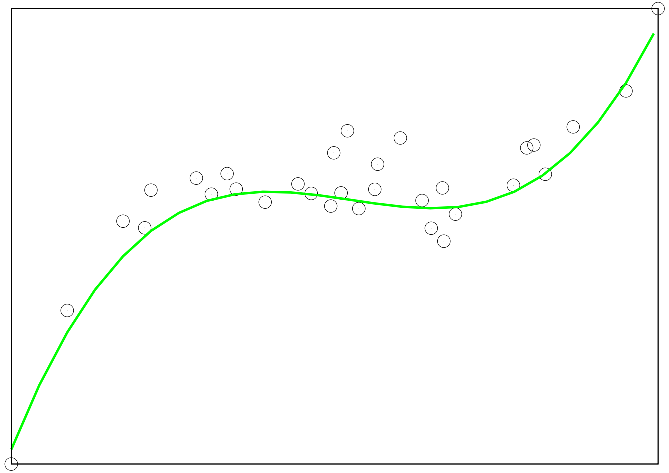
- **Bias**
- **Variance**
- **Irreducible Error**

Irreducible error is the one that we can’t avoid or possibly eliminate. They are caused by elements outside of our control, such as noise from observations.

Bias

A model that is too simple fails to fit the signal. In some sense, this signifies a fundamental limitation of the model we are using to fail to fit the signal. We call this type of error **bias**.

Bias is the difference between the average prediction of our model **and** the expected value which we are trying to predict.

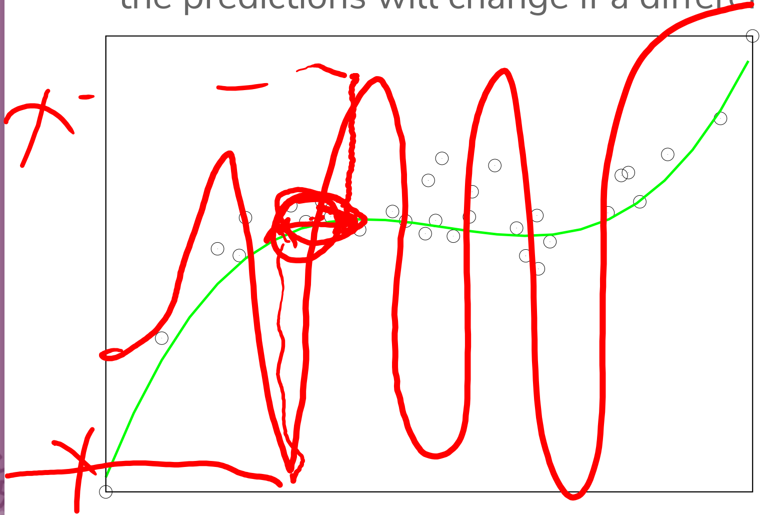


Low complexity (simple) models tend to have high bias.

Variance

A model that is too complicated for the task overly fits to small fluctuations. The flexibility of the complicated model makes it capable of memorizing answers rather than learning general patterns. This contributes to the error as **variance**.

Variance is the variability in the model prediction, meaning how much the predictions will change if a different training dataset is used.



High complexity models tend to have high variance.

Bias-Variance Tradeoff

Tradeoff between bias and variance:

- Simple models: High bias + Low variance
- Complex models: Low bias + High variance

Source of errors for a particular model $\hat{f}$ using MSE loss function:

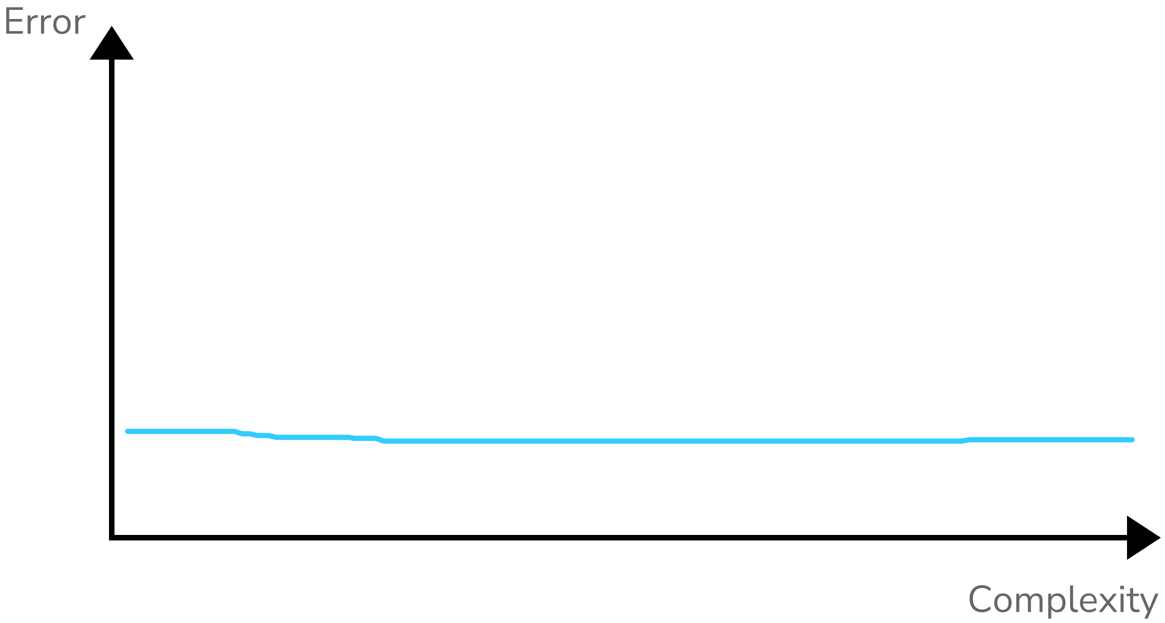
$$\mathbb{E}[(y - \hat{f}(x))^2] = \text{bias}[\hat{f}(x)]^2 + \text{var}(\hat{f}(x)) + \sigma_\epsilon^2$$

Error = Biased squared + Variance + Irreducible Error

Bias-Variance Tradeoff

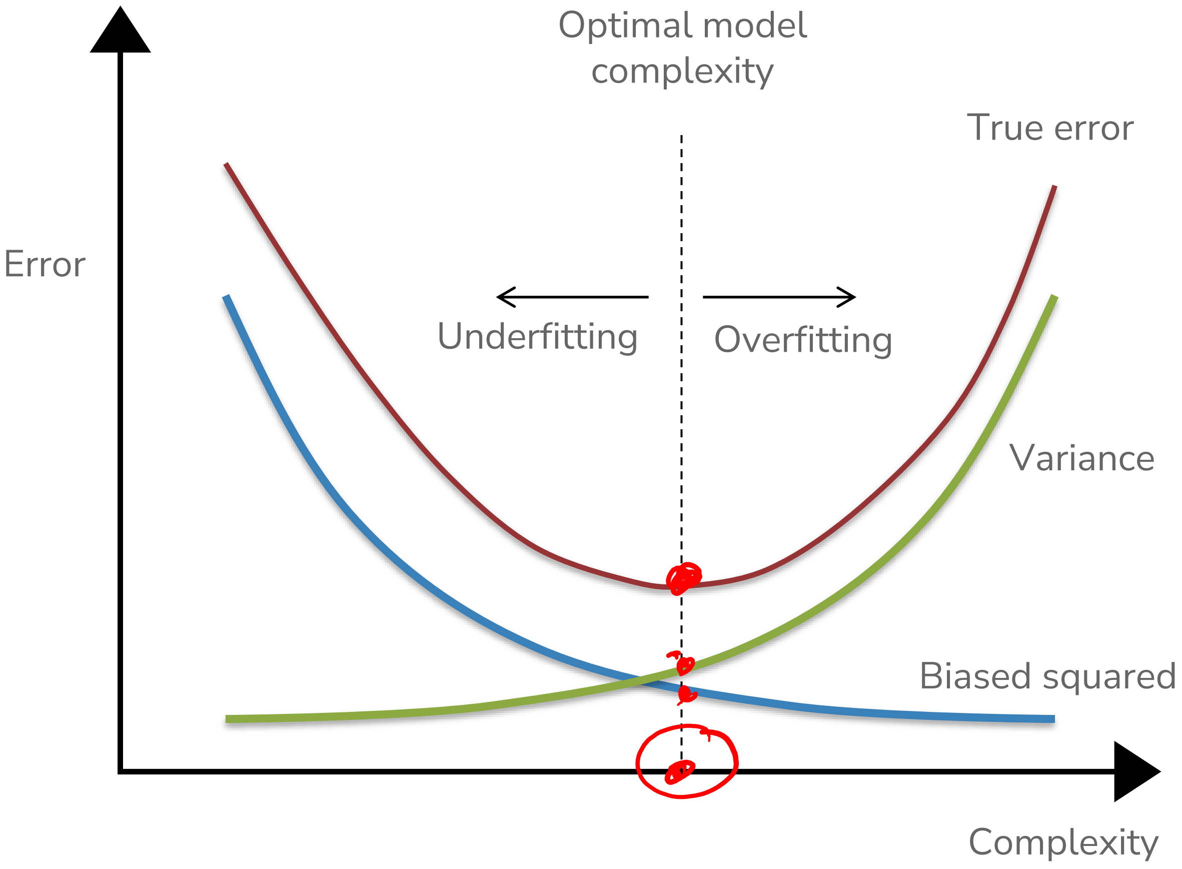
Visually, this looks like the following!

$$Error = Bias^2 + Variance + Irredicible\ Error$$



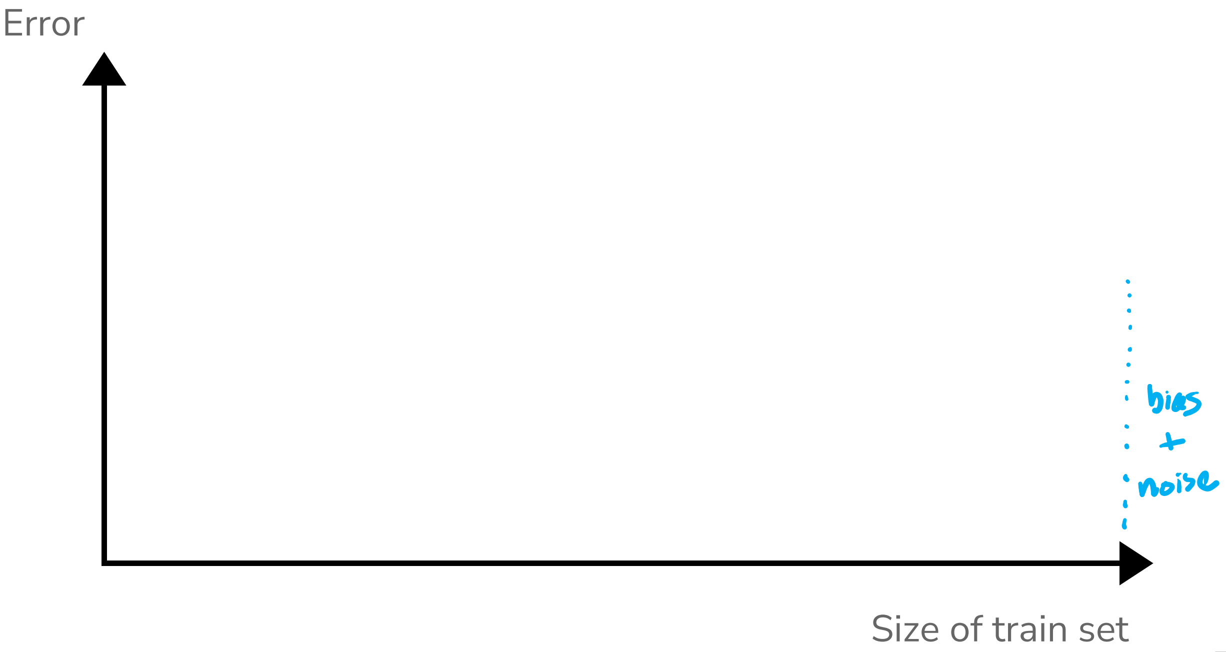
Sq

Bias – Variance Tradeoff



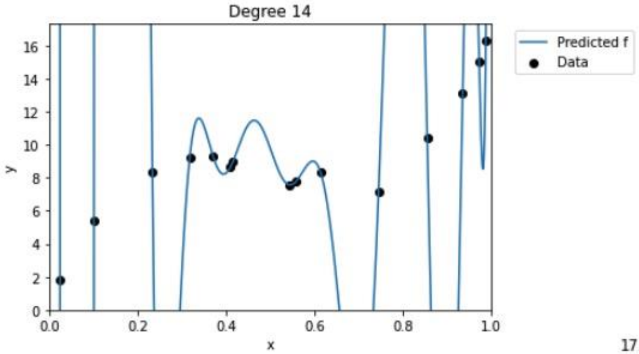
Dataset Size

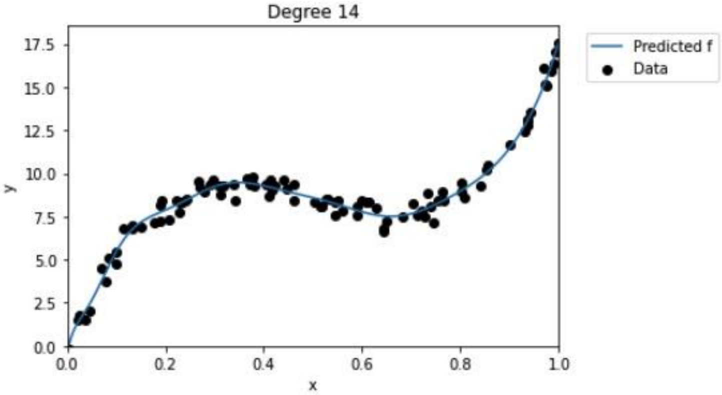
So far our entire discussion of error assumes a fixed amount of data. What happens to our error (true error and training error) as we get more data?



Dataset Size

- Model complexity doesn't depend on the size of the training set
- The larger the training set, the lower the variance of the model, thus less overfitting





Demo

Bias-Variance Tradeoff

- Training a linear regression model in Python
- Observing the effect of the bias-variance tradeoff as compared to model complexity

Brain Break

Choosing Complexity

Choosing Complexity

So far we have talked about the affect of using different complexities on our error. Now, how do we choose the right one?

slido

Think

1 min

sli.do #cs416

Suppose I wanted to figure out the right degree polynomial for my dataset (we'll try p from 1 to 20). What procedure should I use to do this? Pick the best option

For each possible degree polynomial p:

- Train a model with degree p on the training set, pick p that has the lowest test error
- Train a model with degree p on the training set, pick p that has the highest test error
- Train a model with degree p on the test set, pick p that has the lowest test error
- Train a model with degree p on the test set, pick p that has the highest test error
- None of the above

slido

Group

2 min

sli.do #cs416

Suppose I wanted to figure out the right degree polynomial for my dataset (we'll try p from 1 to 20). What procedure should I use to do this? Pick the best option

For each possible degree polynomial p:

- Train a model with degree p on the training set, pick p that has the lowest test error
- Train a model with degree p on the training set, pick p that has the highest test error
- Train a model with degree p on the test set, pick p that has the lowest test error
- Train a model with degree p on the test set, pick p that has the highest test error
- None of the above

Choosing Complexity

We can't just choose the model that has the lowest **train** error because that will favor models that overfit!

It then seems like our only other choice is to choose the model that has the lowest **test** error (since that is our approximation of the true error)

- This is almost right. However, the test set has been **tampered**, thus is no longer is an unbiased estimate of the true error.
- We didn't technically train the model on the test set (that's good), but we chose **which model** to use based on the performance of the test set.
  - It's no longer a stand in for "the unknown" since we probed it many times to figure out which model would be best.

NEVER EVER EVER touch the test set until the end. You only use it ONCE to evaluate the performance of the best model you have selected during training.

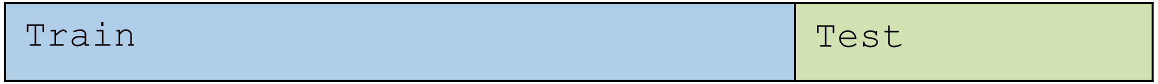
Choosing Complexity

We will talk about two ways to pick the model complexity without ruining our test set.

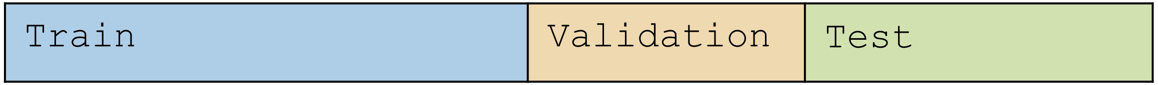
- Using a validation set
- Doing (k-fold) cross validation

Validation Set

So far we have divided our dataset into train and test



We can’t use Test to choose our model complexity, so instead, break up Train into ANOTHER dataset



We will pick the model that does best on validation. Note that this now makes the validation error of the “best” model a biased estimate of true error. The test error will be an unbiased estimate though since we never looked at it!

Validation Set

The process generally goes

```
train, validation, test = random_split(dataset)
for each model complexity p:
    model = train_model(model_p, train)
    val_err = error(model, validation)
    keep track of p and model with smallest val_err
return best p & error(model, test)
```

Validation Set

Pros

Easy to describe and implement

Pretty fast

- Only requires training a model and predicting on the validation set for each complexity of interest

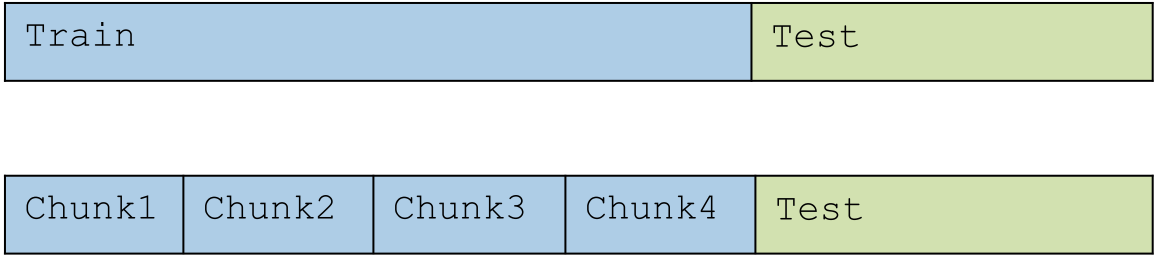
Cons

- Have to sacrifice even more training data
- Prone to overfitting*

Cross-Validation

Clever idea: Use many small validation sets without losing too much training data.

Still need to break off our test set like before. After doing so, break the training set into k chunks.



Train

Test

Chunk1

Chunk2

Chunk3

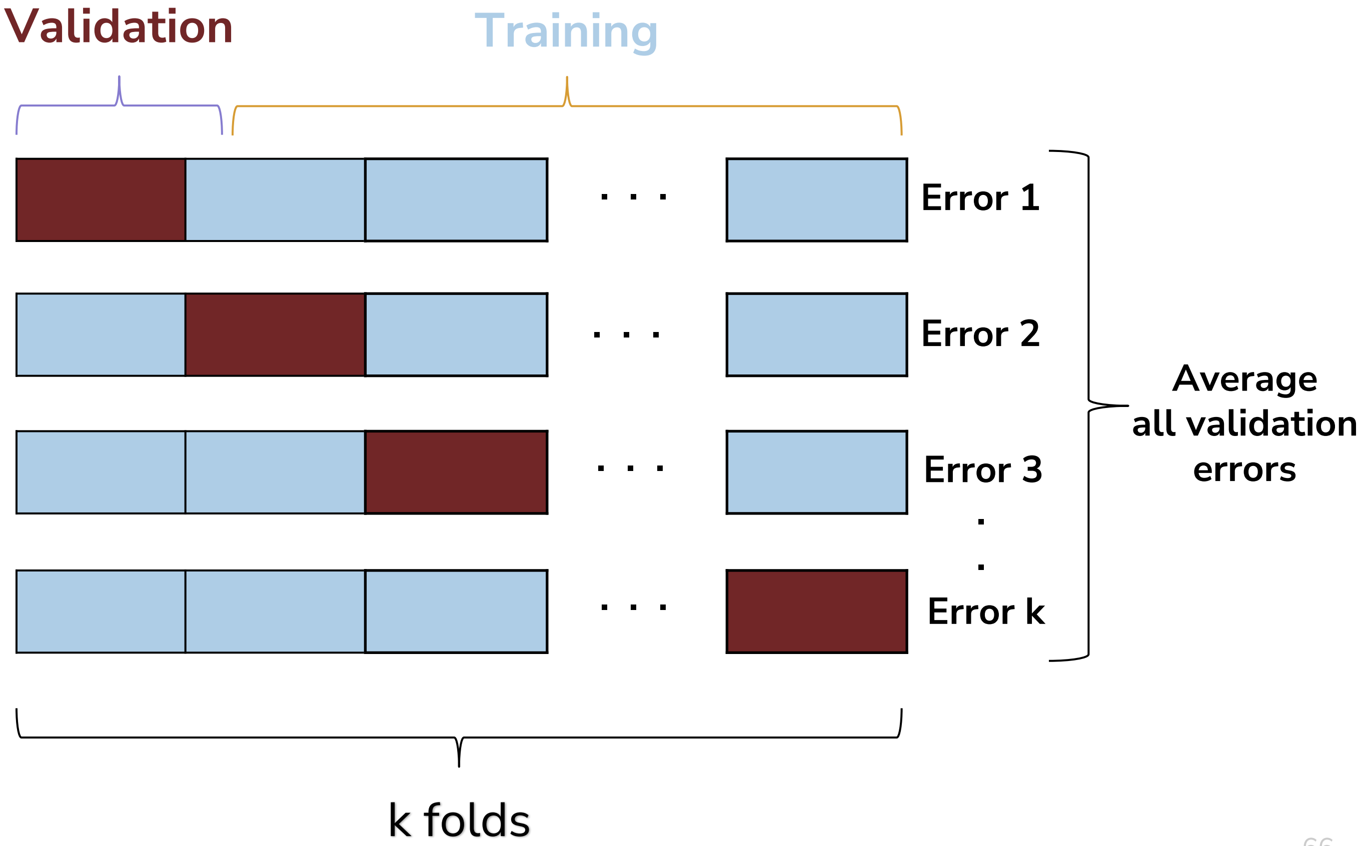
Chunk4

Test

For a given model complexity, train it k times. Each time use all but one chunk and use that left out chunk to determine the validation error.

Cross Validation

For a set of hyperparameters, perform Cross Validation on k folds



Cross-Validation

The process generally goes

```
chunk_1, …, chunk_k, test = random_split(dataset)
for each model complexity p:
    for i in [1, k]:
       model = train_model(model_p, chunks - i)
       val_err = error(model, chunk_i)
    avg_val_err = average val_err over chunks
    keep track of p with smallest avg_val_err
return model trained on train (all chunks) with
best p & error(model, test)
```

Cross-Validation

Pros

- Prevent overfitting: By training the model on multiple folds instead of only 1 training set, this learns the model with the best generalization capabilities.
- Don’t have to actually get rid of any training data!

Cons

- Slow. For each model selection, we have to train k times
- Very computationally expensive

Cross-Validation

Generally, the more folds you use the better as you aren’t relying on the specifics of a single validation fold.

- Theoretical best estimator* is to use $k = n$
    - Called "**Leave One Out Cross Validation**” (LOOCV)
- In practice, people use $k = 5$ to 10 for computational simplicity

Recap

Theme: Assess the performance of our models

Ideas:

- Model complexity
- Train vs. Test vs. True error
- Overfitting and Underfitting
- Bias-Variance Tradeoff
- Error as a function of train set size
- Choosing best model complexity
  - Validation set
  - Cross Validation