

Python Data Products

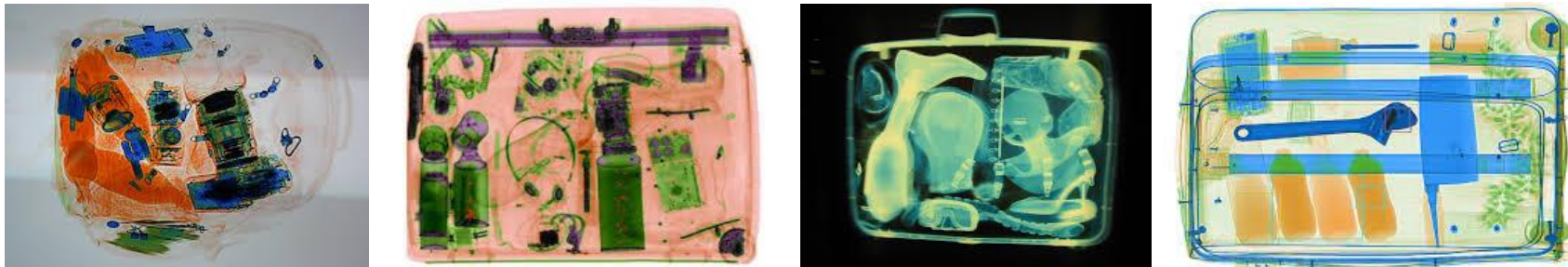
Course 3: Making Meaningful Predictions from Data

Lecture: Course Introduction

Course introduction: Week 1

In this course we will cover...

- Evaluation metrics for **regression** and **classification** algorithms
- How to select the **right** evaluation metrics under different conditions

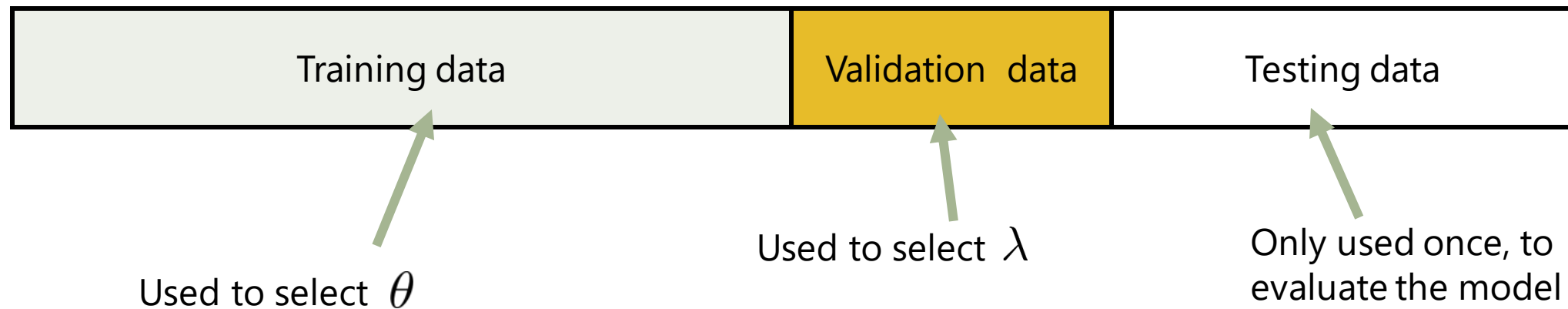


e.g. which of these bags contains a weapon?

Course introduction: Week 2

In this course we will cover...

- **Predictive pipelines:** how do we build models that will generalize well to new data?



Course introduction: Week 3

In this course we will cover...

- Implementation of predictive pipelines in Python
- Rules and guidelines for model selection using training/validation/test sets

Learning outcomes

By the end of this course you should be able to:

- Implement **robust** predictive pipelines in Python, and deal with issues surrounding evaluation metrics, model complexity, and generalization

Week 1

Regression and
Classification Diagnostics

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Introduction

Learning objectives

In this lecture we will...

- Introduce the problem of **evaluating machine learning systems**
- Outline the content of this course

Challenges in model evaluation

- How can we **evaluate** regression and classification models?
 - What does it mean for a model to be "good"?
- Does the meaning of "good performance" change in different contexts?
 - How can we **compare** the performance of different models?
- If a model works well on our training data, how can we ensure that it will still work well on ***unseen*** data?

Challenges in model evaluation

- How can we **evaluate** regression and classification models?
 - E.g. should we consider the average error? The average squared error? The accuracy? The number of false positives? etc.
 - What is the motivation behind (and the consequences of) these different choices?

Challenges in model evaluation

- What does it mean for a model to be "good"?
- Does the meaning of "good performance" change in different contexts?
- E.g. consider running a classifier that recognizes fingerprints, versus one that detects pedestrians, versus one that detects weapons in luggage
- How can we design a classifier that targets one of these situations?

Challenges in model evaluation

- How can we **compare** the performance of different models?
- Given two classifiers (or regressors) how can we decide which of them is better?
- Is it enough to select whichever classifier has the highest accuracy?
- How can we make other decisions, e.g. if we were using a one-hot encoding to represent time, should we do so at the granularity of weeks, or months?

Challenges in model evaluation

- If a model works well on our training data, how can we ensure that it will still work well on ***unseen*** data?
- E.g. a higher level of granularity (e.g. days rather than weeks) for our one-hot encoding might always lead to better performance on our training data, but does that mean it is the better model?
- If not, how can we develop a training regime that corrects for this issue?

Summary of concepts

- Introduced the main problems and challenges in **evaluating regression and classification** models
- Outlined the remainder of this course

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Recap on mathematical notation

Learning objectives

In this lecture we will...

- Revise the mathematical notation necessary to cover the basics of evaluation and regularization

Notation for evaluation and regularization

\bar{x} : **mean** (average) of a vector

$\text{Var}(x)$: Variance of a vector

λ : regularization tradeoff parameter

$\|\theta\|_2^2 = \sum_i \theta_i^2$: vector **norm**

(in general $\|\theta\|_p = (\sum_i |\theta_i|^p)^{\frac{1}{p}}$)

Summary of concepts

- Covered basic notation for evaluation and regularization

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Motivation behind the MSE

Learning objectives

In this lecture we will...

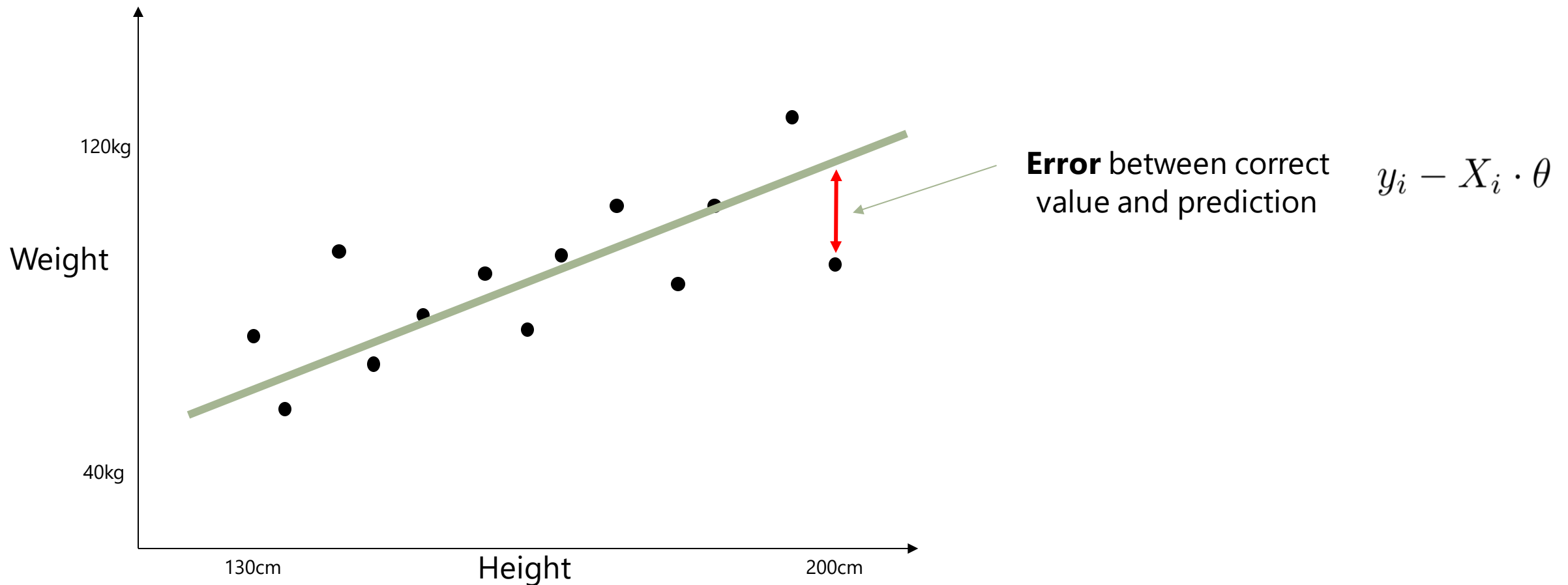
- Present the most common evaluation measures used for regression models (the Mean Squared Error)
- Motivate the choice of this particular error measure using statistics and probability

Regression diagnostics

Q: How should we evaluate our regression model?

Regression diagnostics

Q: Can we find a line that (approximately) fits the data)?



Concept: Mean Squared Error

Mean-squared error (MSE)

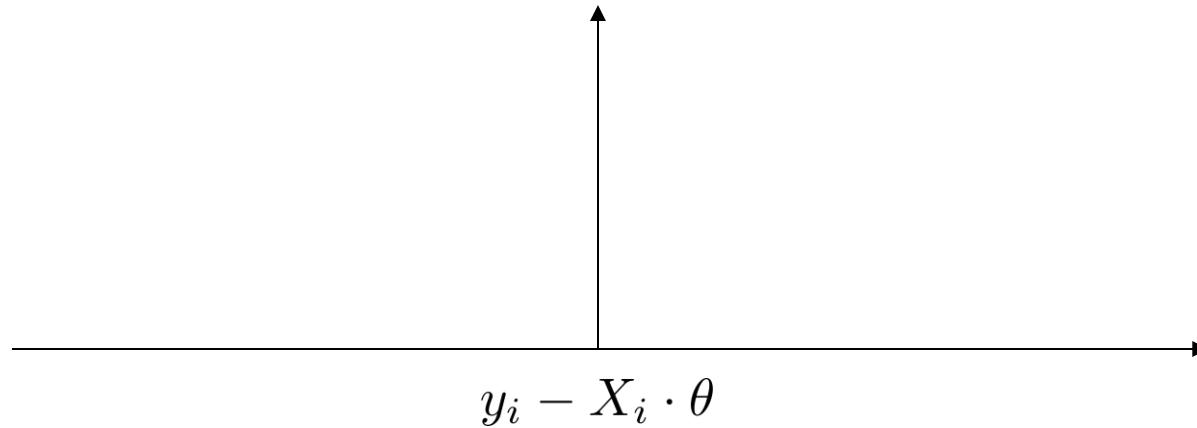
$$\frac{1}{N} \|y - X\theta\|_2^2$$

$$= \frac{1}{N} \sum_{i=1}^N (y_i - X_i \cdot \theta)^2$$

Regression diagnostics

Q: Why MSE (and not mean-absolute-error or something else)

Regression diagnostics



label = prediction + error

$$y_i = X_i \cdot \theta + \mathcal{N}(0, \sigma)$$

Regression diagnostics

$$p_{\theta}(y|X) = \prod_i \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(y_i - X_i \cdot \theta)^2}{2\sigma^2}}$$

$$\begin{aligned} \max_{\theta} p_{\theta}(y|X) &= \max_{\theta} \prod_i e^{-(y_i - X_i \cdot \theta)^2} \\ &= \min_{\theta} \sum_i (y_i - X_i \cdot \theta)^2 \end{aligned}$$

Summary of concepts

- Understand the motivation behind the MSE in terms of probability
- Understand the notion of "error distributions"
- (at a high level) understand the relationship between likelihood (probability) and error (prediction)

On your own...

- Compute MSE and related statistics (like Mean **Absolute** Error) and compare cases where the errors are high and low

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Regression Diagnostics: MSE and R^2

Learning objectives

In this lecture we will...

- Present the R^2 statistic
- Explain the relationship between MSE and mean+variance

Regression diagnostics: Coefficient of determination

Q: How low does the MSE have to be before it's "low enough"?

A: It depends! The MSE is proportional to the **variance** of the data

Coefficient of determination (R^2 statistic)

Mean:

Variance:

MSE:

Coefficient of determination (R^2 statistic)

Mean: $\bar{y} = \frac{1}{N} \sum_{i=1}^N y_i$



Variance: $Var(y) = \frac{1}{N} \sum_{i=1}^N (\bar{y} - y_i)^2$

MSE: $\frac{1}{N} \sum_{i=1}^N (X_i \cdot \theta - y_i)^2$

Coefficient of determination (R^2 statistic)

$$FVU(f) = \frac{MSE(f)}{Var(y)}$$

(FVU = fraction of variance unexplained)

$FVU(f) = 1$  Trivial predictor
 $FVU(f) = 0$  Perfect predictor

Coefficient of determination (R^2 statistic)

$$R^2 = 1 - FVU(f) = 1 - \frac{MSE(f)}{Var(y)}$$

$R^2 = 0 \longrightarrow$ Trivial predictor

$R^2 = 1 \longrightarrow$ Perfect predictor

Coefficient of determination (R^2 statistic)

Q: Is it possible to have a **negative** R^2 ?

A: Yes! A "trivial" predictor has an R^2 of zero, but it's possible to have a predictor that's worse than a trivial predictor

e.g. predict star ratings as the mean $\rightarrow R^2 = 0$
Predict all star ratings as 0 $\rightarrow R^2 < 0$

Summary of concepts

- Introduced the R^2 statistic
- Explained the relationship between MSE, mean+variance, and R^2

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Over and underfitting

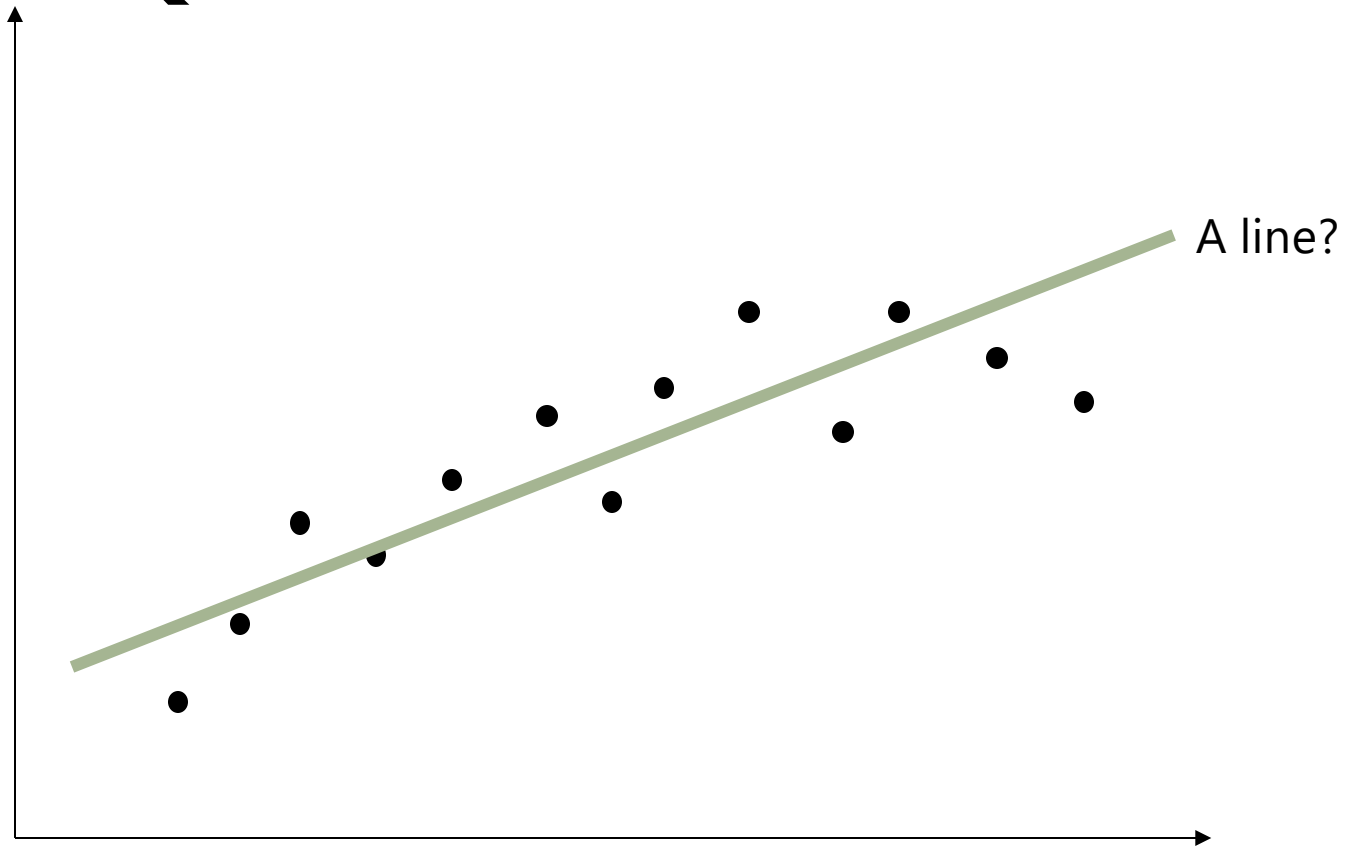
Learning objectives

In this lecture we will...

- Introduce the concept of **overfitting**
- expand our previous discussion training and test sets

Example

Q: What model is the best fit to this data?



Example

A high-degree polynomial might be the best fit to the data (in terms of the mean-squared error), but intuition tells us this is not a good solution

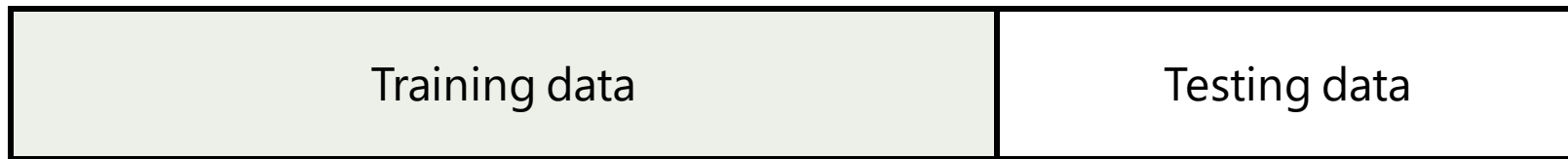
Overfitting

Q: But can't we get an R^2 of 1 (MSE of 0) just by throwing in enough random features?

A: Yes! This is why MSE and R^2 (or other statistics) should be evaluated on data that **wasn't** used to train the model

A good model is one that
generalizes to new data

Training and test sets



$$\begin{array}{c} \text{train} \rightarrow \\ \text{test} \rightarrow \end{array} \left[\begin{array}{c} \\ \hline \end{array} \right] X \theta = \left[\begin{array}{c} \\ \hline \end{array} \right] y$$

Training and test sets

- We first split our data into a **training** and a **test** set
- The **training set** is used to tune the model parameters (i.e., θ)
- The **test set** is used to evaluate the model's performance on unseen data

Training and test sets

How should the training and test sets be selected?

- The training and test sets should be **non-overlapping** samples of the data
- They should each be a **random** sample of the data

Training and test sets

The **size** of the training and test sets should be chosen to balance various considerations:

- The training set should be large enough (compared to the model complexity) so that we don't overfit too badly
- The test set should be large enough so that it is representative of the variance in the data
 - We might also be constrained by running time, etc.

Summary of concepts

- Explained the difference between training performance versus generalization ability
- Showed how a training and test set can be used to measure generalization ability

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Classification Diagnostics:

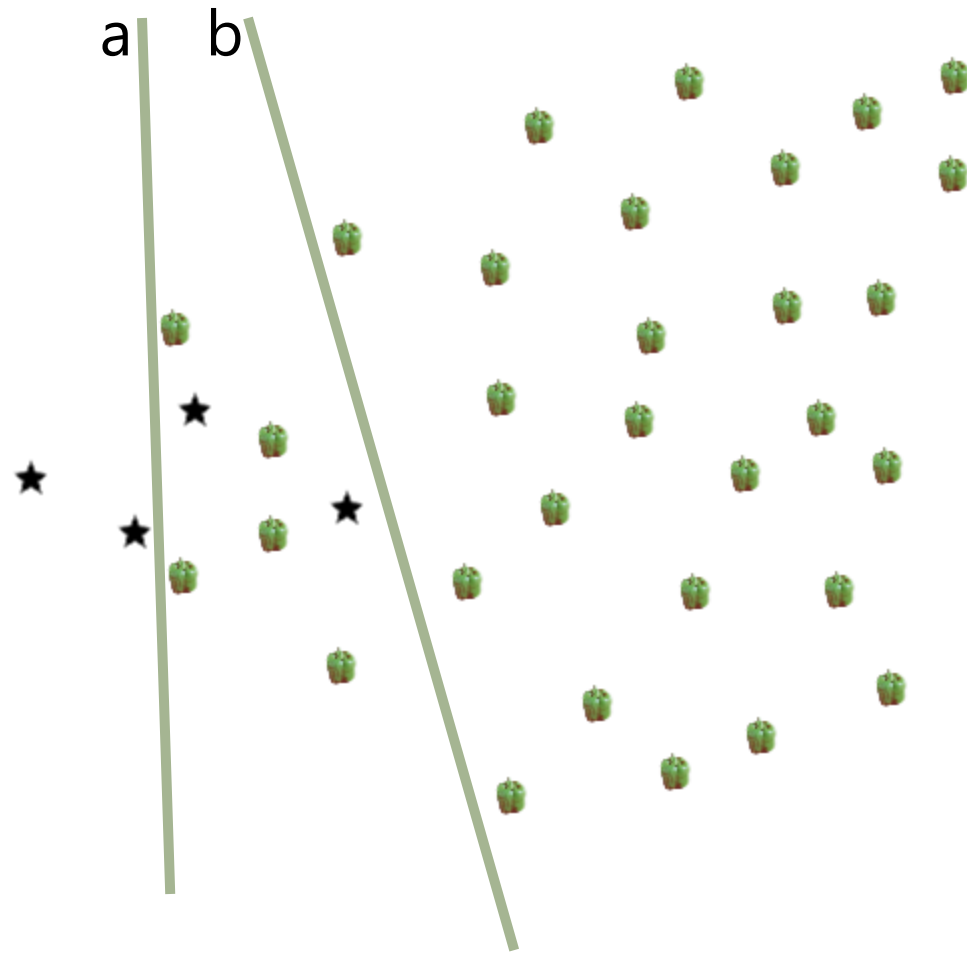
Accuracy & Error

Learning objectives

In this lecture we will...

- Introduce a variety of diagnostics for evaluating classifiers
- Describe different situations where different performance measures may be preferable

Which of these classifiers is best?



Which of these classifiers is best?

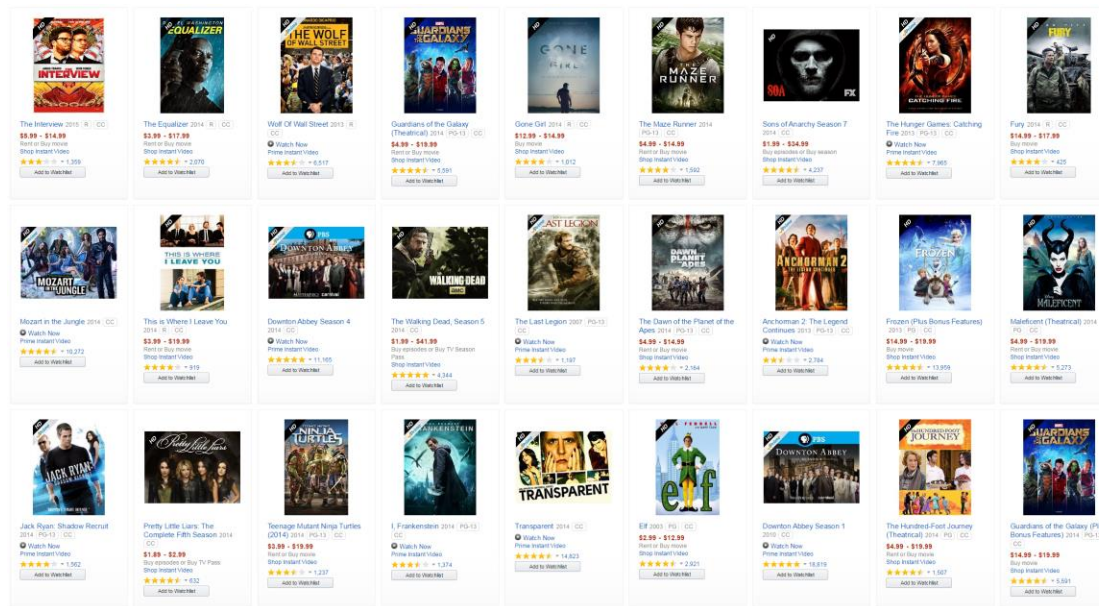
The solution which minimizes the
#errors may not be the best one

Which of these classifiers is best?

1. When data are highly imbalanced

If there are far fewer positive examples than negative examples we may want to assign additional weight to negative instances (or vice versa)

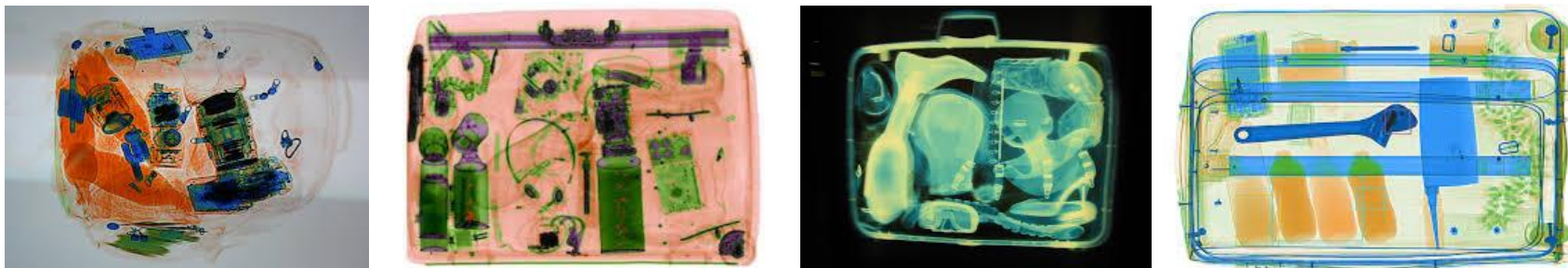
e.g. will I purchase a product? If I purchase 0.00001% of products, then a classifier which just predicts "no" everywhere is 99.99999% accurate, but not very useful



Which of these classifiers is best?

2. When mistakes are more costly in one direction

False positives are nuisances but false negatives are disastrous (or vice versa)

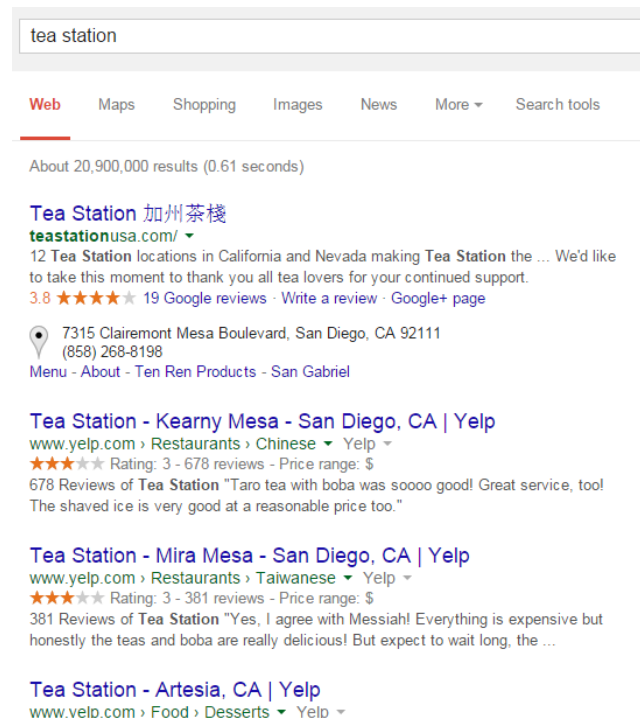


e.g. which of these bags contains a weapon?

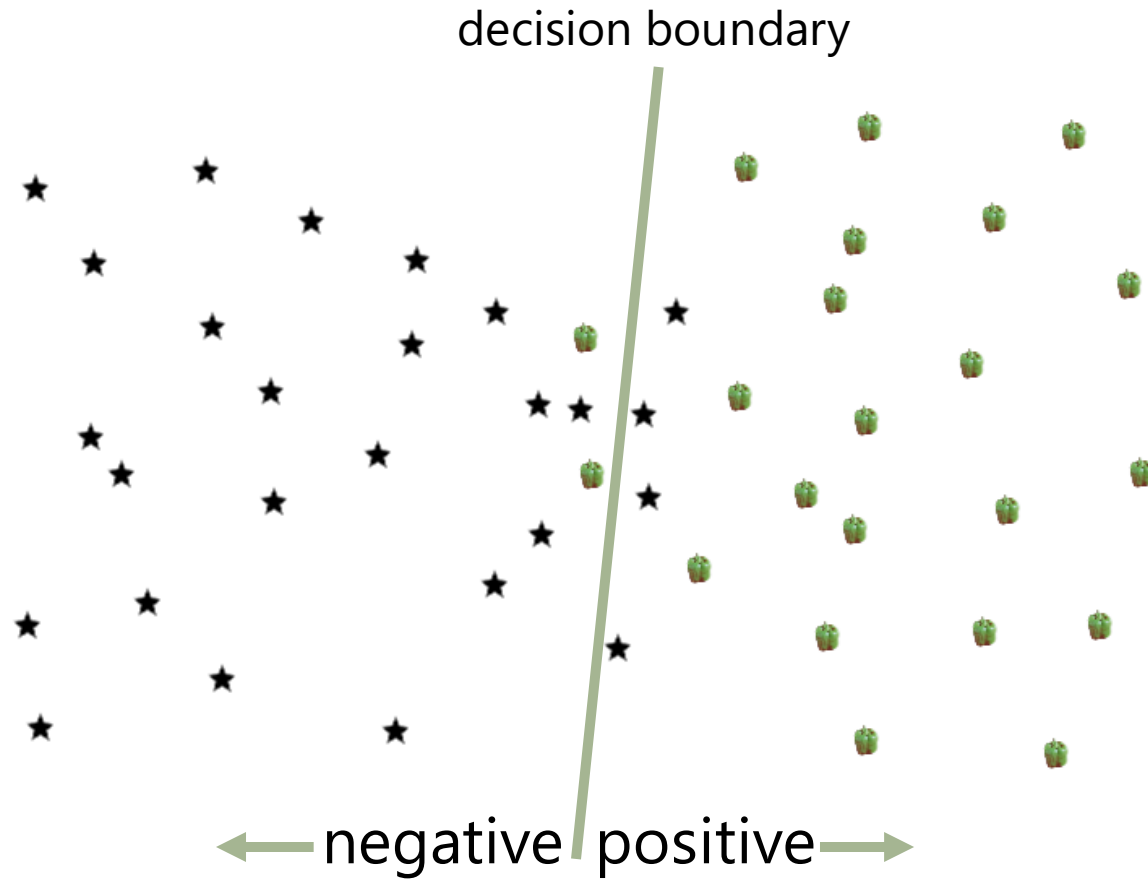
Which of these classifiers is best?

3. When we only care about the “most confident” predictions

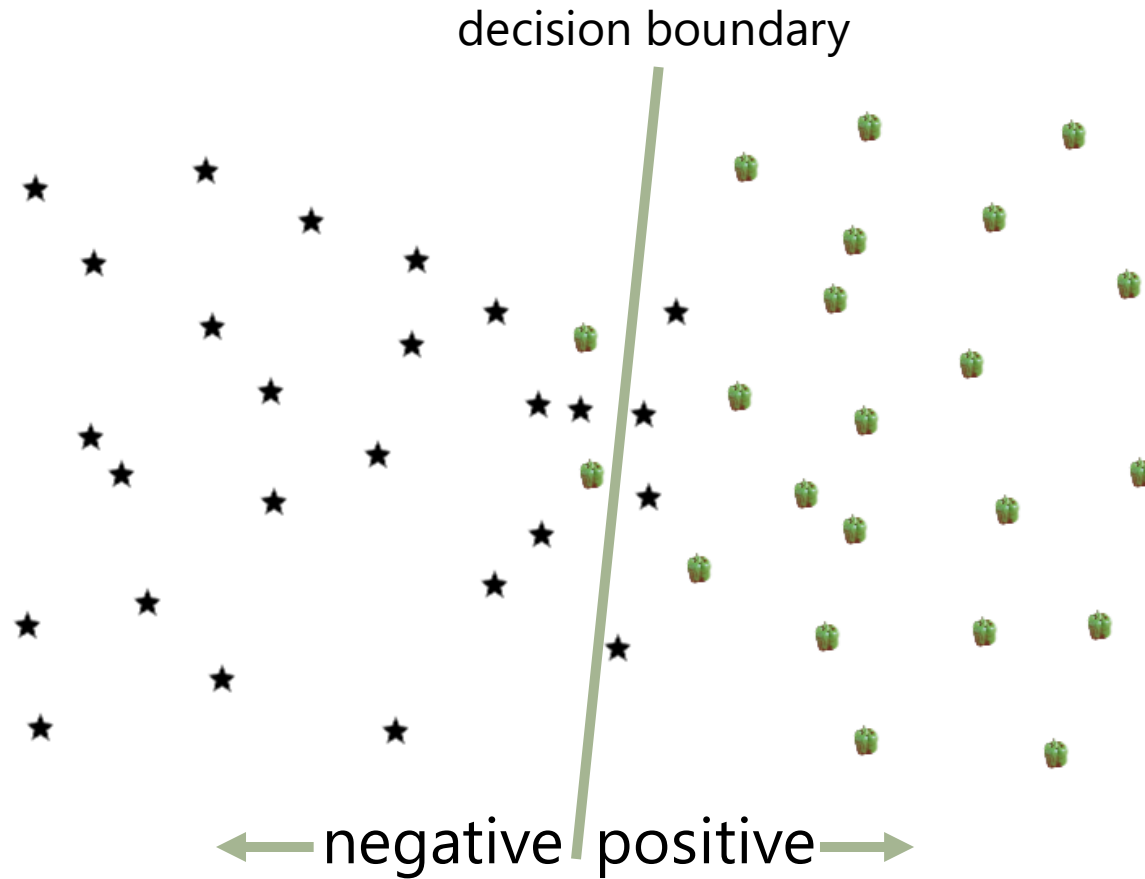
e.g. does a relevant result appear among the first page of results?



Evaluating classifiers

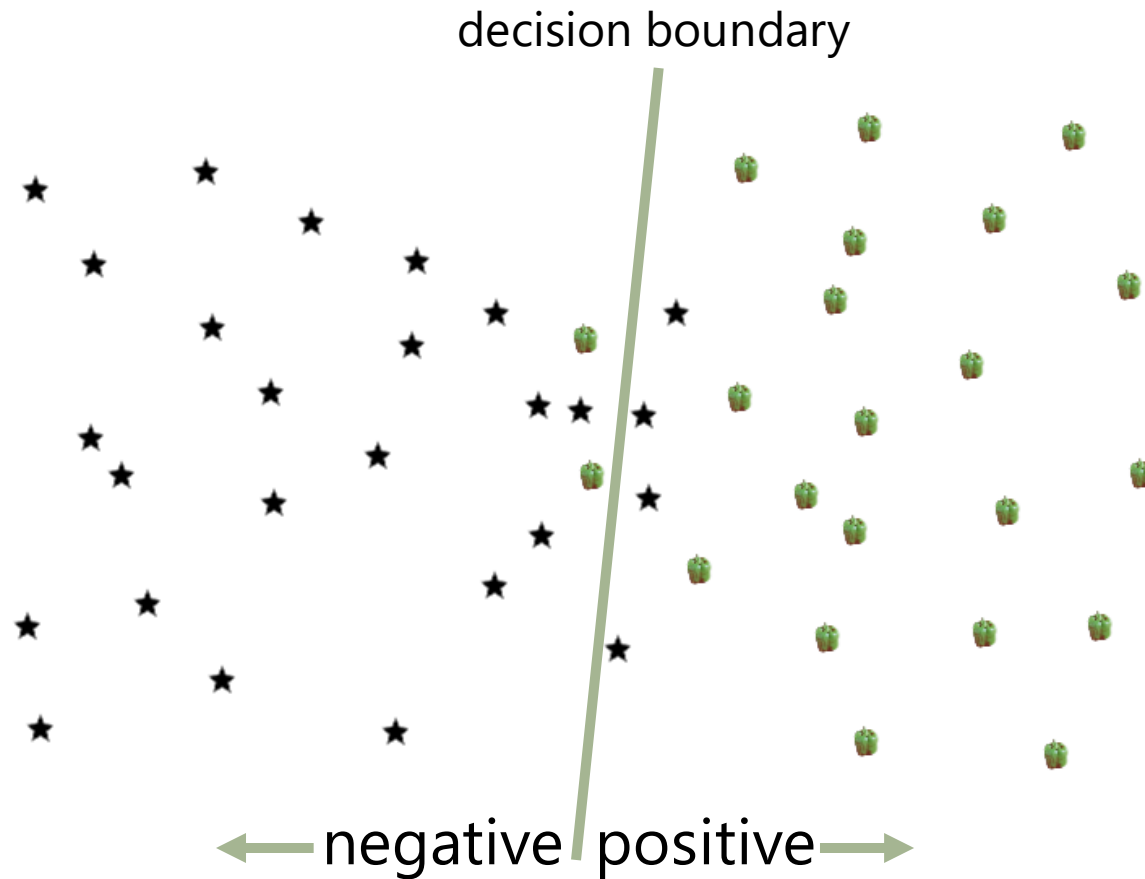


Evaluating classifiers



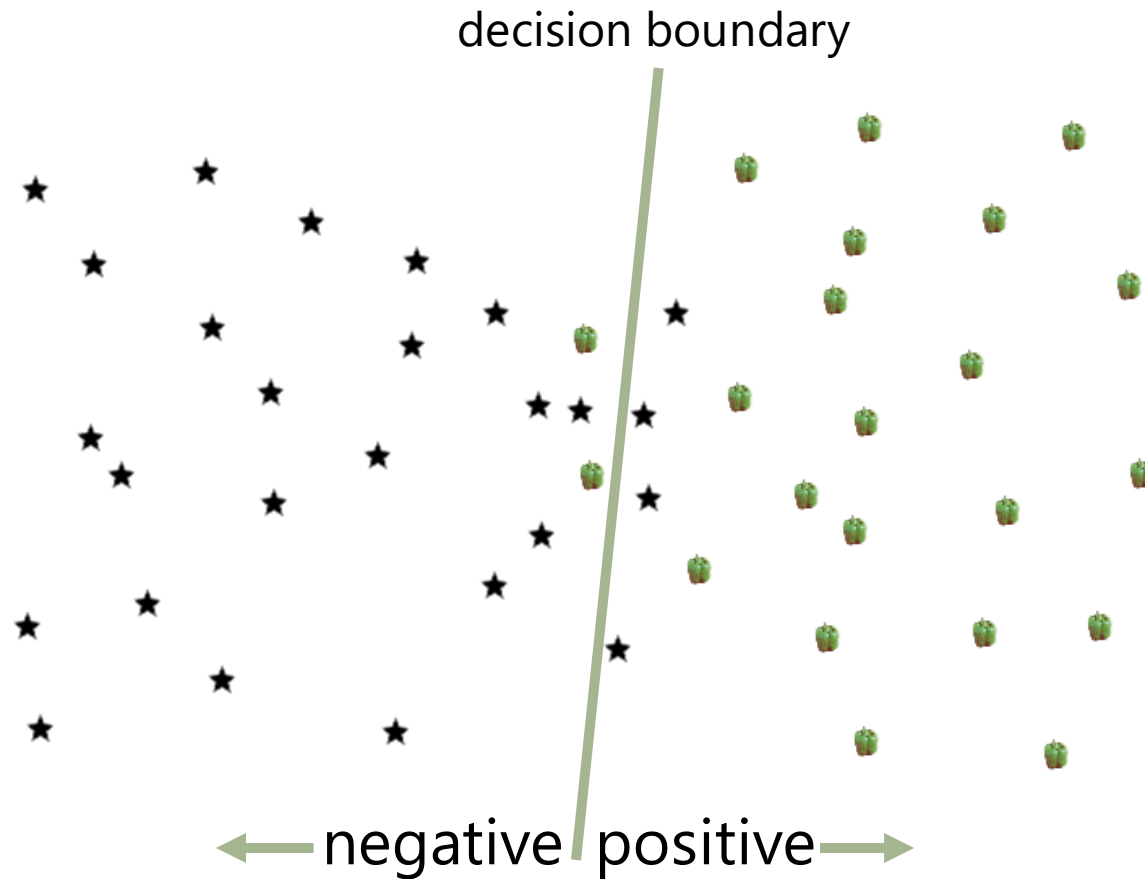
TP (true positive): Labeled as , predicted as

Evaluating classifiers



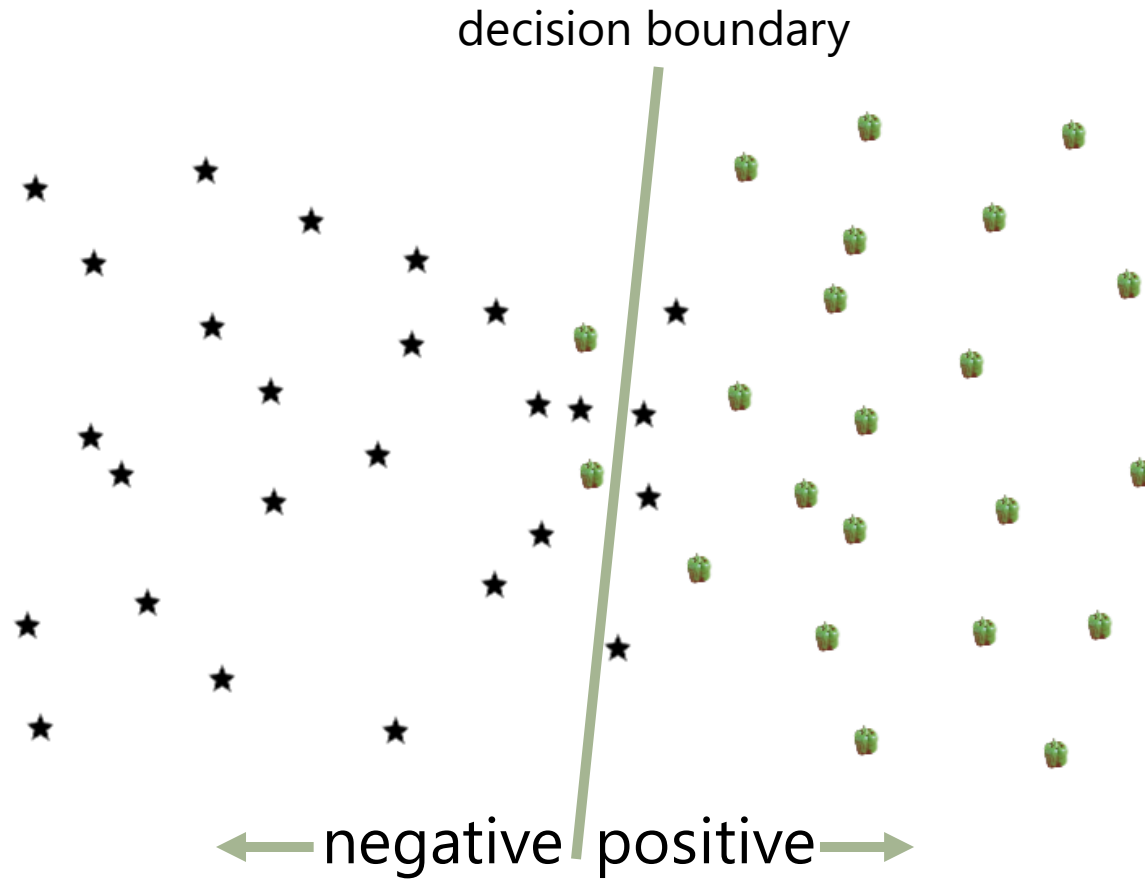
TN (true negative): Labeled as , predicted as

Evaluating classifiers



FP (false positive): Labeled as , predicted as

Evaluating classifiers



FN (false negative): Labeled as , predicted as

Evaluating classifiers

		Label	
		true	false
Prediction	true	true positive	false positive
	false	false negative	true negative

Classification accuracy = correct predictions / #predictions
=

Error rate = incorrect predictions / #predictions
=

Evaluating classifiers

		Label	
		true	false
Prediction	true	true positive	false positive
	false	false negative	true negative

True positive rate (**TPR**) = true positives / #labeled positive
=

True negative rate (**TNR**) = true negatives / #labeled negative
=

Evaluating classifiers

		Label	
		true	false
Prediction	true	true positive	false positive
	false	false negative	true negative

$$\text{Balanced Error Rate (BER)} = \frac{1}{2} (\text{FPR} + \text{FNR})$$

= $\frac{1}{2}$ for a random/naïve classifier, 0 for a perfect classifier

Summary of concepts

- Introduced several diagnostics for evaluating classification algorithms
- Described situations where each diagnostic technique might be useful

On your own...

- Using one of our previous classification examples, try evaluating each of these measures, e.g. TPR, TNR, BER, etc.

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Classification Diagnostics:

Precision & Recall

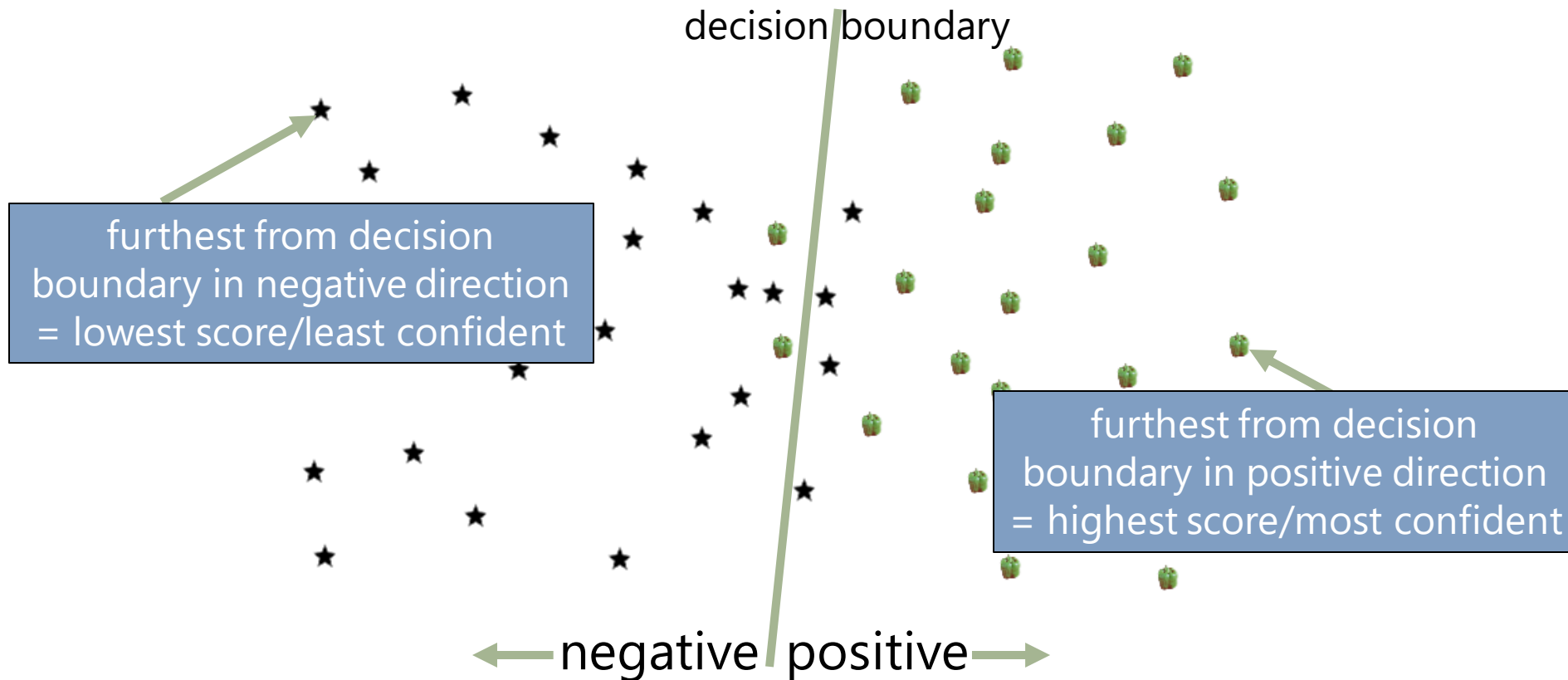
Learning objectives

In this lecture we will...

- Introduce the relationship between classification and **ranking**
- Describe techniques for evaluating classifiers when our goal is to use them for ranking
- Introduce the concepts of **precision** and **recall**

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction



Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

- In ranking settings, the actual labels assigned to the points (i.e., which side of the decision boundary they lie on) **don't matter**
- All that matters is that positively labeled points tend to be at **higher ranks** than negative ones

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

- For naïve Bayes, the “score” is the ratio between an item having a positive or negative class
 - For logistic regression, the “score” is just the probability associated with the label being 1
- For Support Vector Machines, the score is the distance of the item from the decision boundary (together with the sign indicating what side it's on)

Evaluating classifiers – ranking

The classifiers we've seen can
associate **scores** with each prediction

e.g.

$\mathbf{y} = [1, -1, 1, 1, 1, -1, 1, 1, -1, 1]$
Confidence = $[1.3, -0.2, -0.1, -0.4, 1.4, 0.1, 0.8, 0.6, -0.8, 1.0]$

Sort **both** according to confidence

Evaluating classifiers

e.g.

y = [1, -1, 1, 1, 1, -1, 1, 1, -1, 1]
Confidence = [1.3, -0.2, -0.1, -0.4, 1.4, 0.1, 0.8, 0.6, -0.8, 1.0]

Evaluating classifiers

e.g.

y = [1, -1, 1, 1, 1, -1, 1, 1, -1, 1]
Confidence = [1.3, -0.2, -0.1, -0.4, 1.4, 0.1, 0.8, 0.6, -0.8, 1.0]




y = [1, 1, 1, 1, 1, -1, 1, -1, 1, -1]
Confidence = [1.4, 1.3, 1.0, 0.8, 0.6, 0.1, -0.1, -0.2, -0.4, -0.8]

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

Labels sorted by confidence:

[1, 1, 1, 1, 1, -1, 1, -1, 1, -1]



Suppose we have a fixed budget (say, six) of items that we can return (e.g. we have space for six results in an interface)

- Total number of **relevant** items =
- Number of items we returned =
- Number of **relevant items** we returned =

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

“fraction of retrieved documents that are relevant”

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

“fraction of relevant documents that were retrieved”

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

$\text{precision@}k$ = precision when we have a budget of k retrieved documents

e.g.

- Total number of **relevant** items = 7
- Number of items we returned = 6
- Number of **relevant items** we returned = 5

$\text{precision@}6 =$

Evaluating classifiers – ranking

The classifiers we've seen can associate **scores** with each prediction

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

(harmonic mean of precision and recall)

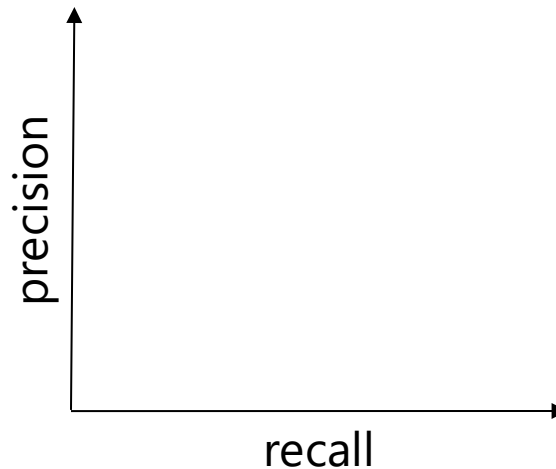
$$F_\beta = (1 + \beta^2) \cdot \frac{\text{precision} \cdot \text{recall}}{\beta^2 \text{precision} + \text{recall}}$$

(weighted, in case precision is more important
(low beta), or recall is more important (high beta))

Precision/recall curves

How does our classifier behave as we “increase the budget” of the number retrieved items?

- For budgets of size 1 to N, compute the precision and recall
- Plot the precision against the recall



Summary of concepts

- Discussed the relationship between ranking and classification
- Introduced additional diagnostic techniques for evaluating classifiers for ranking-based settings

On your own...

- Using one of our previous classification examples, try evaluating it in terms of ranking performance, e.g. precision, recall, F1-score

Week 2

Training and testing
pipelines

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Setting up a codebase for
evaluation and validation

Learning objectives

In this lecture we will...

- Setup a simple codebase to be used in future lectures for the purpose of evaluating regressors and classifiers, and for implementing training/validation/testing pipelines

Code example: sentiment analysis

In this lecture, we'll build a model that implements **sentiment analysis**, i.e., our goal is to predict star ratings based on the text in a review

We choose this problem mainly because it includes **complex, high-dimensional features**, such that model tuning and evaluation becomes important

Code example: sentiment analysis

Importing libraries and reading data:

```
In [1]: import gzip
        from collections import defaultdict
        import string # Some string utilities
        import random
        from nltk.stem.porter import PorterStemmer # Stemming
        import numpy
```

```
In [2]: path = "/home/jmcauley/datasets/mooc/amazon/amazon_reviews_us_Gift_Card_v1_00.tsv.gz"
```

```
In [3]: f = gzip.open(path, 'rt', encoding="utf8")
```

Code example: sentiment analysis

Importing libraries and reading data:

```
In [4]: header = f.readline()
header = header.strip().split('\t')
```

```
In [5]: dataset = []
```

```
In [6]: for line in f:
    fields = line.strip().split('\t')
    d = dict(zip(header, fields))
    d['star_rating'] = int(d['star_rating'])
    d['helpful_votes'] = int(d['helpful_votes'])
    d['total_votes'] = int(d['total_votes'])
    dataset.append(d)
```

Code example: sentiment analysis

Our goal is going to be to build a classifier which estimates **sentiment** (e.g. a star-rating) based on the occurrence of words in a document:

i.e., Let's build a predictor of the form:

$$f(\text{text}) \rightarrow \text{rating}$$

using a model based on linear regression:

$$\text{rating} \simeq \alpha + \sum_{w \in \text{text}} \text{count}(w) \cdot \theta_w$$

Code example: sentiment analysis

Our first challenge is to build a (relatively) small dictionary of words to use in our model (since it would be impractical to include every word)

Code example: sentiment analysis

Counting unique words:

In [7]: *# How many unique words are there?*

```
In [8]: wordCount = defaultdict(int)
        for d in dataset:
            for w in d['review_body'].split():
                wordCount[w] += 1

        print(len(wordCount))
```

97289

Code example: sentiment analysis

Removing capitalization and punctuation

In [9]: *# What if we ignore capitalization and punctuation?*

```
In [10]: wordCount = defaultdict(int)
punctuation = set(string.punctuation)
for d in dataset:
    r = ''.join([c for c in d['review_body'].lower() if not c in punctuation])
    for w in r.split():
        wordCount[w] += 1

print(len(wordCount))
```

46283

Concept: Stemming

Stemming is a process that maps different instances of words to a unique word "stem":

drinks → drink
drinking → drink
drinker → drink

E.g.

argue → argu
arguing → argu
argues → argu
arguing → argu
argus → argu

Code example: sentiment analysis

Stemming

In [11]: *# What if we apply stemming?*

```
In [12]: wordCount = defaultdict(int)
punctuation = set(string.punctuation)
stemmer = PorterStemmer()
for d in dataset:
    r = ''.join([c for c in d['review_body'].lower() if not c in punctuation])
    for w in r.split():
        w = stemmer.stem(w) # with stemming
        wordCount[w] += 1

print(len(wordCount))
```

37480

We use a stemmer from the Python Natural Language Toolkit (NLTK) called the **Porter Stemmer**

Code example: sentiment analysis

Even after removing punctuation, capitalization, and stemming, we still have a dictionary that is **too large** to deal with practically

So, let's just take the subset of the **most popular** words to build our dictionary

Code example: sentiment analysis

Extracting the most popular words:

In [13]: *# Extract and build features from the most common words*

```
In [14]: wordCount = defaultdict(int)
punctuation = set(string.punctuation)

for d in dataset:
    r = ''.join([c for c in d['review_body'].lower() if not c in punctuation])
    for w in r.split():
        wordCount[w] += 1
```

Just removing capitalization and punctuation for the purposes of this example

```
In [15]: counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()

words = [x[1] for x in counts[:1000]]

wordId = dict(zip(words, range(len(words))))
wordSet = set(words)
```

Sort words by popularity and keep the **top 1000** most popular

Utility data structures to map each word to a unique ID

Code example: sentiment analysis

Extracting features from the most popular words

```
In [16]: def feature(datum):  
    feat = [0]*len(words)  
    r = ''.join([c for c in datum['review_body'].lower() if not c in punctuation])  
    for w in r.split():  
        if w in words:  
            feat[wordId[w]] += 1  
    feat.append(1) #offset  
    return feat
```

Append the offset
feature to the end

Increment the counter of the
corresponding word each time
we see that word in the text

Code example: sentiment analysis

Finally, having extract a (fixed-length) feature vector for each document (review), we can train our sentiment analysis model:

$$\text{rating} \simeq \alpha + \sum_{w \in \text{text}} \text{count}(w) \cdot \theta_w$$

Code example: sentiment analysis

For the moment, we fit our model much as we have done in previous lectures (though will change this in later lectures)

```
In [17]: random.shuffle(dataset)
```

```
In [18]: X = [feature(d) for d in dataset]
```

```
In [19]: y = [d['star_rating'] for d in dataset]
```

```
In [20]: theta, residuals, rank, s = numpy.linalg.lstsq(X, y)
```

Code example: sentiment analysis

Finally, we can examine which words have the most positive/negative sentiment by looking at their corresponding coefficients:

```
In [21]: wordWeights = list(zip(theta, words + ['offset']))  
wordWeights.sort()
```

```
In [22]: wordWeights[:10]
```

```
Out[22]: [(-1.2154565072717696, 'disappointing'),  
          (-0.8574720172738775, 'disappointed'),  
          (-0.7905359349220544, 'unable'),  
          (-0.6808380275904105, 'waste'),  
          (-0.6634111839366597, 'charged'),  
          (-0.5391972452016565, 'supposed'),  
          (-0.5292354754787302, 'unfortunately'),  
          (-0.49739634446194575, 'australia'),  
          (-0.4964445645712913, 'tried'),  
          (-0.47776774788212384, 'wont')]
```

```
In [23]: wordWeights[-10:]
```

```
Out[23]: [(0.23601901891940102, 'whats'),  
          (0.2383326014985222, 'problems'),  
          (0.24436555664648224, 'particular'),  
          (0.24700474913779302, 'worry'),  
          (0.2536120024564045, 'exelente'),  
          (0.2597913183314589, 'excelent'),  
          (0.27148055221480827, 'excelente')]
```

Summary of concepts

- Developed a new codebase for a sentiment analysis problems
- Showed some of the challenges in modeling features from text

On your own...

- Modify the code to experiment with alternative feature representations (e.g. removing or keeping capitalization, punctuation, changing the dictionary size, etc.) to determine their impact on model accuracy

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Model Complexity and
regularization

Learning objectives

In this lecture we will...

- Introduce different notions of **model complexity**, and show how we can encourage our model to favor simpler (over more complex) solutions
- Further discuss the concept of **overfitting**

Concept: Overfitting

When a model performs well on **training** data but doesn't generalize, we are said to be **overfitting**

Concept: Overfitting

When a model performs well on **training** data but doesn't generalize, we are said to be **overfitting**

Q: What can be done to avoid overfitting?

How to avoid overfitting

Q: What can be done to avoid overfitting?

A: Introduce a **cost function** that penalizes model complexity. Then, we can train a model that balances two goals:

- (1) The model should have a high accuracy (e.g. low Mean Squared Error)
- (2) The model should have a **low complexity**

How to measure complexity?

Q: How do we measure whether a model is simple or complex?

E.g. can we come up with a measurement that states that a line is a "simpler" model than a high-degree polynomial?

How to measure complexity?

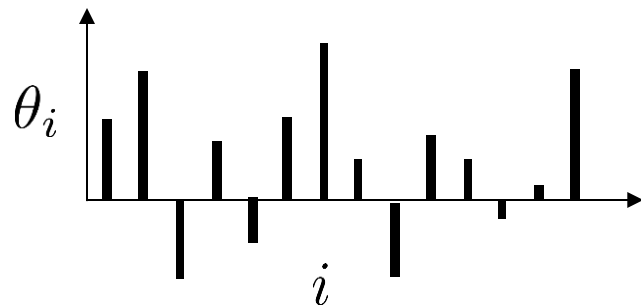
Consider the case of a linear model to describe a polynomial:

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \dots$$

How to measure complexity?

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \dots$$

Then a high-degree polynomial might look like:

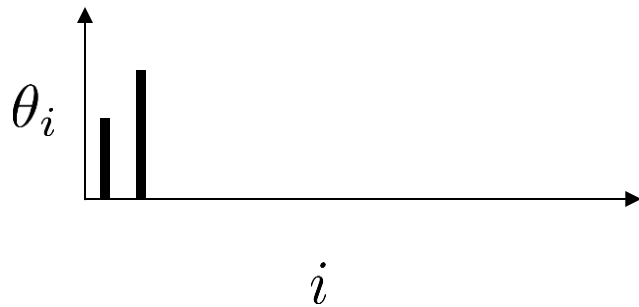


(i.e., most coefficients are non-zero)

How to measure complexity?

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \dots$$

Whereas a line function would look like:

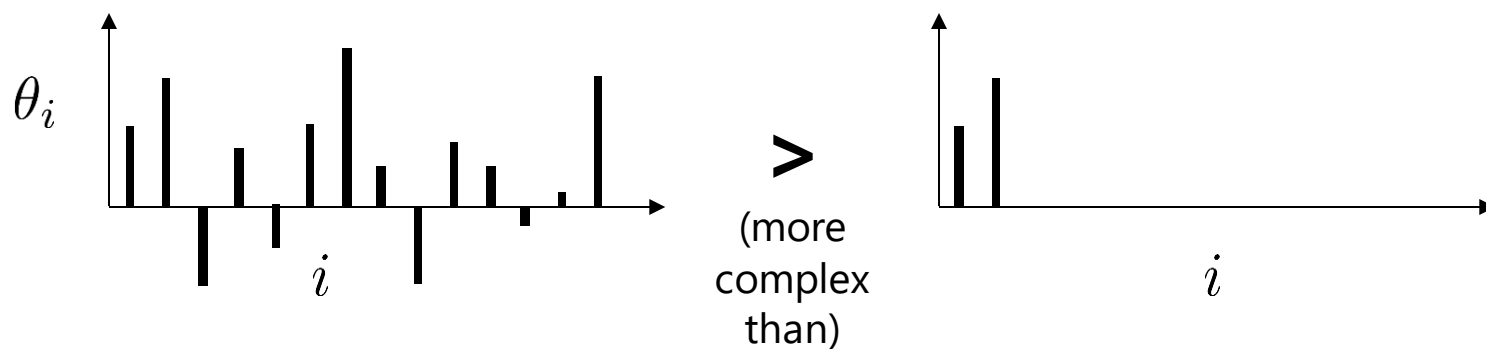


(i.e., only the intercept and slope terms are non-zero)

How to measure complexity?

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \dots$$

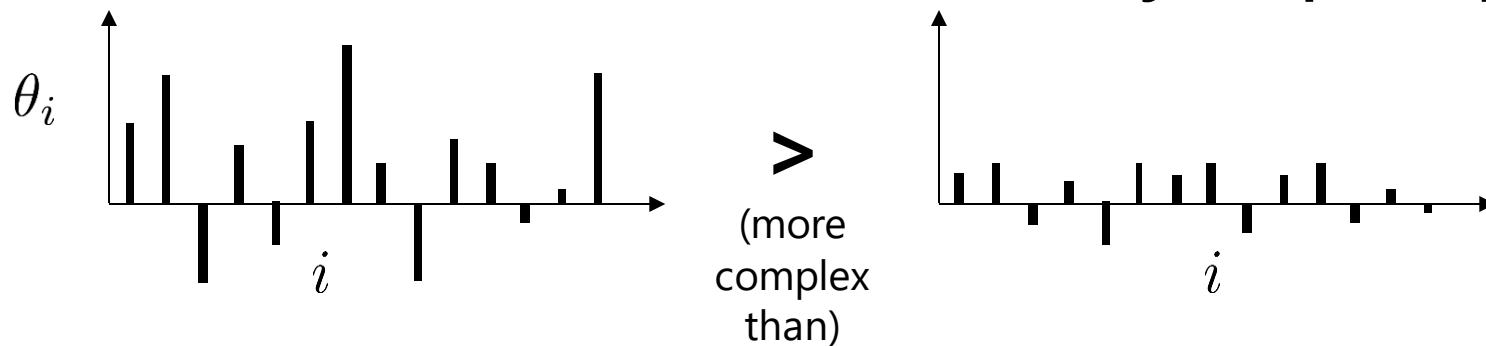
So, one definition might be to count the number of non-zero parameters in theta.



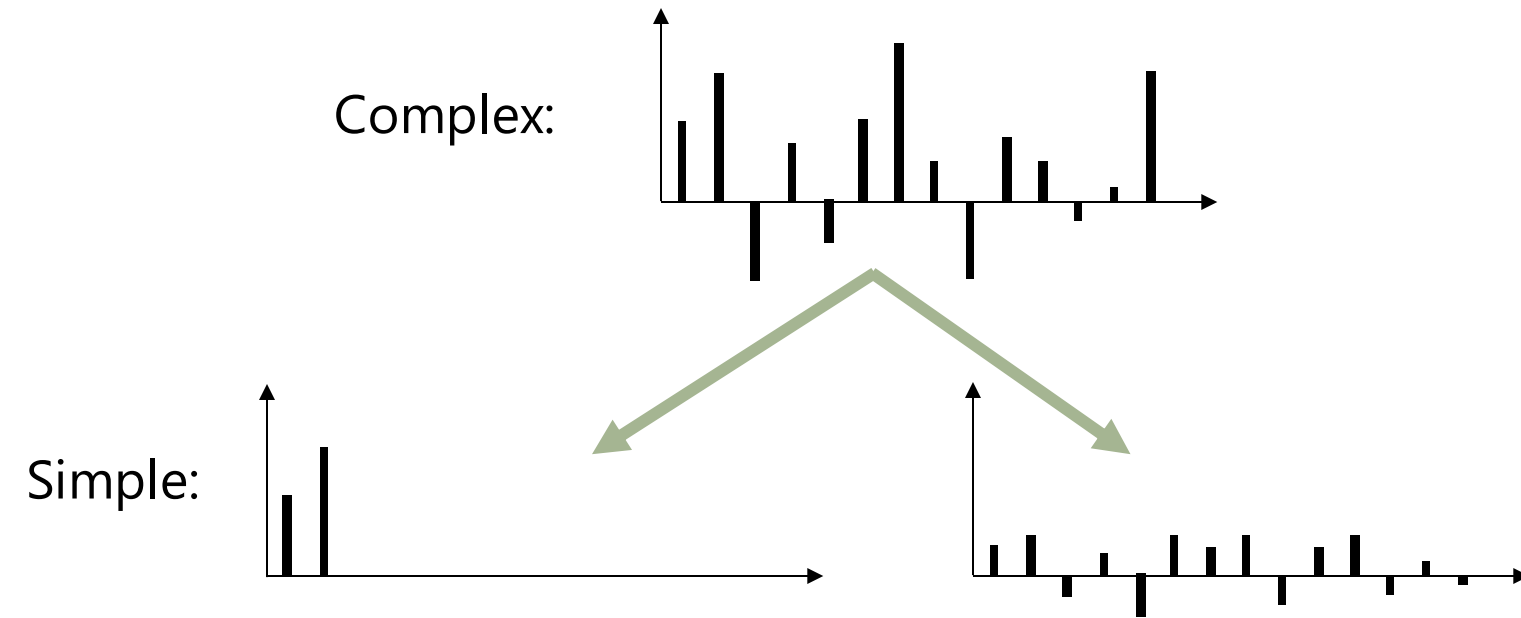
How to measure complexity?

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \dots$$

Another commonly used definition is to count the amount of variance in the parameters, i.e., a "simple" model has mostly equal parameters

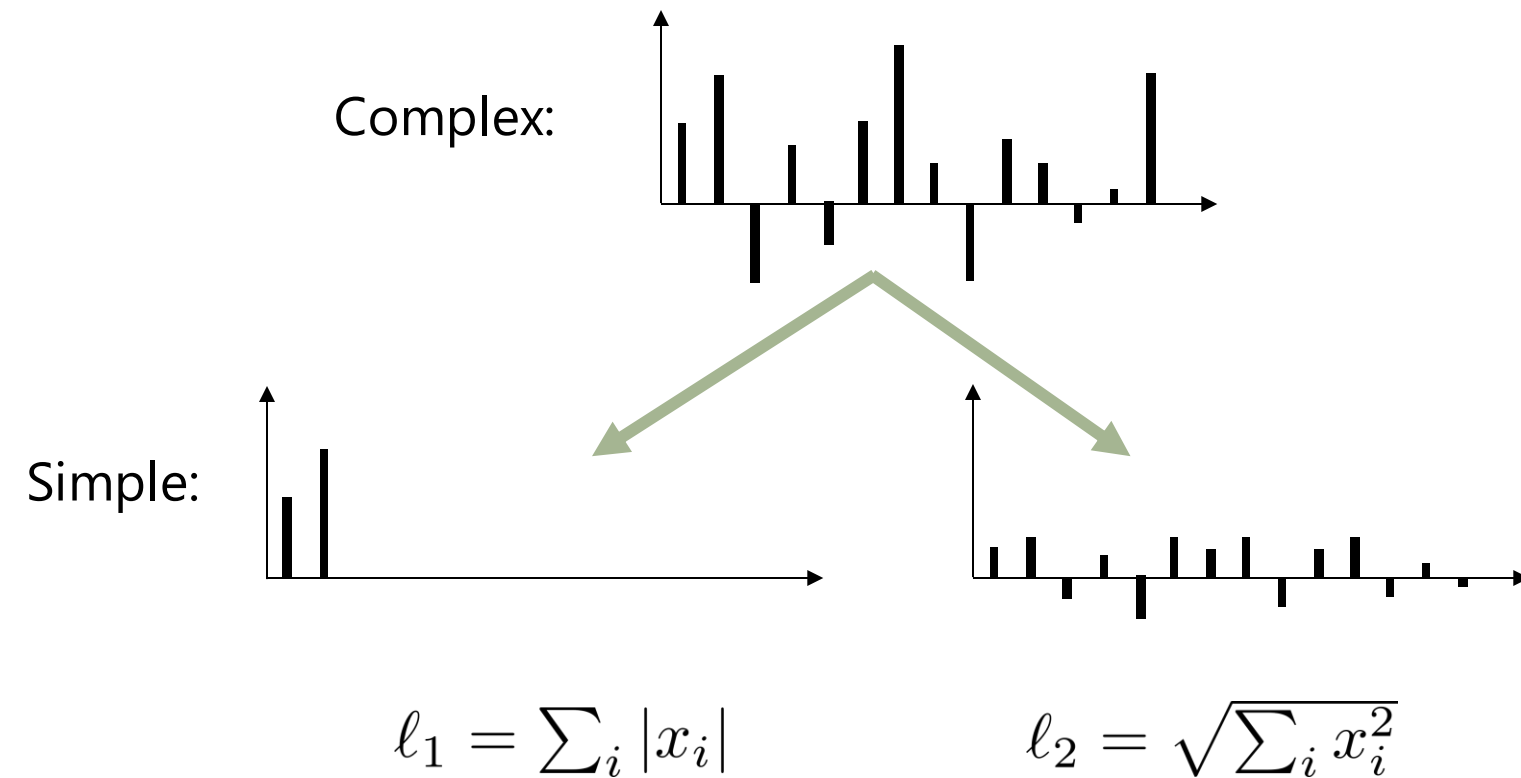


How to measure complexity?



How can we measure (and encourage)
these two notions of simplicity?

How to measure complexity?



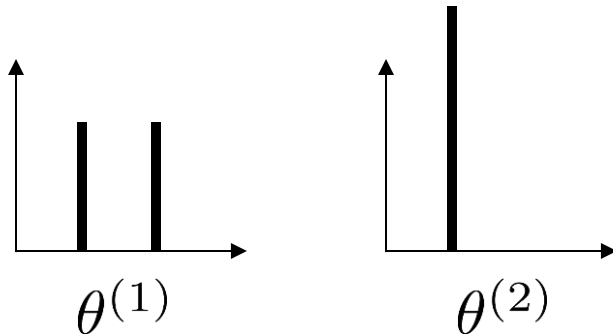
"Theorem": These notions of simplicity will be encouraged by minimizing the L1 and L2 norms

How to measure complexity?

Rough proof...

- Consider a linear model with two highly correlated features:

$$\text{Height} = \theta_0 + \theta_1[\text{gender is female}] + \theta_2[\text{has long hair}]$$



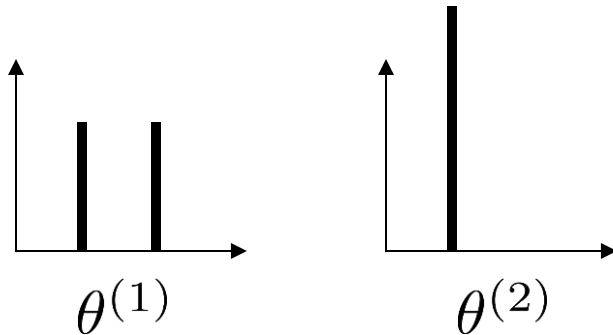
$$\|\theta^{(1)}\|_2 < \|\theta^{(2)}\|_2$$

$$\|\theta^{(1)}\|_1 = \|\theta^{(2)}\|_1$$

How to measure complexity?

Rough proof...

- So, if we penalized the l1-norm, we would select model 2
- If we penalized the l2-norm, we would select model 1



$$\|\theta^{(1)}\|_2 < \|\theta^{(2)}\|_2$$

$$\|\theta^{(1)}\|_1 = \|\theta^{(2)}\|_1$$

How to measure complexity?

So (again roughly speaking)

- If we penalize the l_1 -norm, we will tend to get a model with **sparse** features, with few non-zero terms
- If we penalize the l_2 -norm, we will tend to get a model with mostly **uniform** features, with few large outliers
- We'll see in the following lecture how to incorporate these penalties

Summary of concepts

- Introduced and compared complexity measures for machine learning models
- Explained the consequences behind different choices of complexity measure

On your own...

- Considering one or more of the models you've trained so far, evaluate its complexity in terms of the measures covered in this lecture (l1 and l2 complexity)

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Adding a regularizer to our model, and
evaluating the regularized model

Learning objectives

In this lecture we will...

- Extend our sentiment analysis codebase to incorporate a regularizer
- Demonstrate some of the model performance measures we covered previously

Incorporating a regularizer into our model

The first thing we want to do is to improve our previous model (for sentiment analysis) to include a regularizer:

$$\underbrace{\frac{1}{N} \sum_i (y_i - X_i \cdot \theta)^2}_{\text{MSE}} + \lambda \underbrace{\sum_k \theta_k^2}_{\text{regularizer}}$$

Code example: Regularization

This can be done using the "Ridge" model in sklearn

```
In [26]: from sklearn import linear_model
```

```
In [27]: help(linear_model.Ridge)
```

Help on class Ridge in module sklearn.linear_model.ridge:

```
class Ridge( BaseRidge, sklearn.base.RegressorMixin)
```

Linear least squares with l2 regularization.

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the :ref:`User Guide <ridge_regression>`.

Parameters

alpha : {float, array-like}, shape (n_targets)

Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to C^{-1} in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific

Regularization strength (i.e., lambda)

Code example: Regularization

We can now fit the model much as before:

```
In [28]: model = linear_model.Ridge(1.0, fit_intercept=False)  
         model.fit(X, y)
```

```
Out[28]: Ridge(alpha=1.0, copy_X=True, fit_intercept=False, max_iter=None,  
               normalize=False, random_state=None, solver='auto', tol=0.001)
```



Regularization strength (i.e., lambda)

Code example: Regularization

Again we can extract parameters etc. from the model, which may be slightly different than they were before:

```
In [29]: theta = model.coef_
```

```
In [30]: wordWeights = list(zip(theta, words + ['offset']))  
wordWeights.sort()
```

```
In [31]: wordWeights[:10]
```

```
Out[31]: [(-1.210460068900258, 'disappointing'),  
          (-0.856640197404887, 'disappointed'),  
          (-0.7889876776526171, 'unable'),  
          (-0.6787442786286616, 'waste'),  
          (-0.6621805930969973, 'charged'),  
          (-0.5383370441665155, 'supposed'),  
          (-0.5275057765332417, 'unfortunately'),  
          (-0.49621911813910957, 'tried'),  
          (-0.49620102935875465, 'australia'),  
          (-0.47713054138625355, 'wont')]
```

```
In [32]: wordWeights[-10:]
```

```
Out[32]: [(0.23572772781658063, 'whats'),  
          (0.23820563310858286, 'problems'),  
          (0.24343075578690235, 'particular'),  
          (0.24673282091840637, 'worry'),
```

Model evaluation

Next, let's try to evaluate our model using some of the measures introduced previously

Code example: MSE and R²

Calculating the MSE and R² statistic:

```
In [34]: predictions = model.predict(X)
```

```
In [35]: differences = [(x-y)**2 for (x,y) in zip(predictions,y)]
```

List of squared differences between labels and predictions

```
In [36]: MSE = sum(differences) / len(differences)
print("MSE = " + str(MSE))
```

```
MSE = 0.4260065431778631
```

MSE = average of squared differences

```
In [37]: FVU = MSE / numpy.var(y)
R2 = 1 - FVU
print("R2 = " + str(R2))
```

```
R2 = 0.38057450510836344
```

- FVU = Fraction of Variance Unexplained
 - $R^2 = 1 - \text{FVU}$

Classifier evaluation

To look at some of the **classifier evaluation** measures we previously introduced, we can set the problem up as a classification problem

To do so, rather than estimating the ratings (a regression problem), we'll estimate whether the rating is greater than 3 (a classification problem)

Code example: Setting up a classification problem

Convert the problem to a classification problem, and solve using logistic regression:

```
In [39]: y_class = [(rating > 3) for rating in y]
```

```
In [40]: model = linear_model.LogisticRegression()  
model.fit(X, y_class)
```

```
Out[40]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,  
                             intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,  
                             penalty='l2', random_state=None, solver='liblinear', tol=0.0001,  
                             verbose=0, warm_start=False)
```


Code example: Accuracy

First we can calculate the accuracy of our classifier:

```
In [41]: predictions = model.predict(X)
```

```
In [42]: correct = predictions == y_class
```


List of True/False values
indicating which
predictions were correct



```
In [43]: accuracy = sum(correct) / len(correct)
print("Accuracy = " + str(accuracy))
```

```
Accuracy = 0.9627999946339697
```

Fraction of predictions
that were correct




Code example: True Positives, True Negatives, etc.

Next, using our lists of predictions and labels, we can calculate true positives, true negatives, etc.

```
In [44]: TP = sum([(p and l) for (p,l) in zip(predictions, y_class)])  
FP = sum([(p and not l) for (p,l) in zip(predictions, y_class)])  
TN = sum([(not p and not l) for (p,l) in zip(predictions, y_class)])  
FN = sum([(not p and l) for (p,l) in zip(predictions, y_class)])
```

```
In [45]: print("TP = " + str(TP))  
print("FP = " + str(FP))  
print("TN = " + str(TN))  
print("FN = " + str(FN))
```

```
TP = 138467  
FP = 4445  
TN = 5073  
FN = 1101
```



Note: should add
up to the total size
of the dataset

Code example: True Positives, True Negatives, etc.

Using these counts (TP/FP/TN/FN), we can now compute related statistics like the **accuracy**:

```
In [46]: (TP + TN) / (TP + FP + TN + FN)
```

```
Out[46]: 0.9627999946339697
```

The True Positive **Rate**, and True Negative **Rate** (etc.):

```
In [47]: TPR = TP / (TP + FN)
         TNR = TN / (TN + FP)
```

Or the **Balanced Error Rate**:

```
In [48]: BER = 1 - 1/2 * (TPR + TNR)
         print("Balanced error rate = " + str(BER))
```

```
Balanced error rate = 0.23755431598412025
```

Summary of concepts

- Showed how to adapt our regression code to incorporate a regularizer
- Computed simple statistics (such as the MSE and R^2) on regression data
- Computed several accuracy measures on classification data

On your own...

- Adapt the code to compute other evaluation measures, like the Mean Absolute Error, or the Precision and Recall

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Evaluating classifiers for ranking

Learning objectives

In this lecture we will...

- Extend our classifier from the previous lecture in order to evaluate its ranking performance
- Demonstrate the precision, recall, and F1 ranking measures

Code example: Precision and Recall

Let's start where we left off in the previous lecture. Previously, we had computed values for the number of **True Positives (TP), False Positives, True Negatives, and False Negatives:**

```
In [44]: TP = sum([(p and l) for (p,l) in zip(predictions, y_class)])  
FP = sum([(p and not l) for (p,l) in zip(predictions, y_class)])  
TN = sum([(not p and not l) for (p,l) in zip(predictions, y_class)])  
FN = sum([(not p and l) for (p,l) in zip(predictions, y_class)])
```

Code example: Precision and Recall

First, we can use these values to compute the precision and recall:

```
In [50]: precision = TP / (TP + FP)
```

```
In [51]: recall = TP / (TP + FN)
```

```
In [52]: precision, recall
```

```
Out[52]: (0.9688901639458971, 0.9921113722343231)
```

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

Code example: Precision and Recall

And the F1-score:

```
In [53]: F1 = 2 * (precision*recall) / (precision + recall)
```

```
In [54]: F1
```

```
Out[54]: 0.9803632810702313
```

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

Code example: Sorting scores by confidence

Next we want to sort our predictions by confidence.
First we obtain the **confidences** from the model:

In [55]: `help(model)`

```
-----
Methods inherited from sklearn.linear_model.base.LinearClassifierMixin:
decision_function(self, X)
    Predict confidence scores for samples.

    The confidence score for a sample is the distance from the
    sample to the hyperplane.

    Parameters
    -----
    X : {array-like, sparse matrix}, shape = (n_samples, n_features)
        Samples.

    Returns
    -----
    array, shape=(n_samples,) if n_classes == 2 else (n_samples, n_classes)
        Confidence scores per (sample, class) combination. In the binary
        case, confidence score for self classes [1] where -10 means this
```

Note: confidence
scores are equivalent
to $X_i \cdot \theta$.

Code example: Sorting scores by confidence

Then we sort them along with the labels:

```
In [56]: confidences = model.decision_function(X)
```

```
In [57]: confidences
```

```
Out[57]: array([4.27180659, 5.34068692, 7.88047918, ..., 6.77652788, 5.7457588 ,  
               1.72125511])
```

```
In [58]: confidencesAndLabels = list(zip(confidences,y_class))
```

```
In [59]: confidencesAndLabels
```

```
Out[59]: [(4.271806590458448, True),  
          (5.340686923174397, True),  
          (7.880479181532099, True),  
          (5.224256954963243, True),  
          (6.436088979353579, True),  
          (12.960106079412048, True),  
          (5.318764178069046, True),  
          (6.235572902486643, True),  
          (5.305301082711418, True)]
```

```
In [60]: confidencesAndLabels.sort()  
         confidencesAndLabels.reverse()
```

At this point we can **discard** the confidences:

```
In [62]: labelsRankedByConfidence = [z[1] for z in confidencesAndLabels]
```

```
In [63]: labelsRankedByConfidence
```

[illegible]

Code example: Precision@K and Recall@K

Now we can compute Precision@K and Recall@K values:

```
In [64]: def precisionAtK(K, y_sorted):  
         return sum(y_sorted[:K]) / K
```

```
In [65]: def recallAtK(K, y_sorted):  
         return sum(y_sorted[:K]) / sum(y_sorted)
```

```
In [66]: precisionAtK(50, labelsRankedByConfidence)
```

```
Out[66]: 1.0
```

```
In [67]: precisionAtK(1000, labelsRankedByConfidence)
```

```
Out[67]: 1.0
```

```
In [68]: precisionAtK(10000, labelsRankedByConfidence)
```

```
Out[68]: 0.998
```

Code example: Precision@K and Recall@K

Now we can compute Precision@K and Recall@K values:

```
In [69]: recallAtK(50, labelsRankedByConfidence)
```

```
Out[69]: 0.0003582483090679812
```

```
In [70]: recallAtK(1000, labelsRankedByConfidence)
```

```
Out[70]: 0.007164966181359624
```

```
In [71]: recallAtK(10000, labelsRankedByConfidence)
```

```
Out[71]: 0.07150636248996904
```


Summary of concepts

- Showed how to compute the precision, recall, and F1-score on our sentiment classification example

On your own...

- Adapt the code to compute a **precision-recall curve**, i.e., plot `precision@k` and `recall@k` values for each value of `k`

Week 3

Implementing the Predictive Pipeline

Python Data Products

Course 1: Basics

Lecture: Validation

Learning objectives

In this lecture we will...

- Introduce the concept of the **validation set**
- Explain the relationship between model **parameters** and **hyperparameters**
- Introduce the **training → validation → test** pipeline

Recap...

In the last few lectures we saw...

- How a **training set** can be used to evaluate model performance on seen data
- How a **test set** can be used to estimate **generalization performance**
- How we can use a **regularizer** to mitigate overfitting

Recap...

In particular, our **regularizer** "trades-off" between model accuracy and model complexity

$$\underbrace{\frac{1}{N} \sum_i (y_i - X_i \cdot \theta)^2}_{\text{MSE}} + \lambda \underbrace{\sum_k \theta_k^2}_{\text{regularizer}}$$

- We want a value of our regularization parameter that balances model accuracy (low MSE) with complexity (low sum of squared parameters)

Recap...

In particular, our **regularizer** "trades-off" between model accuracy and model complexity

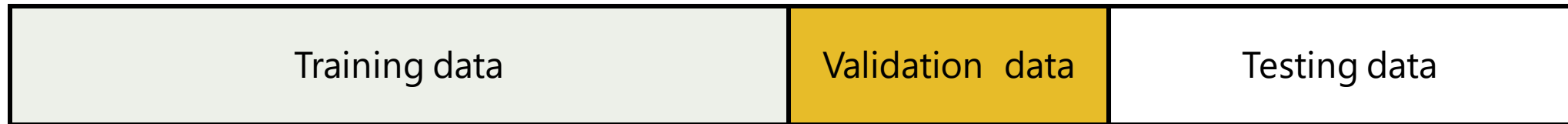
- If we only cared about **training error**, we'd always select the smallest possible value of λ (i.e., $\lambda = 0$)
- We could tune against our **test set**, but that would mean looking at the test set many times (which would be cheating!)

Recap...

In particular, our **regularizer** "trades-off" between model accuracy and model complexity

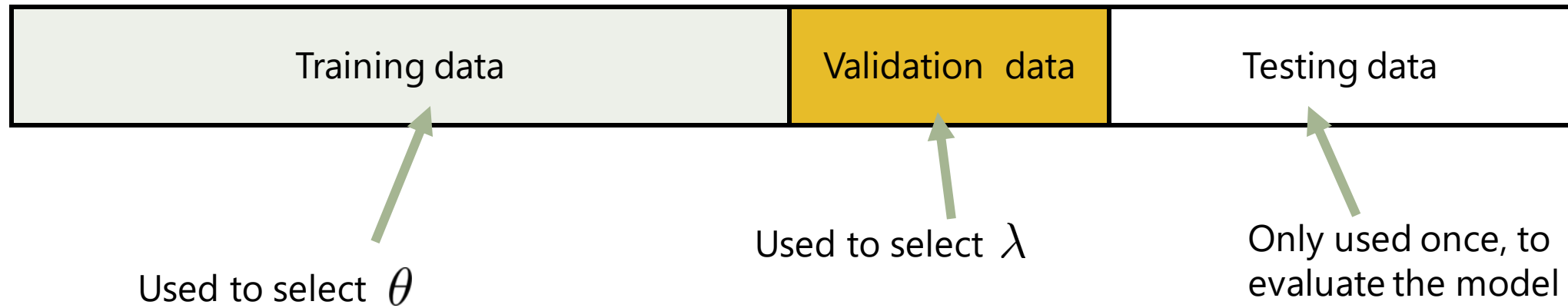
- So, we need a third partition of our data, which is similar to the test set, but which can be used to select hyperparameters like λ
 - This set is called the **validation set**

Training and test sets



A diagram illustrating the matrix notation for training, validation, and test data. On the left, a large square bracket contains the letter X . To the right of this bracket is the equation $\theta =$ followed by a vertical square bracket containing the letter y . Three green arrows point from the labels 'train', 'validation', and 'test' to the top, middle, and bottom rows of the X matrix, respectively.

Training and test sets



Summary of concepts

- We showed how a **validation set** can be used to tune parameters (or “hyperparameters”) that cannot be selected using the training set (or the test set)
- In the following lecture, we’ll explore more how this set can be used to optimize model performance

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: "Theorems" about Training, Testing,
and Validation

Learning objectives

In this lecture we will...

- Explain the relationship between training, validation, and test performance
- Introduce several "theorems" characterizing these relationships
- Briefly describe how these theorems can be used to guide model selection

Recap...

Previously we saw the relationship between training, validation, and test sets, and described how the validation set should be used to select hyperparameters

In practice, how do training, validation, and test errors relate to each other, and how do we select the best model?

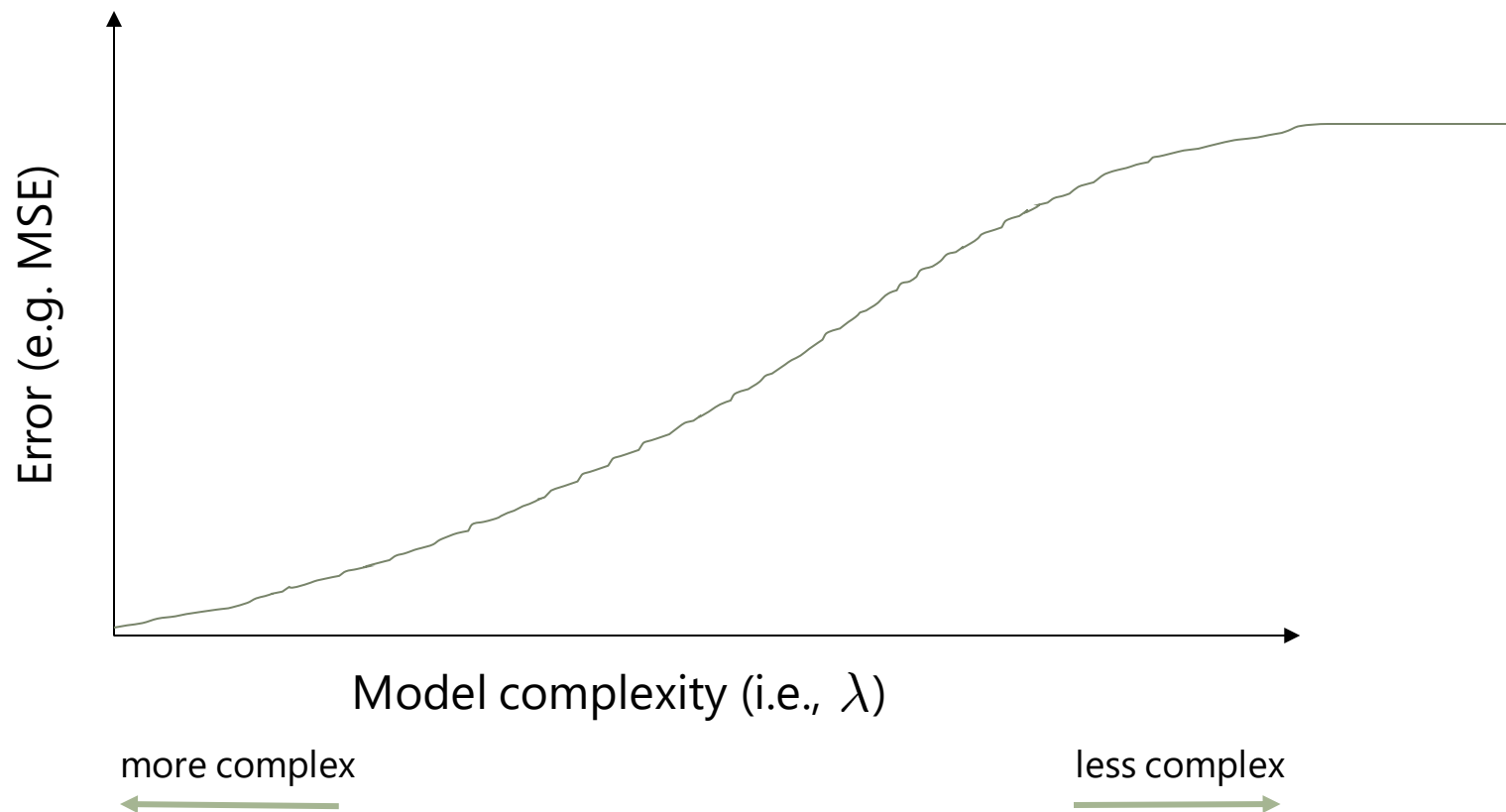
Recap...

Our basic setup consists of

- A series of **hyperparameters** (e.g. values of λ) to experiment with
- One model for each hyperparameter combination
- A training, validation, and test set on which to evaluate performance

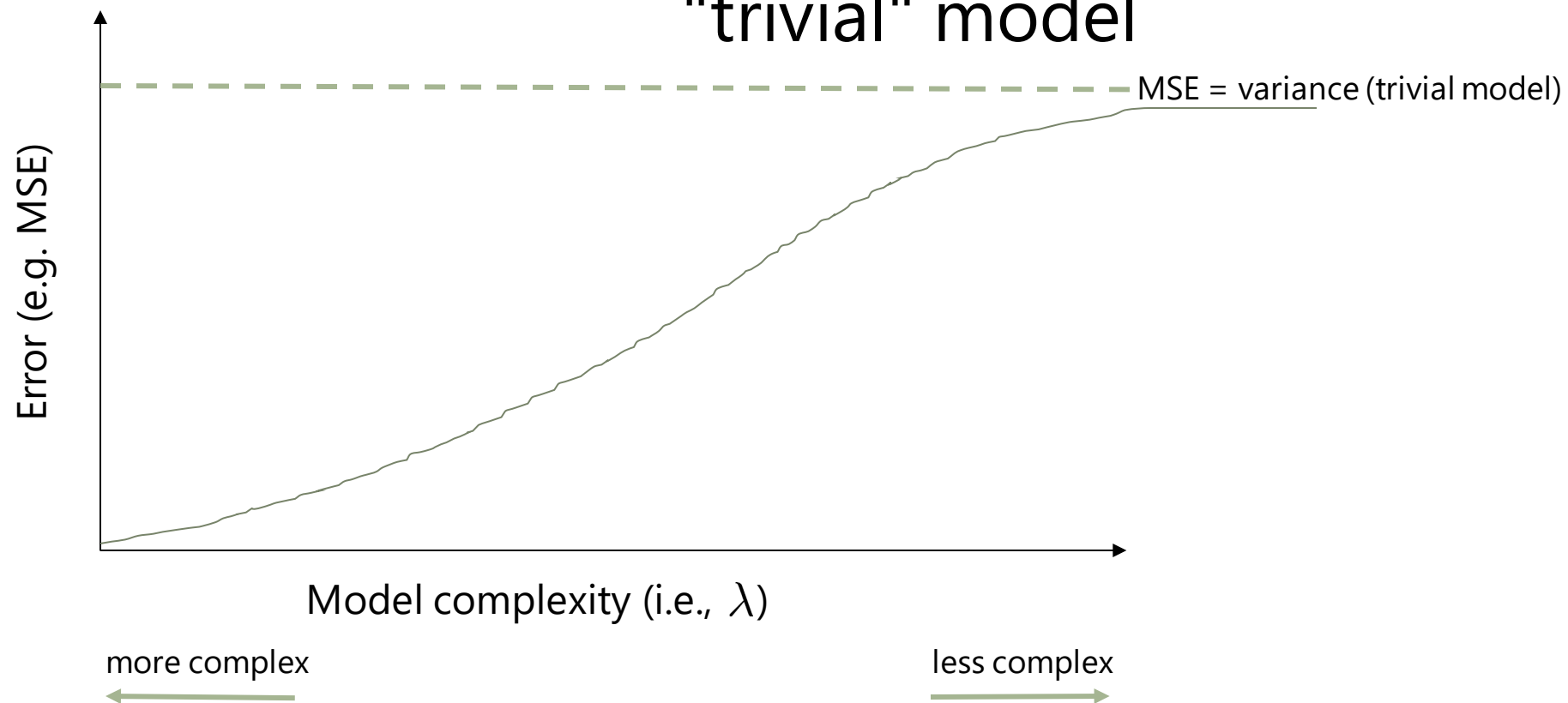
Theorems about training, testing, and validation

1. Error should **increase** as lambda **increases**



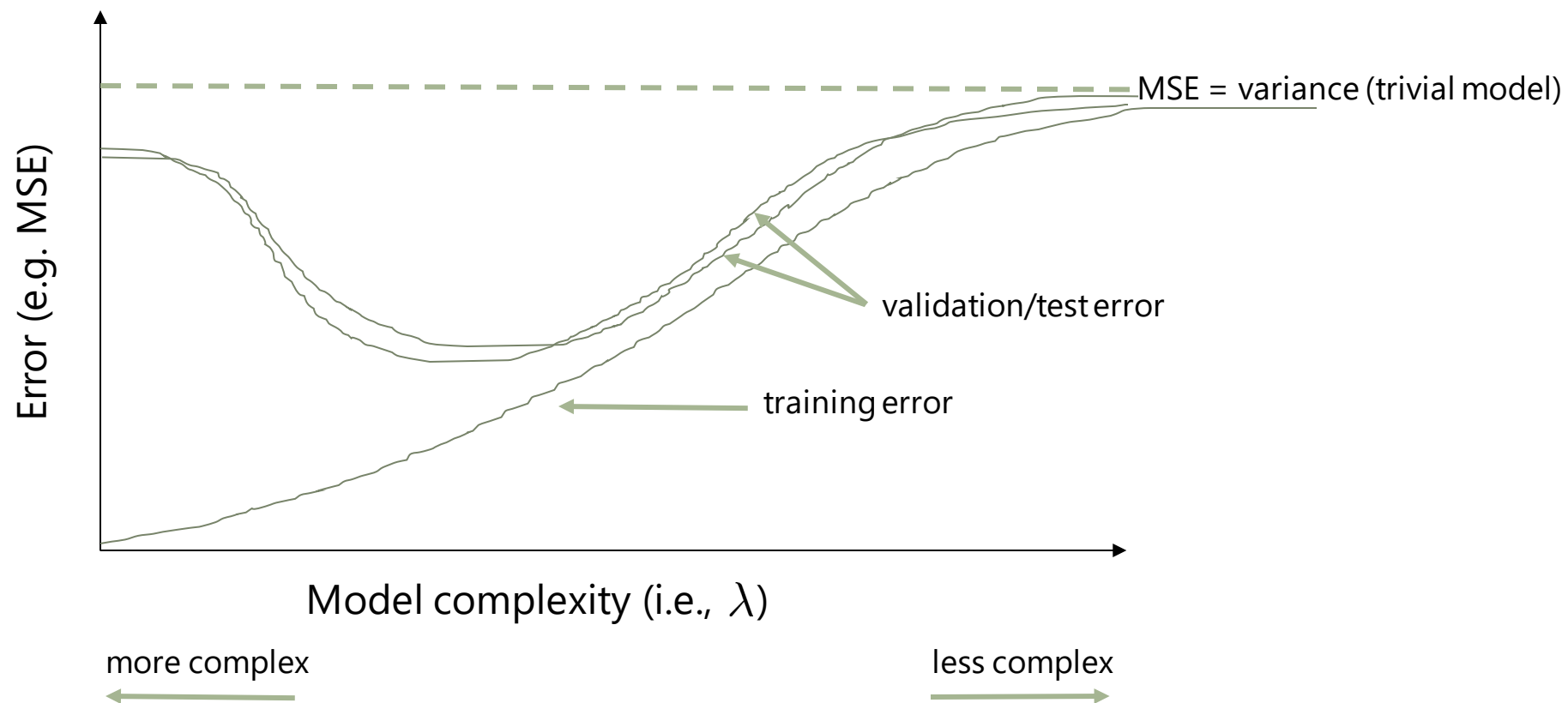
Theorems about training, testing, and validation

1(b). For large lambda, the model will generally behave like a "trivial" model



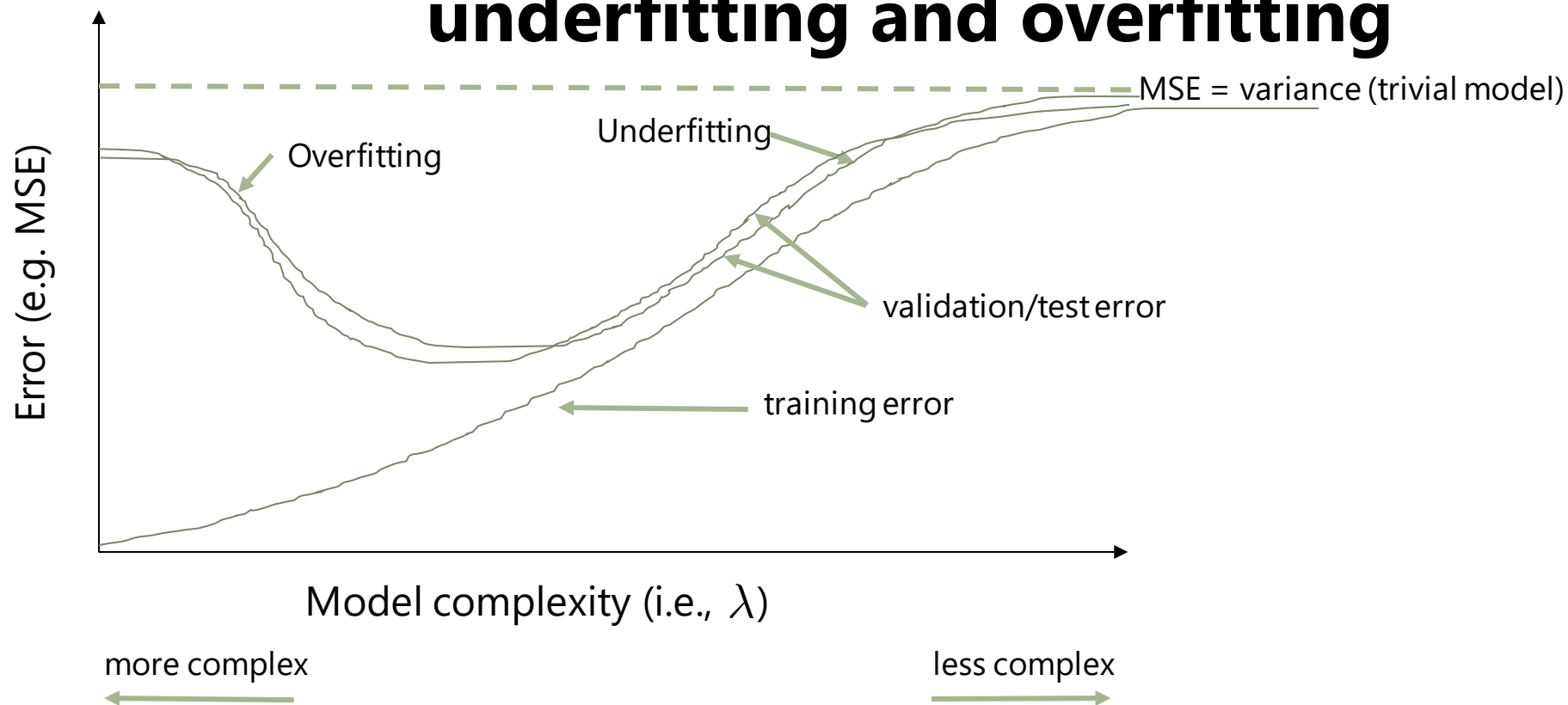
Theorems about training, testing, and validation

2. Validation and test error will be larger than training error



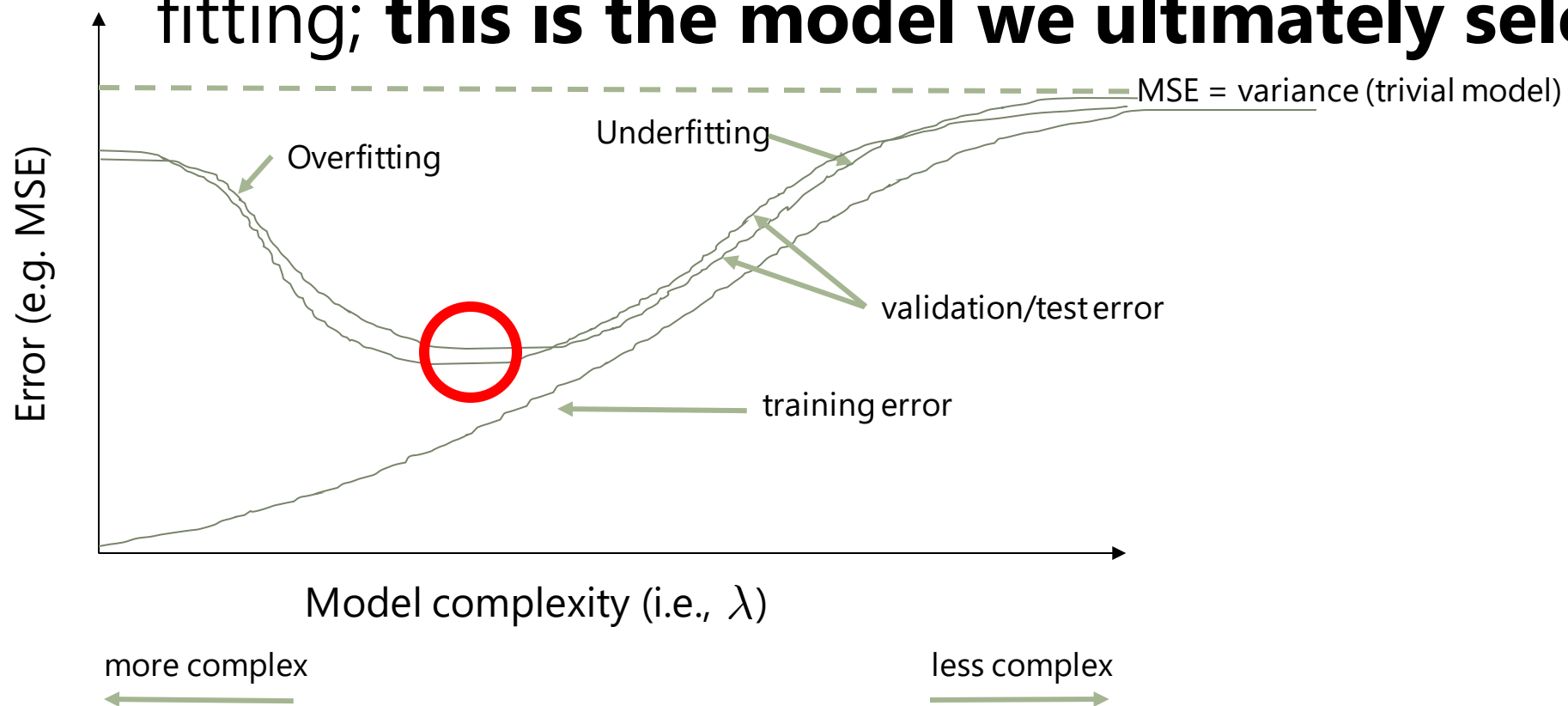
Theorems about training, testing, and validation

2(b). The validation error can be used to identify **underfitting and overfitting**



Theorems about training, testing, and validation

3. There will usually be a "sweet spot" between under- and over-fitting; **this is the model we ultimately select**



Theorems about training, testing, and validation

1. Error should **increase** as lambda **increases**
2. Validation and test error will be larger than training error
3. There will usually be a "sweet spot" between under- and over-fitting

Theorems about training, testing, and validation

Note about these "theorems":

- Due to the randomness of real datasets, these theorems may not always hold precisely
- However they are good guidelines – if these theorems are badly violated it could be a sign of a bug
 - They should hold assuming we have **large** datasets, and assuming that our training, validation, and test sets are **randomly sampled**
 - Note that if we were **maximizing accuracy** (rather than minimizing error), this plot would be inverted

Validation pipeline

So, the validation pipeline looks roughly as follows
(to be described in more detail in later lectures):

- Select a range of values for lambda (I usually use powers of 10)
 - For each value of lambda train a model, and compute its **validation error**
- Select the model with the lowest validation error (or highest accuracy), and compute its performance on the test set

Summary of concepts

- Introduced several theorems characterizing the relationship between training, validation, and test performance
- Showed how these concepts might be used within a training/validation/testing pipeline

On your own...

- Try computing training/validation/test error on one of our previous examples (e.g. our example on bankruptcy data), and experimentally confirm the relationships described in this lecture

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Implementing a regularization
pipeline in Python

Learning objectives

In this lecture we will...

- Show how to implement the training/validation/testing pipeline in Python
- Show how to select regularization parameters, and evaluate model performance using a validation set

Validation pipeline

To summarize our validation pipeline so far, our goal is to:

- Split the data into train/validation/test fractions
- Consider several different values of our hyperparameters (e.g. λ)
- For each of these values, train a model on the **training set**
- Evaluate each model's performance on the **validation set**
- For the model that performs best on the validation set, evaluate its performance on the **test set**

Code: data and problem setup

First, let's set up our prediction problem (which is mostly code we've seen before):

```
In [49]: # Training / validation / test pipeline
```

```
In [50]: import gzip
         from collections import defaultdict
         import string
         import random
```

```
In [51]: path = "/home/jmcauley/datasets/mooc/amazon/amazon_reviews_us_Gift_Card_v1_00.tsv.gz"
```


```
In [52]: f = gzip.open(path, 'rt', encoding="utf8")
```

```
In [53]: header = f.readline()
         header = header.strip().split('\t')
```

```
In [54]: dataset = []
```

```
In [55]: for line in f:
         fields = line.strip().split('\t')
         d = dict(zip(header, fields))
         d['star_rating'] = int(d['star_rating'])
         d['helpful_votes'] = int(d['helpful_votes'])
         d['total_votes'] = int(d['total_votes'])
         dataset.append(d)
```

Read the data, and
convert numerical
values to integers



Code: data and problem setup

Next we extract features (again, much as we did in previous examples):

```
In [57]: wordCount = defaultdict(int)
punctuation = set(string.punctuation)


for d in dataset:
    r = ''.join([c for c in d['review_body'].lower() if not c in punctuation])
    for w in r.split():
        wordCount[w] += 1

counts = [(wordCount[w], w) for w in wordCount]
counts.sort()
counts.reverse()

words = [x[1] for x in counts[:1000]]

wordId = dict(zip(words, range(len(words))))
wordSet = set(words)
```

Counting instances
of words in each
review



```
In [58]: def feature(datum):
    feat = [0]*len(words)
    r = ''.join([c for c in datum['review_body'].lower() if not c in punctuation])
    for w in r.split():
        if w in words:
            feat[wordId[w]] += 1
    feat.append(1) #offset
    return feat
```

Code: splitting the data

Now, our first task is to split the data into training, validation, and test samples:

```
In [59]: random.shuffle(dataset)
```

```
In [60]: X = [feature(d) for d in dataset]
```

```
In [61]: y = [d['star_rating'] for d in dataset]
```

```
In [62]: N = len(X)
X_train = X[:N//2]
X_valid = X[N//2:3*N//4]
X_test = X[3*N//4:]
y_train = y[:N//2]
y_valid = y[N//2:3*N//4]
y_test = y[3*N//4:]
```

```
In [63]: len(X), len(X_train), len(X_valid), len(X_test)
```

```
Out[63]: (149086, 74543, 37271, 37272)
```

Remember to **shuffle** the dataset, so that our train/valid/test sets are i.i.d. samples

This example uses 50%/25%/25% (non-overlapping) splits, though other ratios would also be possible

Code: regression model

Again, we'll use the "Ridge" model from sklearn, which allows us to implement regression with a regularizer

```
In [64]: from sklearn import linear_model
```

```
In [65]: help(linear_model.Ridge)
```

Help on class Ridge in module sklearn.linear_model.ridge:

```
class Ridge(_BaseRidge, sklearn.base.RegressorMixin)
```

Linear least squares with l2 regularization.

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the :ref:`User Guide <ridge_regression>`.

Parameters

alpha : {float, array-like}, shape (n_targets)

Small positive values of alpha improve the conditioning of the problem and reduce the variance of the estimates. Alpha corresponds to C^{-1} in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific

Regularization parameter



Code: regression model

Set up a quick utility function to calculate the MSE for a particular model:

```
In [66]: def MSE(model, X, y):  
         predictions = model.predict(X)  
         differences = [(a-b)**2 for (a,b) in zip(predictions, y)]  
         return sum(differences) / len(differences)
```


Code: regression model

Train the model for a range of regularization parameters:

```
In [67]: bestModel = None  
bestMSE = None
```

Keep track of which model worked the best

```
In [68]: for lamb in [0.01, 0.1, 1, 10, 100]:  
    model = linear_model.Ridge(lamb, fit_intercept=False)  
    model.fit(X_train, y_train)  
  
    mseTrain = MSE(model, X_train, y_train)  
    mseValid = MSE(model, X_valid, y_valid)  
  
    print("lambda = " + str(lamb) + ", training/validation error = " +  
          str(mseTrain) + '/' + str(mseValid))  
    if not bestModel or mseValid < bestMSE:  
        bestModel = model  
        bestMSE = mseValid
```

Fit a model for each lambda value

Report the training and validation error (but not the test error yet!)

```
lambda = 0.01, training/validation error = 0.4187297059927889/0.4481159192463995  
lambda = 0.1, training/validation error = 0.41872971449864577/0.44810067260735315  
lambda = 1, training/validation error = 0.41873055597795/0.44795079625281725  
lambda = 10, training/validation error = 0.41880648152201755/0.44668116707720923  
lambda = 100, training/validation error = 0.42244982510706414/0.4437512972171302
```

Code: regression model

Finally, report the **test error** for the model that had the best performance on the **validation set**

```
In [69]: mseTest = MSE(bestModel, X_test, y_test)
print("test error = " + str(mseTest))

test error = 0.44550653361941783
```

Summary of concepts

- Showed a full pipeline of model selection and evaluation on a real dataset

On your own...

- Reproduce this pipeline for a different task, e.g. for a classification experiment

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Guidelines on the implementation of
predictive pipelines

Learning objectives

In this lecture we will...

- Suggest practical guidelines for model selection
- Show how our “theorems” about model selection can be applied in practice
- Demonstrate other cases where the validation set can be used, besides selection regularization parameters

Choosing among several different models

1. As well as selecting model hyperparameters (like λ) the validation set can also be used to select among model alternatives. E.g.:

- Should I use an SVM or a logistic regressor?
 - How deep should my decision tree be?
- How many layers should my neural network have?

Choosing among several different models

2. When using iterative models (like gradient descent/ascent), it is not necessary (or desirable) to train the model to convergence

Rather, we should periodically compute the validation error, and **stop once the validation error is no longer improving**

Choosing among several different models

3. The validation set can also be used to guide feature engineering choices. E.g.:

- How many words should we include in our dictionary?
In our example we used 1000 but we could make a better choice using our validation set
- Should we remove punctuation, capitalization, etc.?

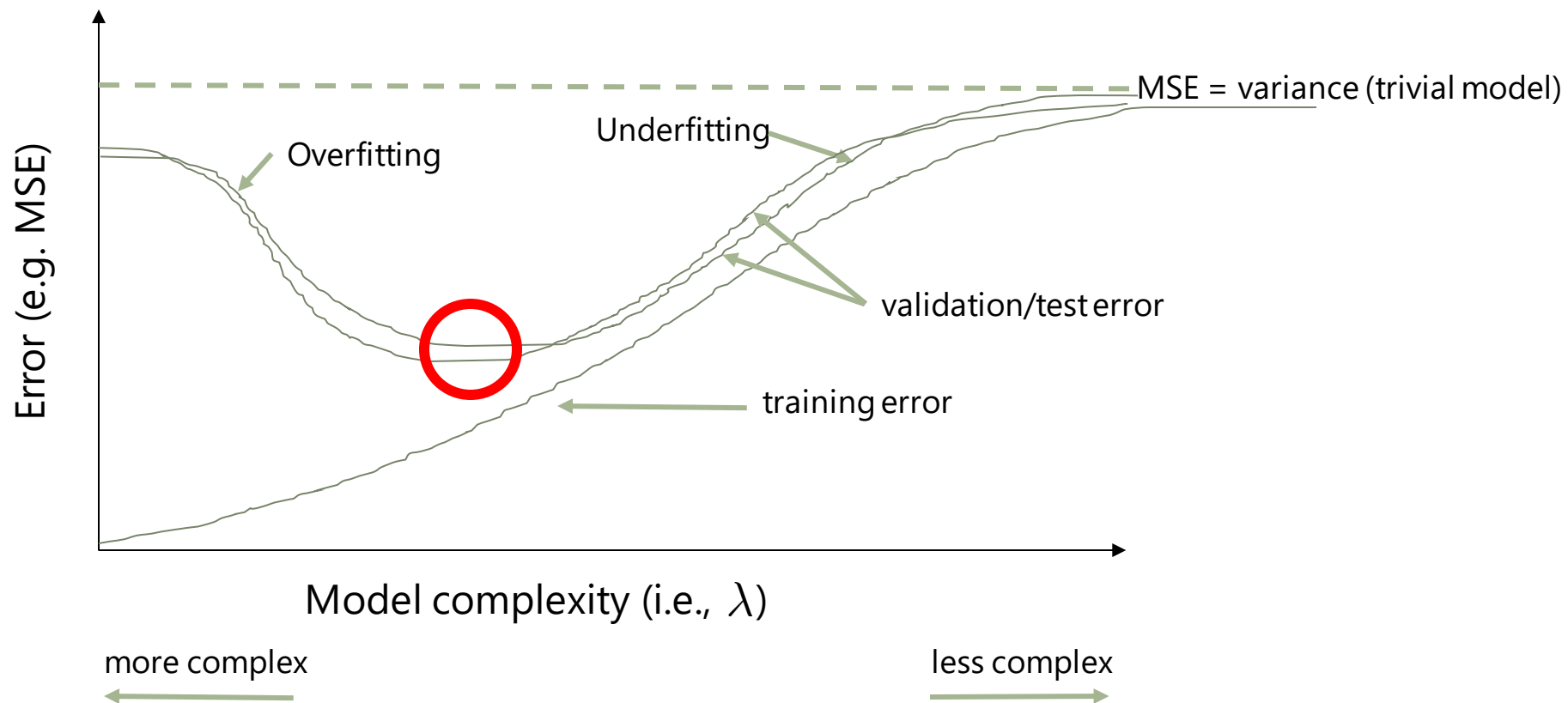
Choosing among several different models

4. What values of lambda should we choose

- Our validation "theorems" can guide us to good choices of values
- E.g. given two values of lambda (a and $b > a$), if the **validation error** is higher for a , then we should try **larger** values than b ; if the validation error is higher for b , we should try **smaller** values than a

Choosing among several different models

4. What values of lambda should we choose



Summary of concepts

- Gave practical advice as to how to select regularization parameters
- Showed how the same concepts can be used to guide other modeling decisions (e.g. different feature representation options)

On your own...

- Use these guidelines to optimize various model parameters, e.g. the dictionary size, text-processing options, etc., as well as the hyperparameter λ

Python Data Products

Course 3: Making Meaningful Predictions from Data

Lecture: Course Summary

Course summary: Week 1

In this course we covered...

- Evaluation metrics for **regression** algorithms: MSE, R^2 , and the motivation behind them
- Evaluation metrics for **classification** algorithms: Accuracy, Error, Precision, Recall, etc.
- How to select the **right** evaluation metrics under different conditions

Course summary: Week 2

In this course we covered...

- **Predictive pipelines:** how do we build models that will generalize well to new data?
 - Model complexity and overfitting
 - Regularization
 - How can we train models that will not be too complex?
I.e., how can we trade-off accuracy versus complexity?

Course summary: Week 3

In this course we covered...

- Implementation of the predictive pipeline in Python
- Rules and guidelines for model selection using training/validation/test sets

Course summary

By now you should be able to...

- Implement **robust** predictive pipelines in Python. In the previous course, we saw the same basic pipelines, but by now you should be able to:
 - a) Select appropriate **evaluation metrics** based on sound principles
 - b) Ensure that your models are not too complex, by using a **regularizer**
 - c) Ensure that your models will **generalize** well to new data

Course summary

In the **next course** we will...

- Introduce **recommender systems**, to be used for our capstone project
- See how to implement simple, working recommender systems on real datasets, and deal with the various implementation issues involved with doing so
- Combine the knowledge that we have developed so far to deal with various issues around evaluation, model fitting, implementation, and deployment