Introduction to Machine Learning (IML)

LECTURE #8: PRACTICAL ASPECTS

236756 – 2023-2024 WINTER – TECHNION

LECTURER: YONATAN BELINKOV

Today

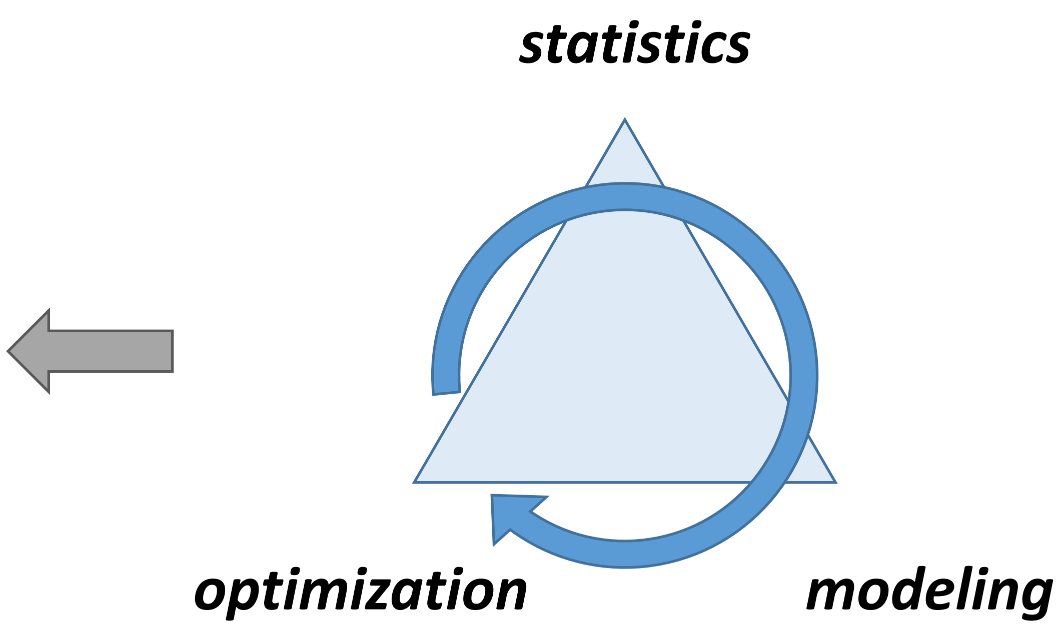
- **Part II**: *the different aspects of learning*
    1. Statistics: generalization and PAC theory
    2. Modeling:
        - Error decomposition
        - Regularization
        - Model selection
    3. Optimization: convexity, gradient descent
    4. Practical aspects and potential pitfalls

Tying it all together

Interim

<u>Template</u>:

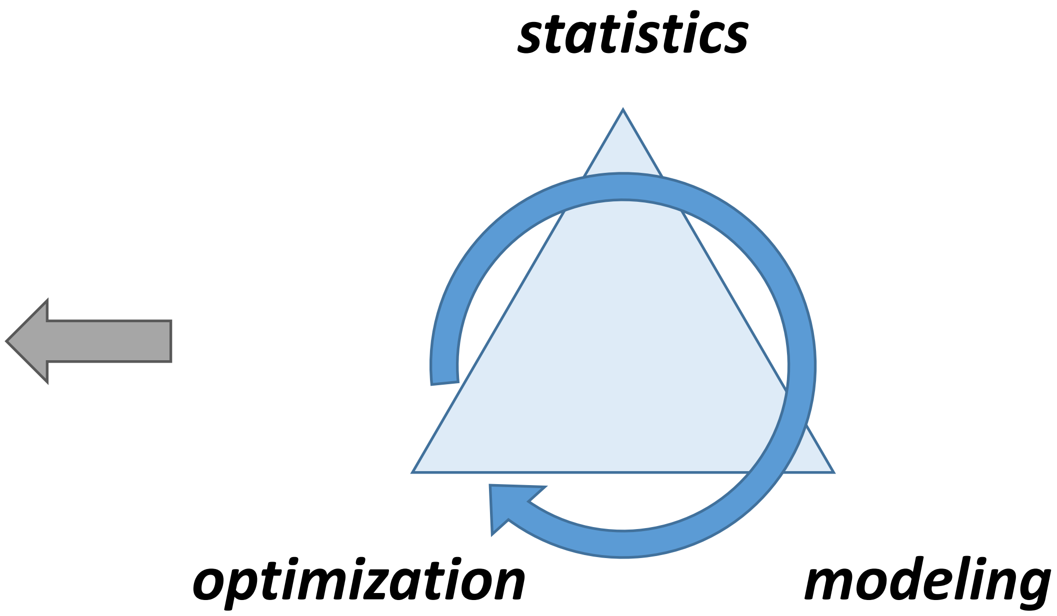
1. choose model class
2. set learning objective
3. optimize



Interim

<u>**Template**:</u>

1. choose model class: linear
2. set learning objective: hinge + L_2 reg.
3. optimize: SGD



- **Let's code it up!**

```
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’)
model = optimizer.train(x, y, objective)
yhat = model.predict(x)
err = error(y, yhat)
print(err)
err: 0.05
```

are we done?

no: we care about expected error, not empirical error

```
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model='linear', loss='hinge', reg='l2')
optimizer = Optimizer(algo='SGD')
model = optimizer.train(x_trn, y_trn, objective)
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
print(err_trn, err_tst)
err_trn: 0.05
err_tst: 0.22
```

are we done?

no: something is causing this discrepancy – what?

used default param!

```
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model='linear', loss='hinge', reg='l2', lambda=1)
optimizer = Optimizer(algo='SGD')
model = optimizer.train(x_trn, y_trn, objective)
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
print(err_trn, err_tst)
err_trn: 0.05
err_tst: 0.22
```

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_val, y_val, x_tst, y_tst = split(x, y, [0.6, 0.2, 0.2])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’)
for i,lam in enumerate(lambdas):
    model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam))
    yhat_val = model.predict(x_val)
    errs_val[i] = error(y_tval, yhat_val)
lam_opt = lambdas[argmin(errs_val)]
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_tv, yhat_tst = model.predict([x_trn,x_val]), model.predict(x_tst)
err_tv, err_tst = error([y_trn,y_val], yhat_tv), error(y_tst, yhat_tst)
print(err_tv, err_tst)
err_trn: 0.09
err_tst: 0.06
```

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’)
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
print(err_trn, err_tst)
err_trn: 0.09
err_tst: 0.06
```

are we done?

no: estimating expected performance using a single split is noisy

```
lambdas = [10^-5,…,10^5]
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
    objective = Objective(model='linear', loss='hinge', reg='l2')
    optimizer = Optimizer(algo='SGD')
    lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
    model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
    yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
    errs_trn[i], errs_tst[i] = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
print(mean(errs_trn), mean(errs_tst))
err_trn: 0.10
err_tst: 0.13
```

are we done?

no: remember – default parameters...

```
lambdas = [10^-5,…,10^5]
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
    objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
    optimizer = Optimizer(algo=‘SGD’, lr=0.01, T=1000)
    lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
    model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
    yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
    errs_trn[i], errs_tst[i] = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
print(mean(errs_trn), mean(errs_tst))
err_trn: 0.07
err_tst: 0.09
```

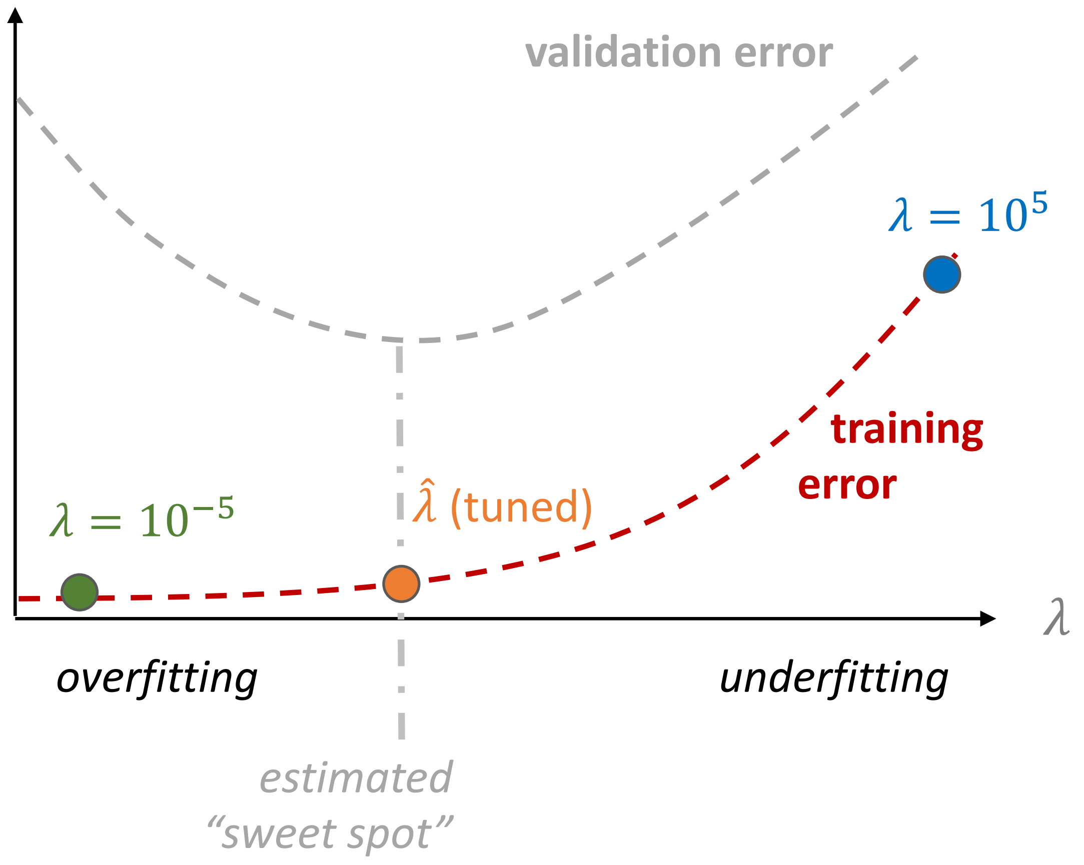
finally… done!

Choices, choices

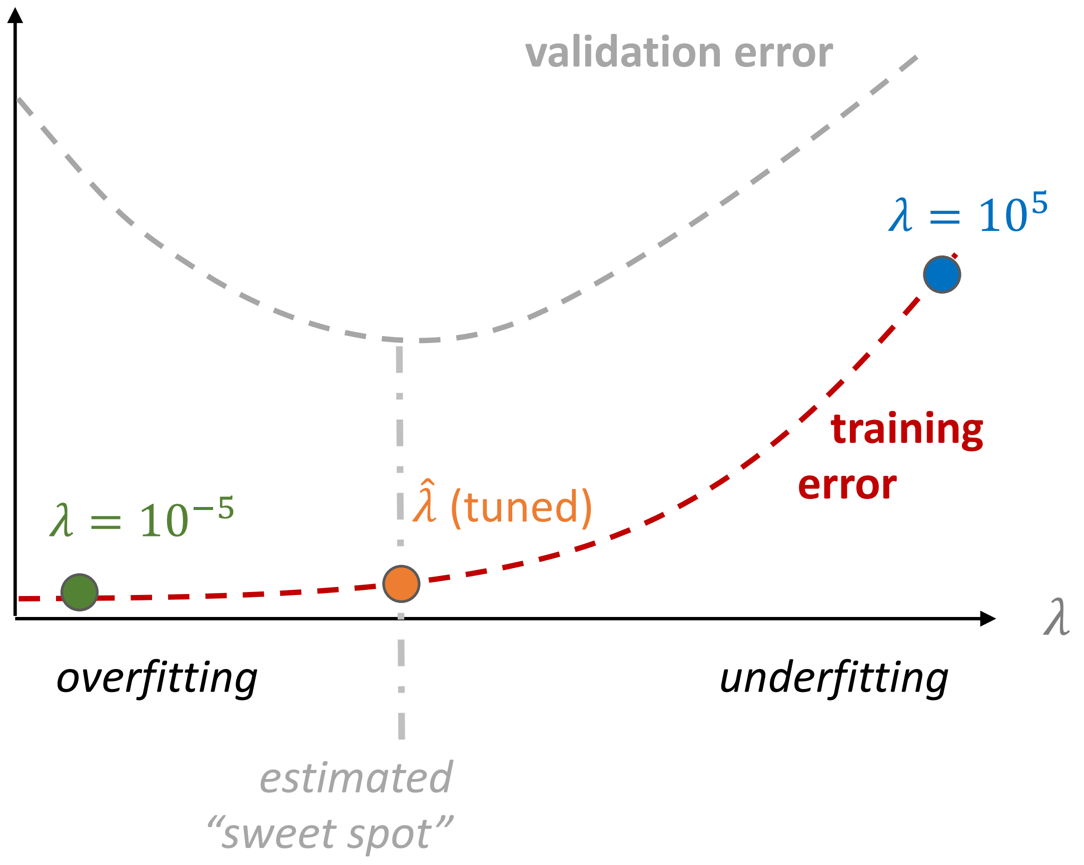
- How should we choose `lr` and `T`?
- Answer turns out to be quite elaborate
- We will explore this through three seemingly unrelated aspects:
  1. regularization
  2. early stopping
  3. feature scaling (preprocessing)

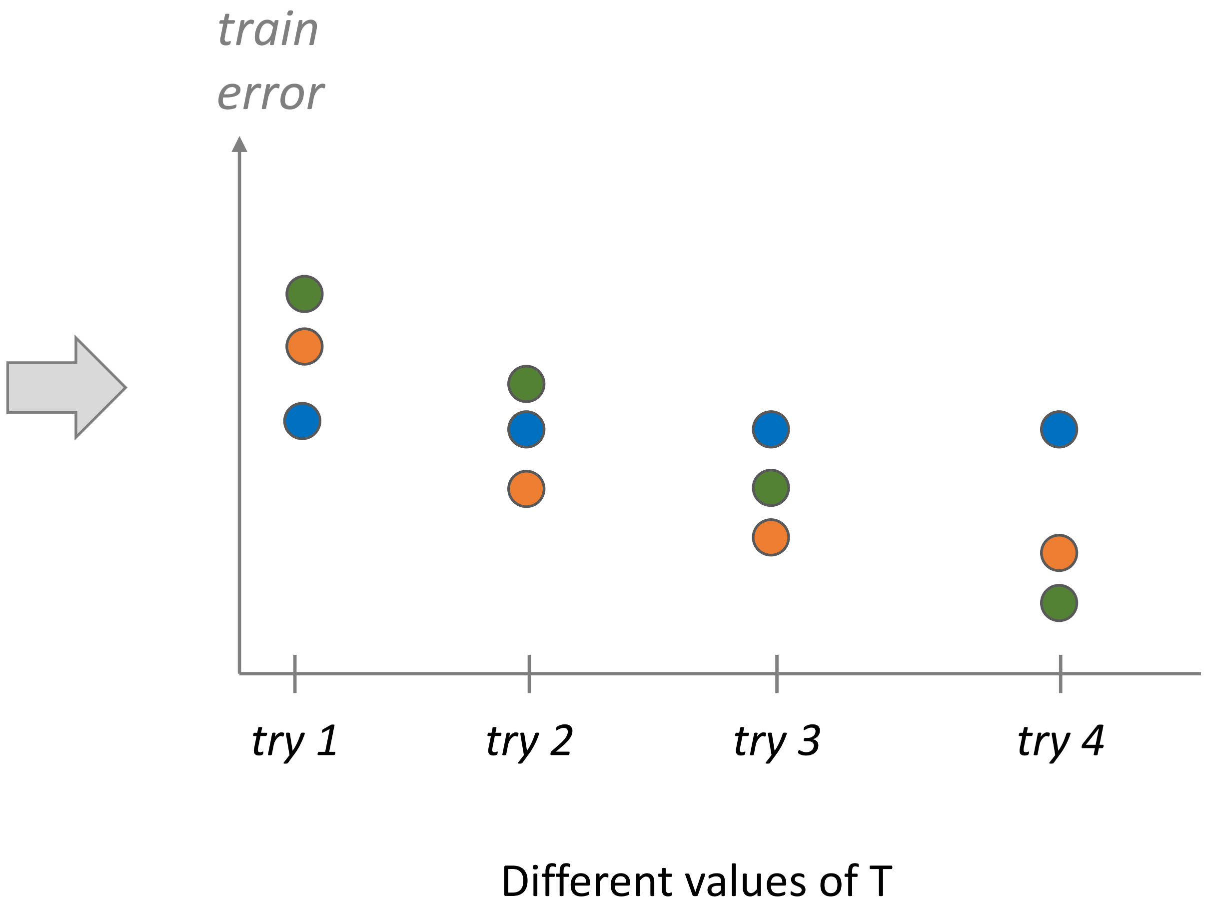
- What about the batch size b?
  - Convex case: Larger `b` → better gradient estimation → use larger `lr`
  - Non-convex: no guarantees, but (some theory, mostly heuristics): $lr \propto b$ or $lr \propto \sqrt{b}$

Regularization

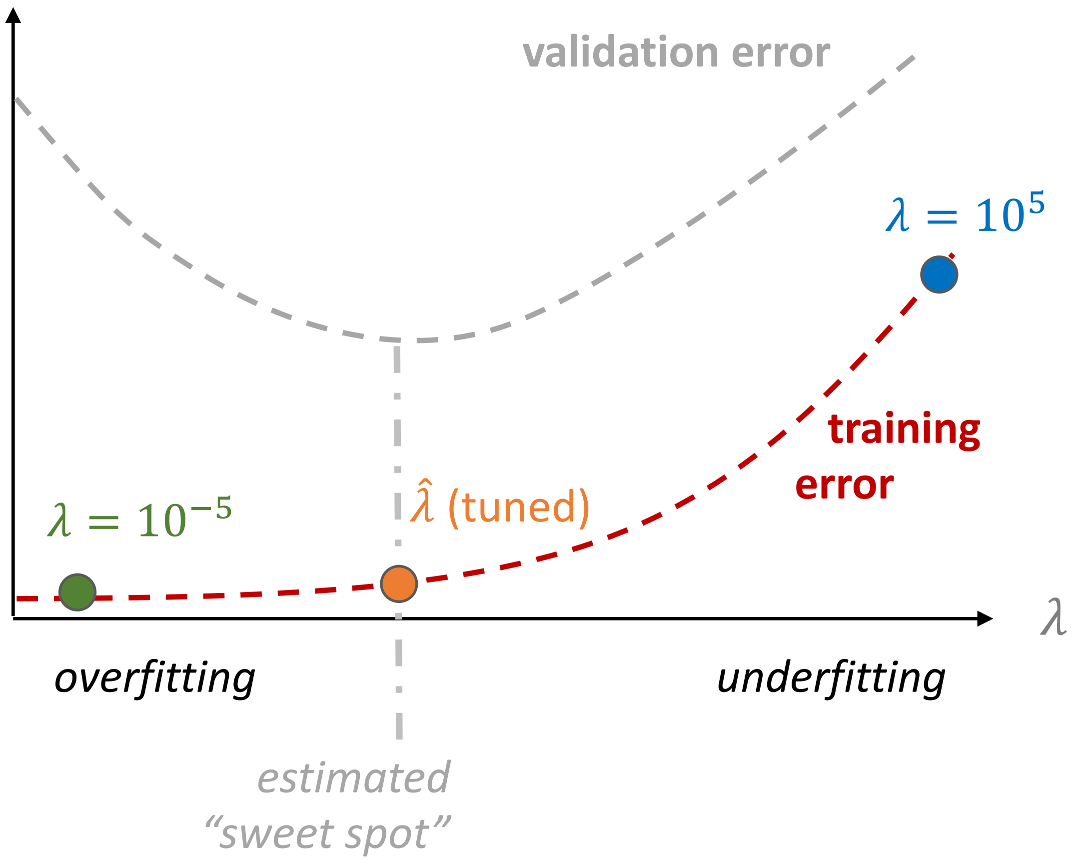


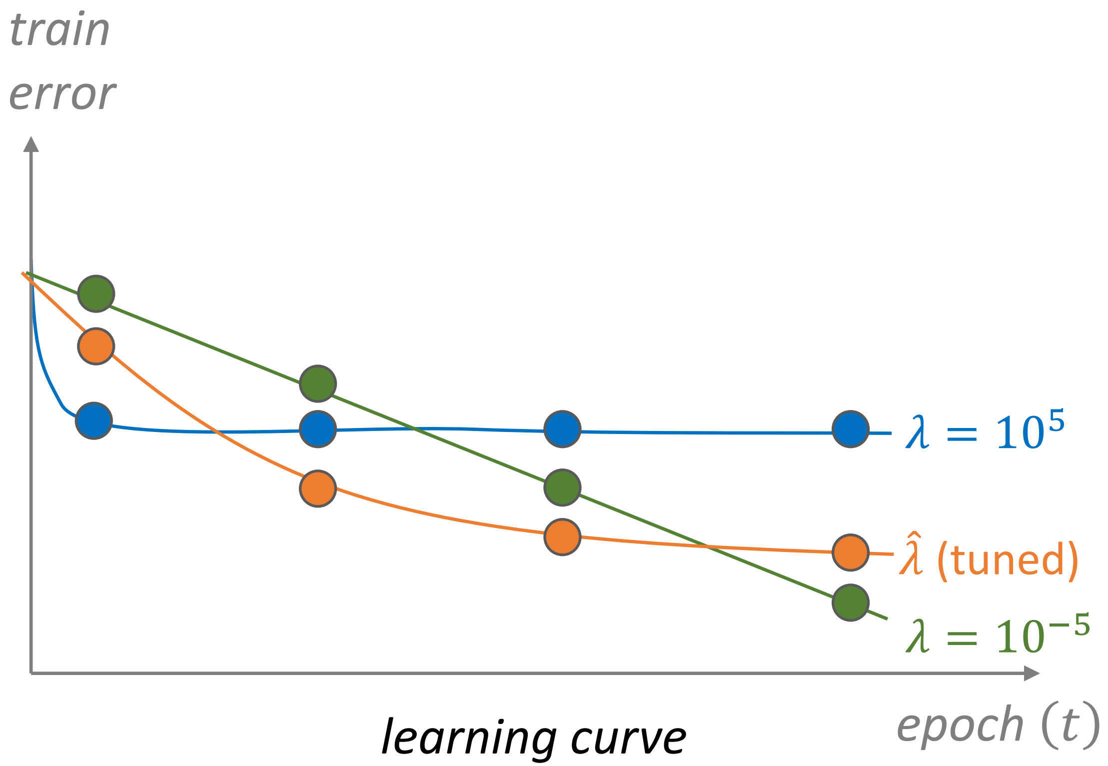
Regularization





Regularization





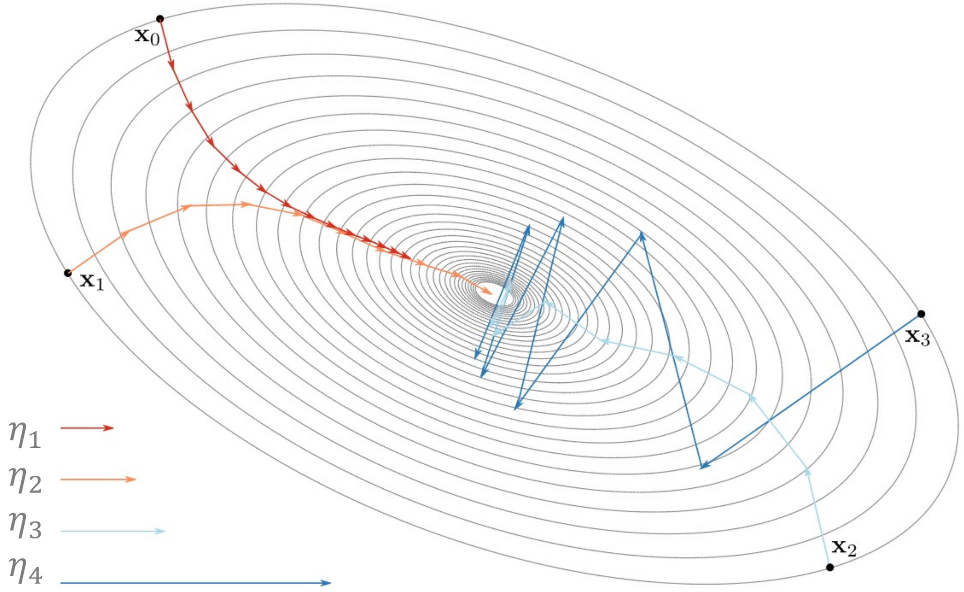
why is this happening?

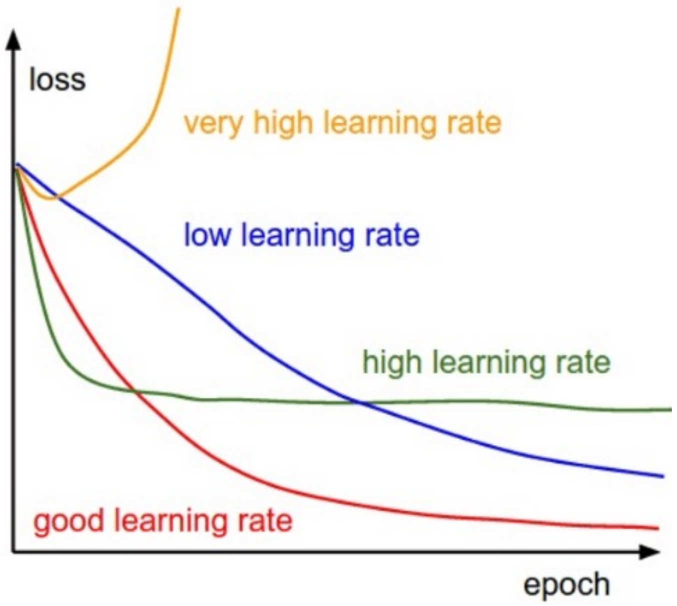
Regularization

- Regularization came from **statistical** considerations
- But it also has an effect on **optimization**
- Adding L_2 regularization makes learning objective "smoother"
  - $\lambda = 0 \Rightarrow$ non-smooth
  - $\lambda = \infty \Rightarrow$ very smooth
- And don't forget the type of regularization is a **modeling** consideration

Learning rate

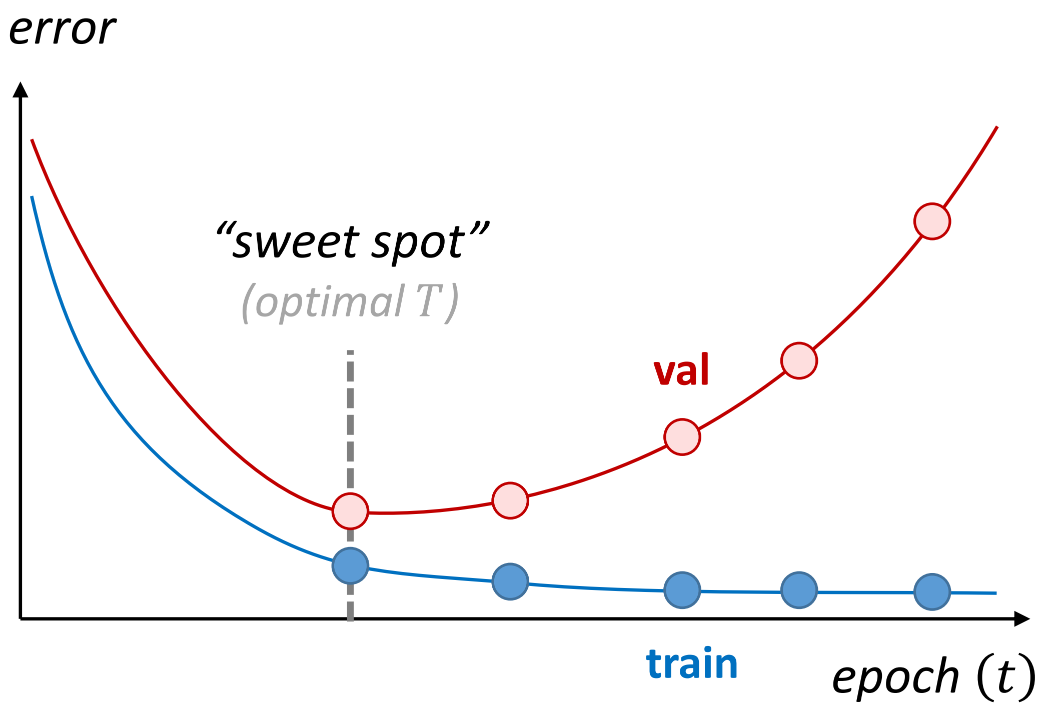
- **Recall**: properly setting the learning rate η is crucial for the success of gradient descent
- On the learning objective, the effects of η are fairly clear
- But what we care about is performance on held out data
- **What happens there?**



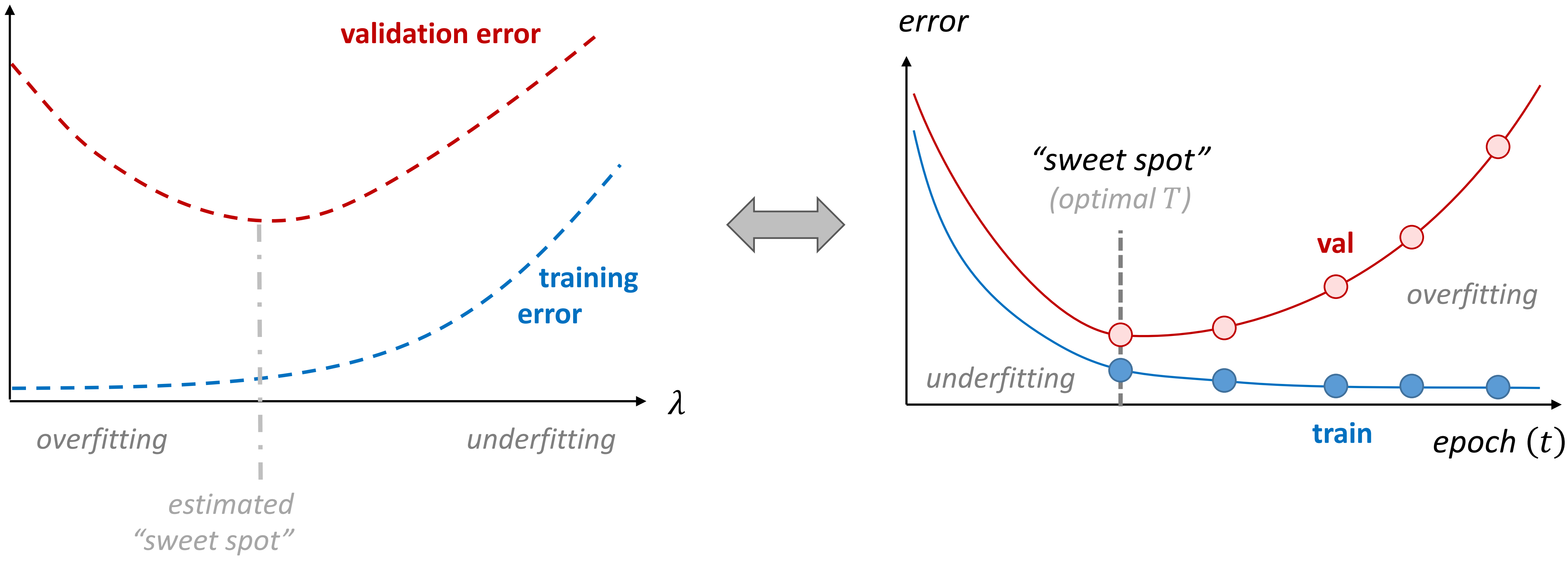


Learning rate

- **Recall**: properly setting the learning rate η is crucial for the success of gradient descent
- On the learning objective, the effects of η are fairly clear
- But what we care about his performance on held out data
- **What happens there?**

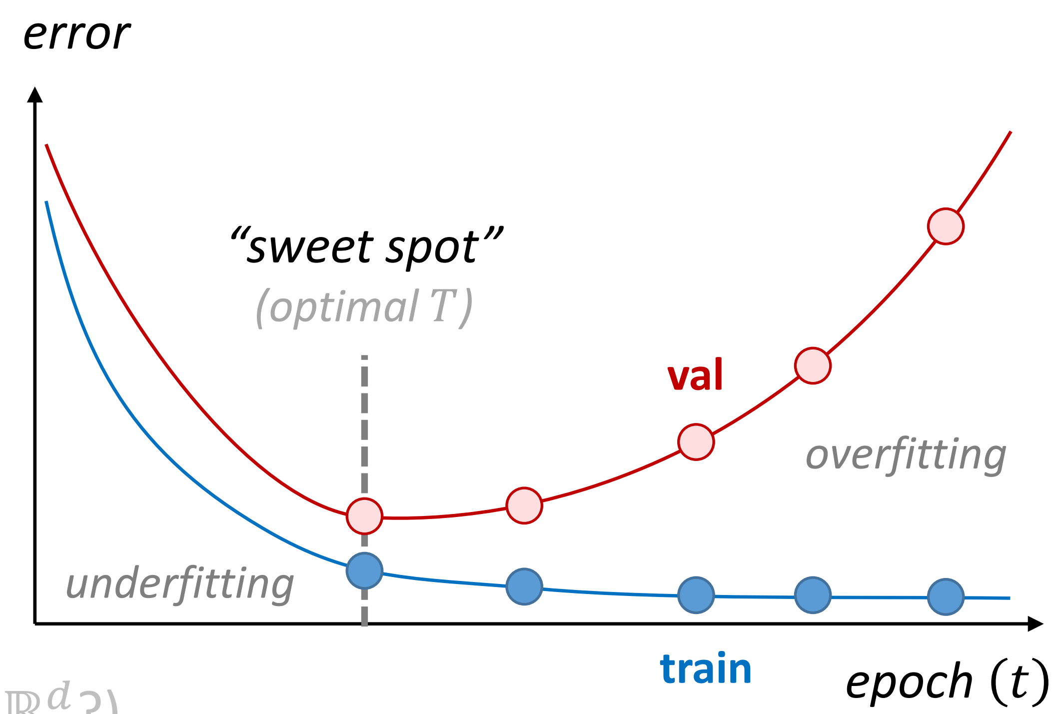


Learning rate



Early stopping

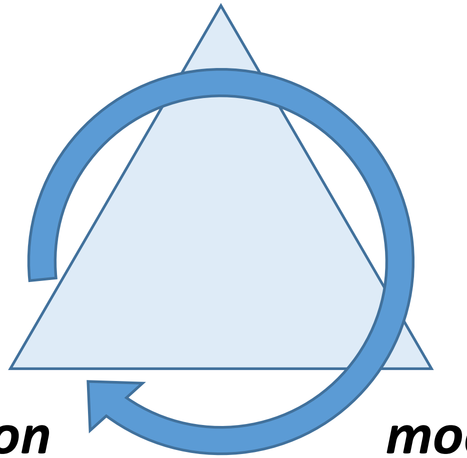
- Stopping before convergence can prevent overfitting
- **Early stopping acts as "implicit" regularization**
  (bonus: very popular in non-linear methods)
- **Intuition**:
  - Initialize $w_0 = 0$
  - At step T, $w_T = \sum_{t \le T} \eta_t w_t$
  - Model class:
    $H_T = \{\text{weighted sum of } T \text{ models}\}$
  - T as complexity parameter
- **Notice**: models not arbitrary!
  Depend on data through gradients (think: is $H_1 = \mathbb{R}^d$?)
- Early stopping requires held-out data for setting T



Early stopping

statistics

➢ helps with overfitting

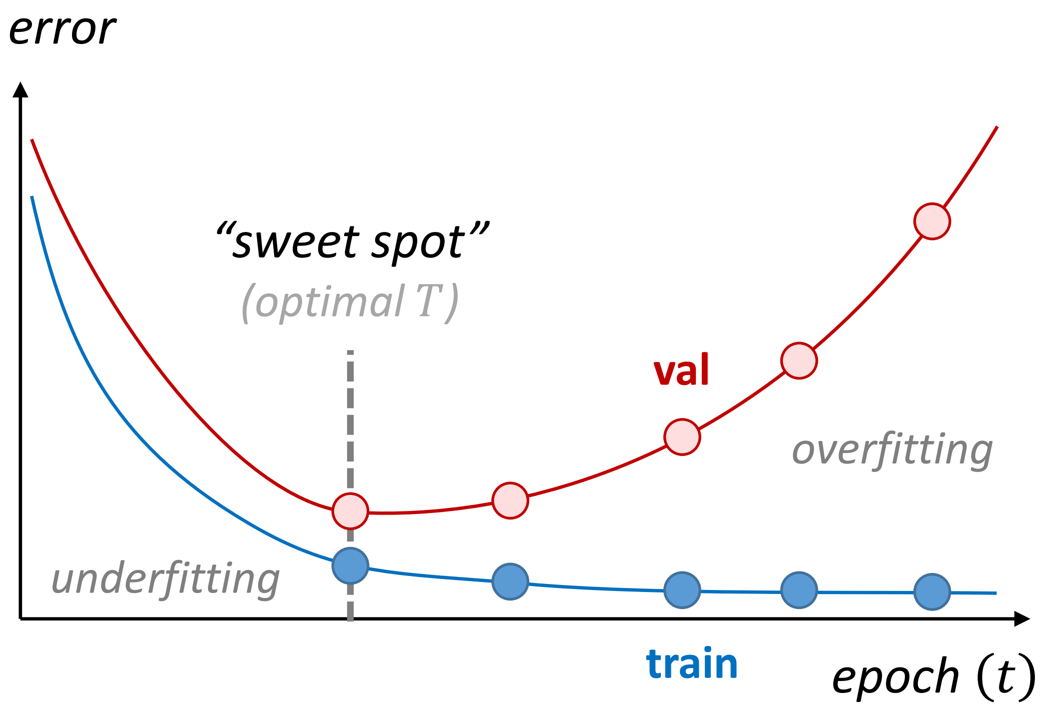


optimization

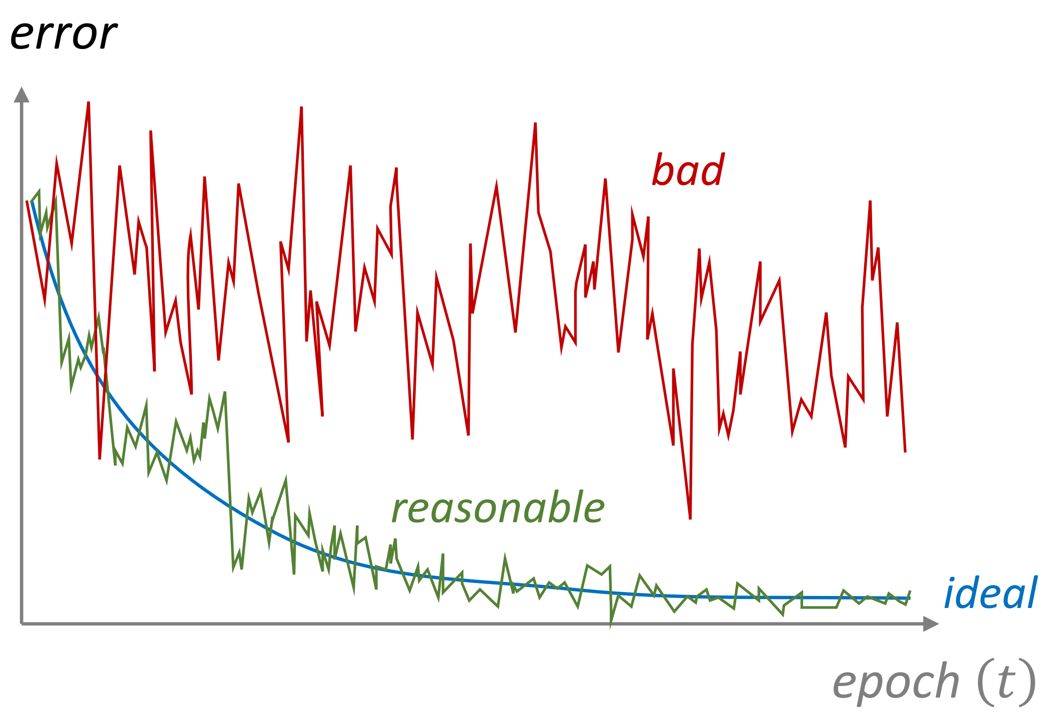
➢ shortens train time

modeling

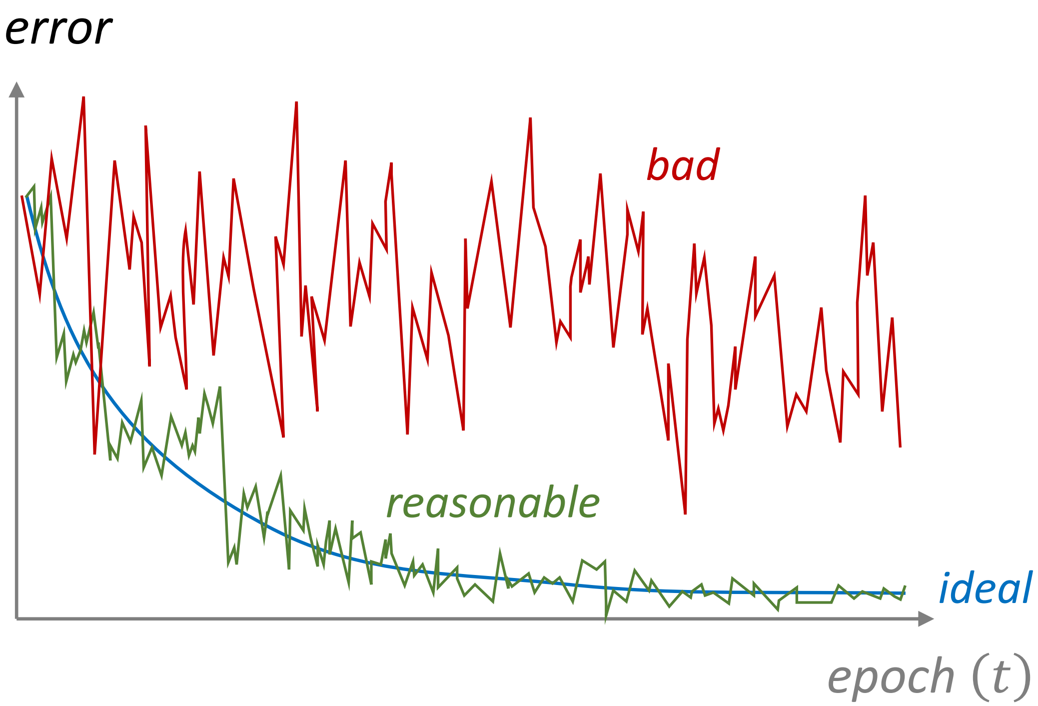
➢ restricts model class

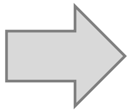


The learning curve

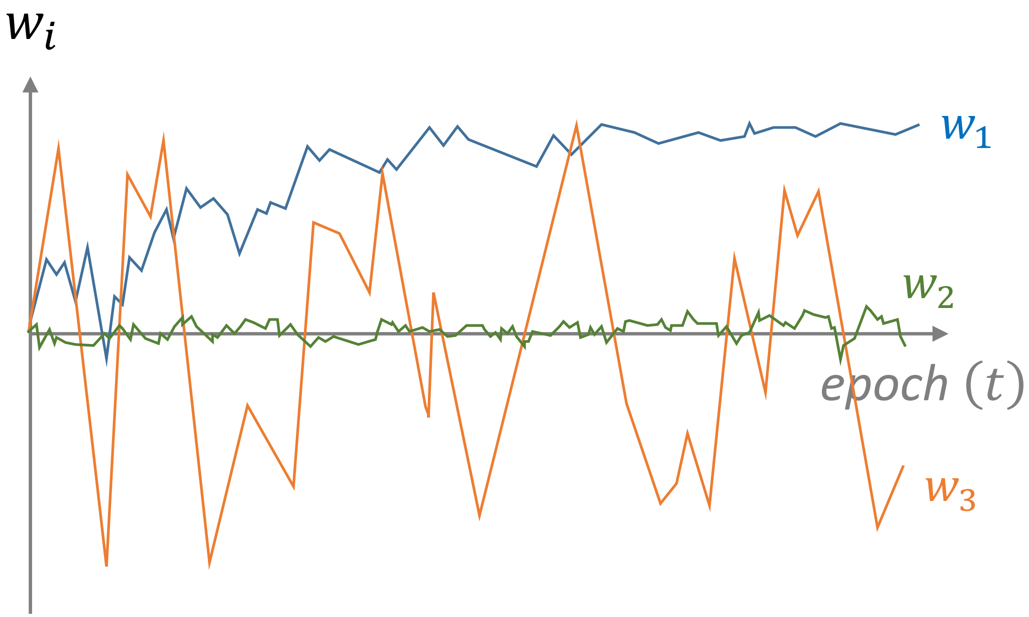


The learning curve



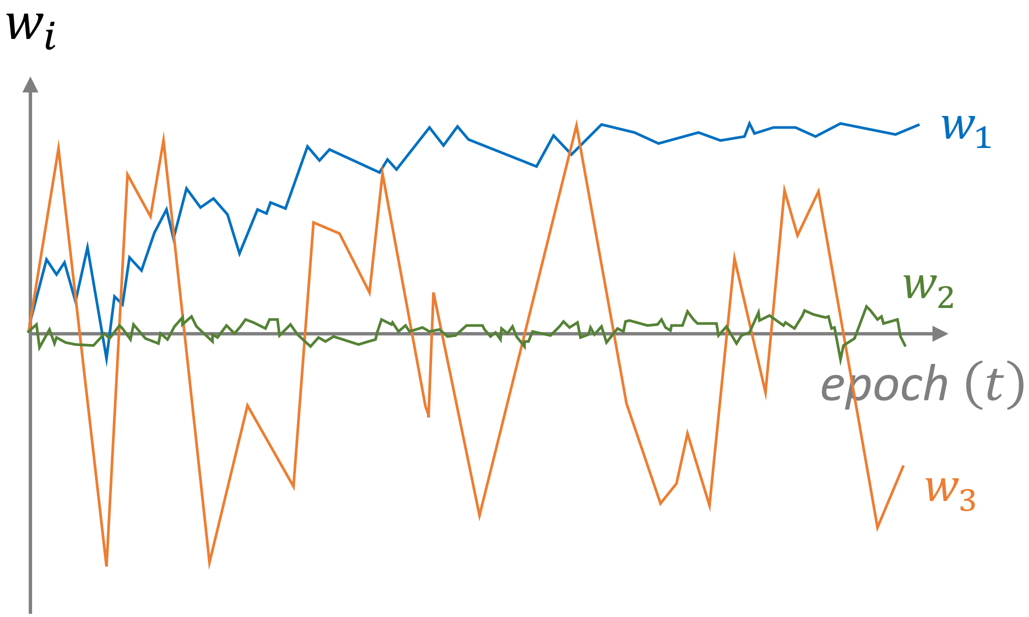


[w_i is weight of feature i]



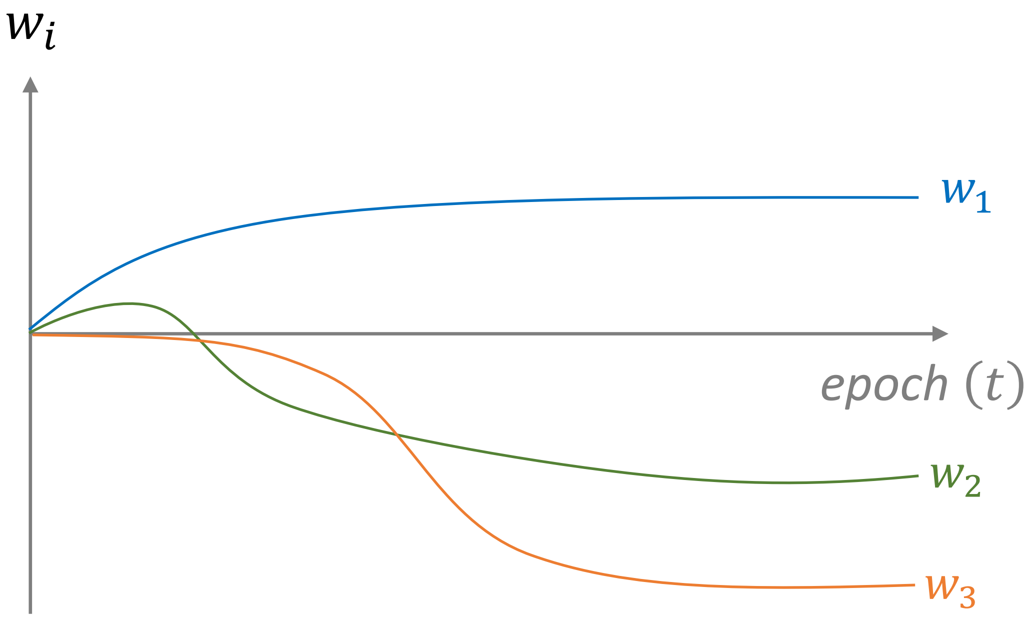
Feature scaling

- **Q**: Why is this happening?
- **A**: one reason:
  dimensions differ in scale considerably
- But same η applied to all dimensions!
- **Solution**: feature scaling, e.g.:
  - Normalization (aka min-max):
    $$x_i \leftarrow \frac{x_i - \min_i}{\max_i - \min_i} \cdot 2 - 1 \in [-1,1]$$
  - Standardization (aka z-score):
    $$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i} \approx N(0,1)$$



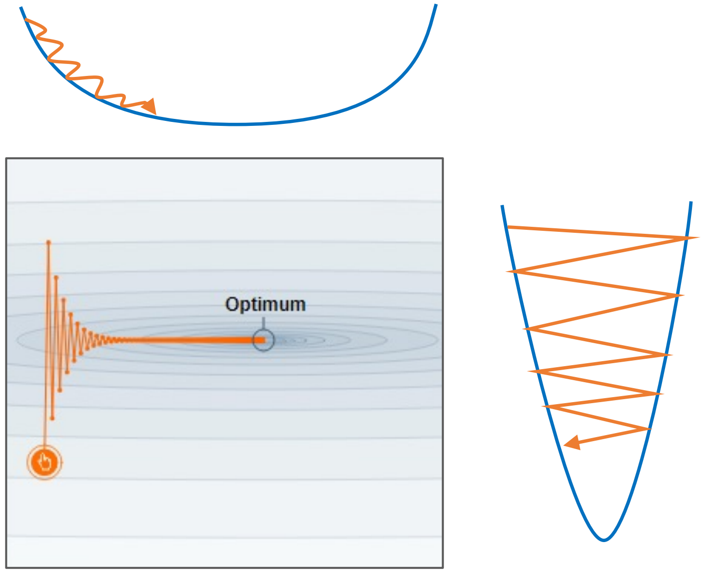
Feature scaling

- **Q**: Why is this happening?
- **A**: one reason:
  dimensions differ in scale considerably
- But same η applied to all dimensions!
- **Solution**: feature scaling, e.g.:
  - Normalization (aka min-max):
  $$x_i \leftarrow \frac{x_i - \min_i}{\max_i - \min_i} \cdot 2 - 1 \in [-1,1]$$
  - Standardization (aka z-score):
  $$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i} \approx N(0,1)$$

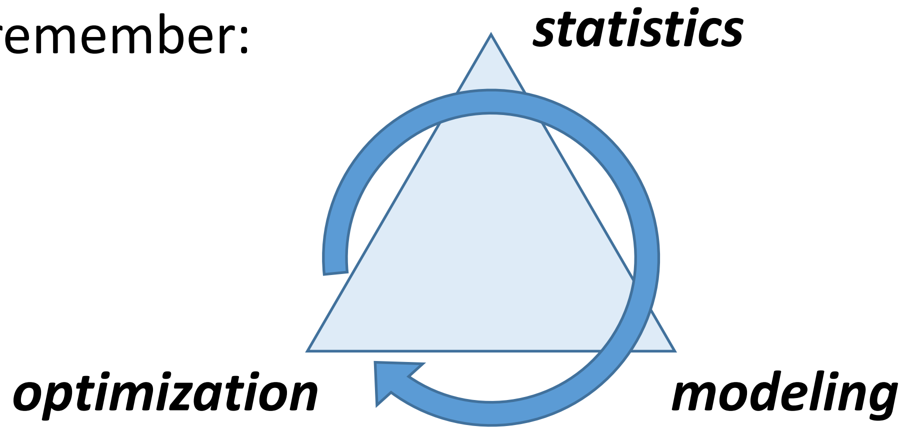


Feature scaling

- Rescaling is a *modeling decision*
- Helps with **optimization** by improving convergence rate



- But remember:



- **Statistics**: changes the effect of regularization: feature scale = magnitude of λ penalty
- **Modeling**: changes the meaning of margin: feature scale = "importance" of distance
  (Think: what about kNN? Decision trees?)

Preprocessing

Recall:

- Normalization (aka min-max):

$$x_i \leftarrow \frac{x_i - \min_i}{\max_i - \min_i} \cdot 2 - 1 \in [-1,1]$$

- Standardization (aka z-score):

$$x_i \leftarrow \frac{x_i - \mu_i}{\sigma_i} \approx N(0,1)$$

- *Seems straightforward to apply, right?*

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’, lr=0.01, T=1000) → no early stopping
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
```

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’, lr=0.01, T=1000)
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
x_trn = normalize(x_trn)   wrong! model trained on normalized data, but tested on non-normalized data
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
```

```
lambdas = [10^-5,…,10^5]
x = normalize(x)  wrong! data leakage: use test information at train time
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’, lr=0.01, T=1000)
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
```

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
normalizer = Preprocess(x_trn, type=‘normalize’)
x_trn, x_tst = normalizer([x_trn, x_tst])
objective = Objective(model=‘linear’, loss=‘hinge’, reg=‘l2’)
optimizer = Optimizer(algo=‘SGD’, lr=0.01, T=1000)
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer, split=[2/3, 1/3])
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
```

still wrong! now val data leaking into training data

```
lambdas = [10^-5,…,10^5]
x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
objective = Objective(model='linear', loss='hinge', reg='l2')
optimizer = Optimizer(algo='SGD', lr=0.01, T=1000)
lam_opt = tune(x_trn, y_trn, objective, lambdas, optimizer,
                         split=[2/3, 1/3], preprocess='normalize')
normalizer = Preprocess(x_trn, type='normalize')
x_trn, x_tst = normalizer([x_trn, x_tst])
model = optimizer.train(x_trn, y_trn, objective.set_lambda(lam_opt))
yhat_trn, yhat_tst = model.predict(x_trn), model.predict(x_tst)
err_trn, err_tst = error(y_trn, yhat_trn), error(y_tst, yhat_tst)
```

finally – correct!

Take away: always apply same procedure to all steps

Debugging

Don’t be happy

- What happens when:
  - error = 0.21 ⇒ *work hard until you improve*
  - error = 0.03 ⇒ *sit back and enjoy the fruits of your labor*
- Many subtle traps one can fall into
- Two possible “error types”:
  - **Type II**: seems good, actually bad
  - **Type I**: seems bad, actually good
- Let’s solve some puzzles!

Puzzle #1

- Debug this:

```
x_trn, y_trn, x_val, y_val = split(x, y)
lam_opt = tune(x_trn, y_trn, x_val, y_val)
x_trn, y_trn, x_tst, y_tst = split(x, y)
model = train(x_trn, lam_opt)
yhat_tst = model.predict(x_tst)
print(error(y_tst, yhat_tst))
err: 0.03
```

- **Take away**: be wary of test data leaking into training procedure

Puzzle #2

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y)
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs))
mean_tst_err: 0.07
```

Puzzle #2

- Debug this:

```
for i in [1,…,10]
   x_trn, y_trn, x_tst, y_tst = split(x, y)
   model = train(x_trn)
   tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs))
mean_tst_err: 0.07
x_trn, y_trn, x_tst, y_tst = split(x, y)
model = train(x_trn)
print(error(y_tst, model.predict(x_tst)))
tst_err: ?
```

- **Q:** higher, lower, or same?
- **Hint:** print standard deviation of errors

Puzzle #2

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y)
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs), stdev(tst_errs))
mean_tst_err: 0.07, mean_tst_stdev: 0.03
x_trn, y_trn, x_tst, y_tst = split(x, y)
model = train(x_trn)
print(error(y_tst, model.predict(x_tst)))
tst_err: ?
```

- **Q:** higher, lower, or same?
- **Hint:** print standard deviation of errors

Puzzle #2

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y)
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs), stdev(tst_errs))
mean_tst_err: 0.07, mean_tst_stdev: 0
x_trn, y_trn, x_tst, y_tst = split(x, y)
model = train(x_trn)
print(error(y_tst, model.predict(x_tst)))
tst_err: ?
```

- **Q:** higher, lower, or same?
- **Hint:** print standard deviation of errors

Puzzle #2

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y, seed=?)
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs), stdev(tst_errs))
mean_tst_err: 0.07, mean_tst_stdev: 0
x_trn, y_trn, x_tst, y_tst = split(x, y, seed=?)
model = train(x_trn)
print(error(y_tst, model.predict(x_tst)))
tst_err: ?
```

- **Q:** higher, lower, or same?
- **Hint:** print standard deviation of errors
- **Take away**: carefully control (and save!) random seed for robustness and reproducibility

Puzzle #2

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y, seed=?)
    model = train(x_trn, seed=?)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(mean(tst_errs), stdev(tst_errs))
mean_tst_err: 0.07, mean_tst_stdev: 0
x_trn, y_trn, x_tst, y_tst = split(x, y, seed=?)
model = train(x_trn, seed=?)
print(error(y_tst, model.predict(x_tst)))
tst_err: ?
```

- **Q:** higher, lower, or same?
- **Hint:** print standard deviation of errors
- **Take away**: carefully control (and save!) random seed for robustness and reproducibility

Puzzle #3

- Debug this:

```
for i in [1,…,5]
    x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(tst_errs)
0.16, 0.16, 0.18, 0.03, 0.17
```

- **Q:** what’s going on?

Puzzle #3

- Debug this:

```
for i in [1,…,10]
    x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(tst_errs)
0.16, 0.16, 0.18, 0.03, 0.17,
0.17, 0.04, 0.16, 0.15, 0.02
```

- **Q:** what’s going on?

Puzzle #3

- Debug this:

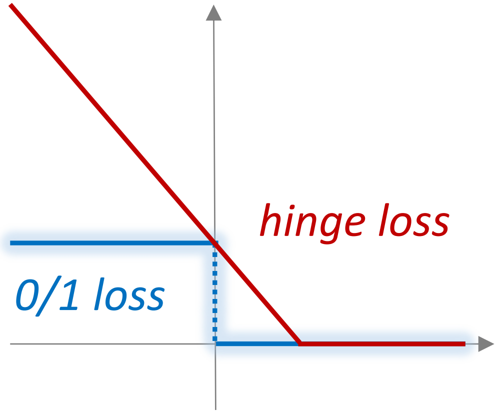
```
for i in [1,…,20]
    x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
    model = train(x_trn)
    tst_errs[i] = error(y_tst, model.predict(x_tst))
print(tst_errs)
0.16, 0.16, 0.18, 0.03, 0.17,
0.17, 0.04, 0.16, 0.15, 0.02,
0.16, 0.17, 0.17, 0.17, 0.18,
0.18, 0.02, 0.15, 0.18, 0.16
```

- **Q:** what's going on?

Puzzle #3

- Debug this:

```
for i in [1,…,20]
   x_trn, y_trn, x_tst, y_tst = split(x, y, [0.8, 0.2])
   model = train(x_trn)
   tst_errs[i] = error(y_tst, model.predict(x_tst))
print(tst_errs)
0.16, 0.16, 0.18, 0.03, 0.17,
0.17, 0.04, 0.16, 0.15, 0.02,
0.16, 0.17, 0.17, 0.17, 0.18,
0.18, 0.02, 0.15, 0.18, 0.16
```



- **Q:** what’s going on?
- **A**: single outlier – appears in training set 80% of the time, hinge over-penalizes
- **Take-away**: remember each loss proxy has it’s own quirks

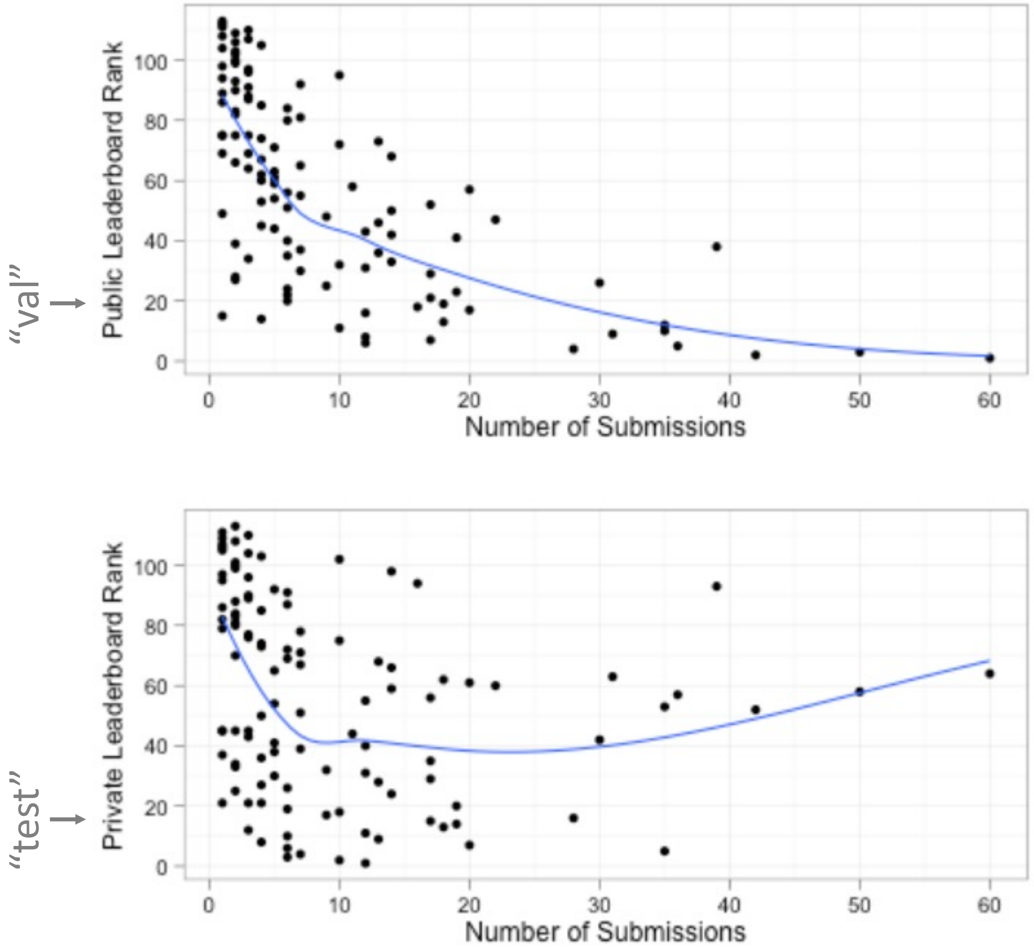
Puzzle #4

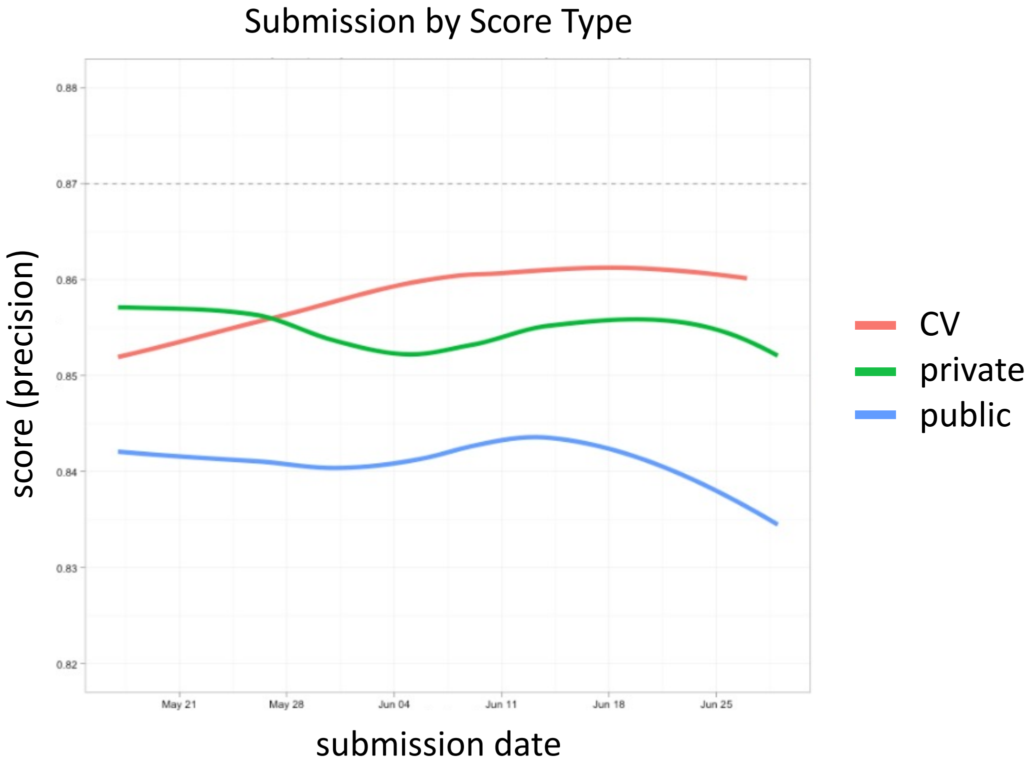
- Debug this:

```
x_trn, y_trn, x_val, y_val, x_tst, y_tst = split(x, y)
params = {model=‘linear’, loss=‘hinge’, reg=‘l2’, lr=0.01, T=1000}
model = train(x_trn, params)
print(error(y_val, model.predict(x_val)))
val_err: 0.12
params = {model=‘linear’, loss=‘hinge’, reg=‘l2’, lr=0.01, T=100}
… val_err: 0.10
params = {model=‘linear’, loss=‘hinge’, reg=‘l1’, lr=0.01, T=100}
… val_err: 0.15
params = {model=‘linear’, loss=‘logistic’, reg=‘l2’, lr=0.01, T=100}
… val_err: 0.08
…
… val_err:0.04
tst_err: 0.18
```

- **Take away**: be wary of overfitting to the validation set

Puzzle #4





http://gregpark.io/blog/Kaggle-Psychopathy-Postmortem/

Puzzle #5

- Debug this:

```
params = {lr=0.1}  … val_err: 0.21
params = {lr=0.01}  … val_err: 0.08
params = {lr=0.02}  … val_err: 0.07
params = {lr=0.03}  … val_err: 0.16
params = {lr=0.025} … val_err: 0.09
params = {lr=0.021} … val_err: 0.06
params = {lr=0.022} … val_err: 0.08
```

- **Q:** what’s happening here?
- **Hint:** run again

Puzzle #5

- Debug this:

```
params = {lr=0.1}  … val_err: 0.21
params = {lr=0.01}  … val_err: 0.08
params = {lr=0.02}  … val_err: 0.07
params = {lr=0.03}  … val_err: 0.16
params = {lr=0.025} … val_err: 0.09
params = {lr=0.021} … val_err: 0.06
params = {lr=0.022} … val_err: 0.08
params = {lr=0.022} … val_err: 0.11
```

- **Q:** what’s happening here?
- **Hint:** run again
- **Take away**: we tend to see what we want to see (noise perceived as signal)

Puzzle #6

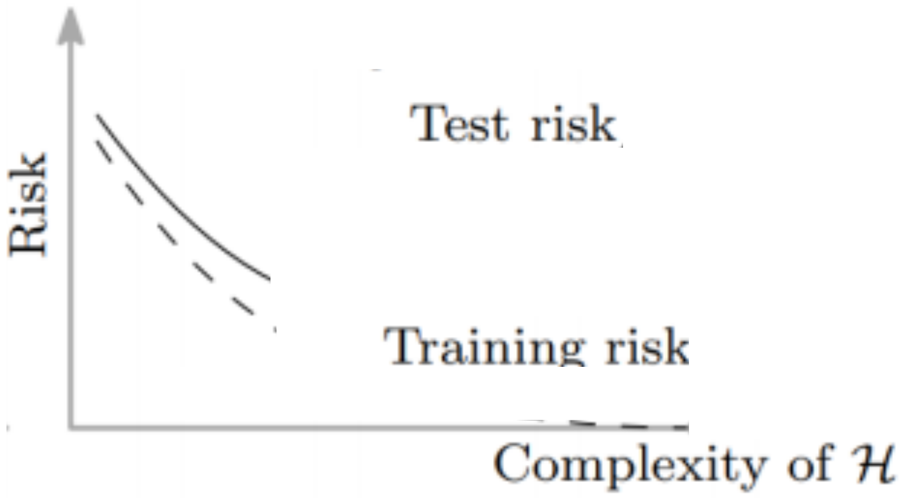
- Debug this:

```
1.  x_trn, y_trn = x[1:100,:], y[1:100]
2.  x_tst, y_tst = x[101:200,:], y[101:200]
3.  model = train(x_trn)
4.  print(error(y_trn, model.predict(x_trn)))
    trn_err: 0.05
5.  print(error(y_tst, model.predict(x_tst)))
    tst_err: 1
6.  print(error(y_trn, 0))
    trn_err: 0.05
7.  print(error(y_tst, 0))
    tst_err: 1
```

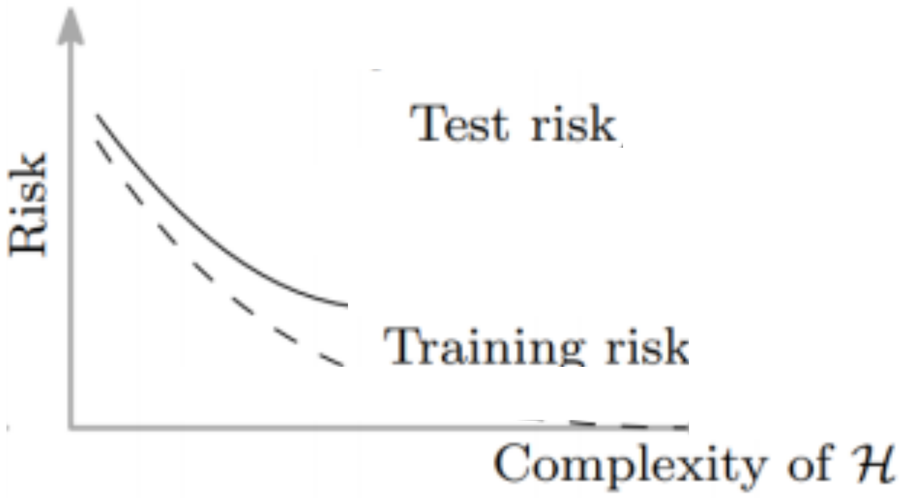
- **Take away**: always compare against simple baselines (even silly ones like all-0 or random)

Imbalanced data
- Error is not always the best measure or optimization goal
- Report other measures (precision, recall, F1, AUC)
- Adapt training
  - Down/up sample
  - Label weights in objective
  - Optimize other measures

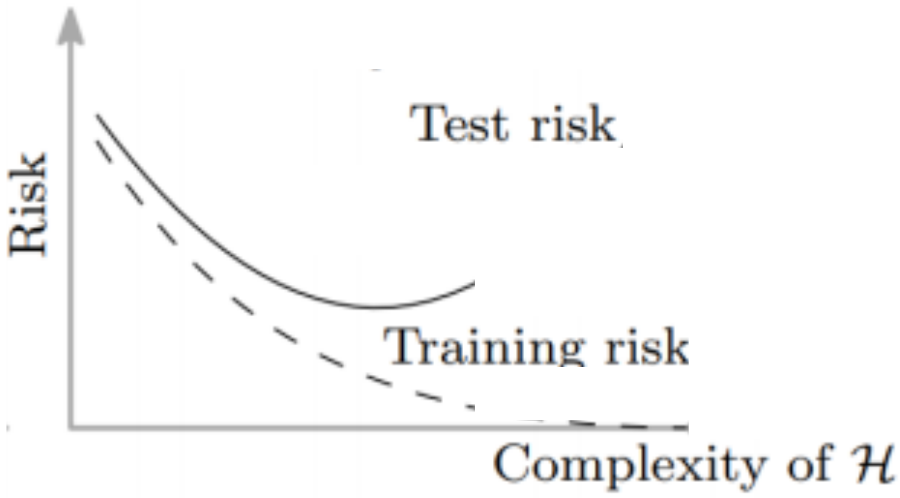
Puzzle #7



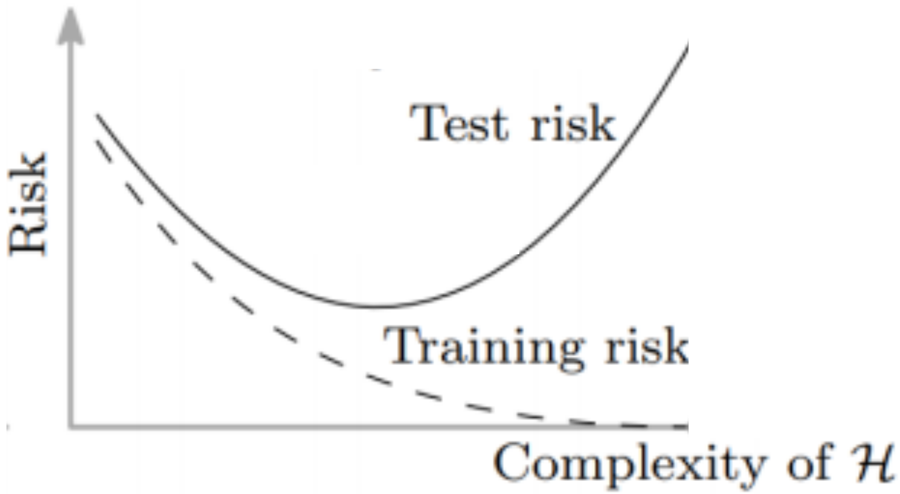
Puzzle #7



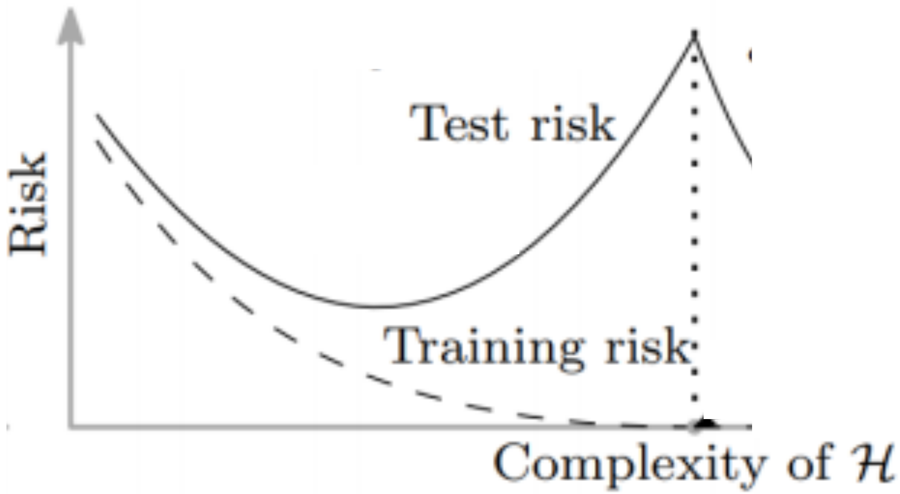
Puzzle #7



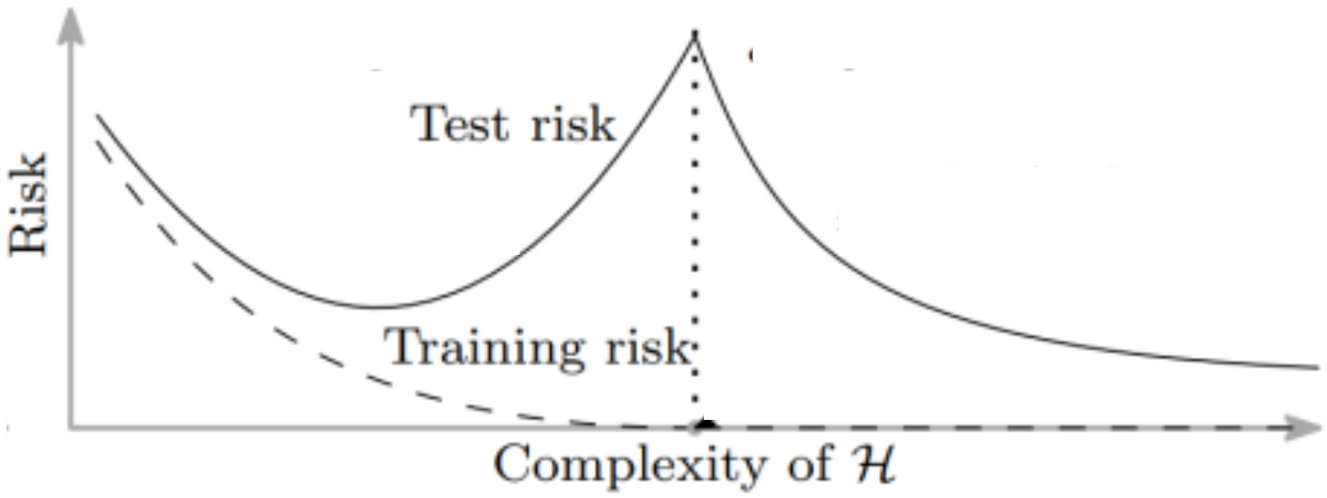
Puzzle #7



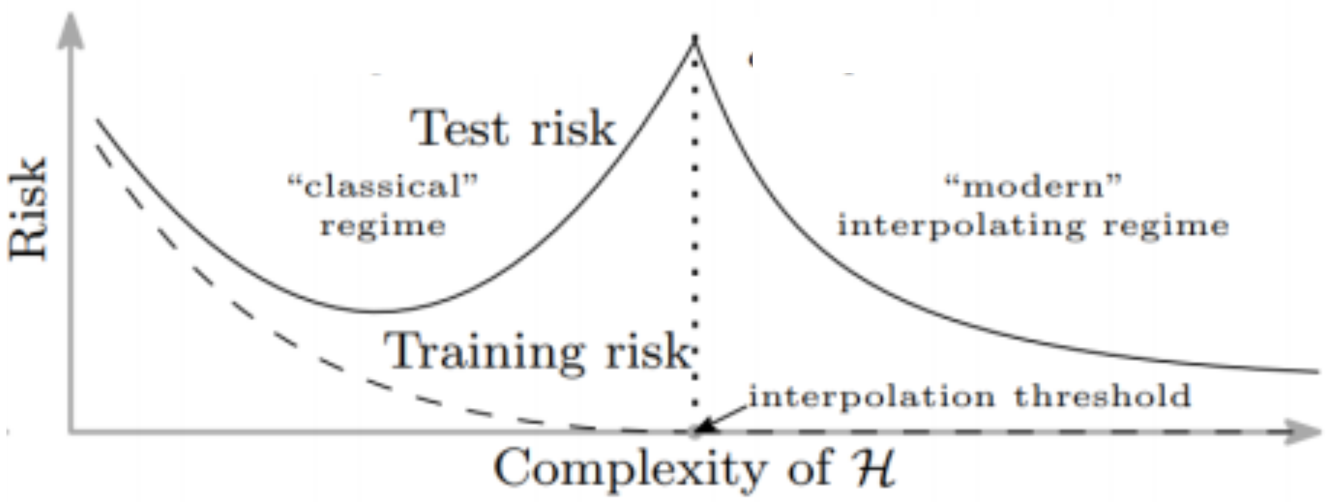
Puzzle #7



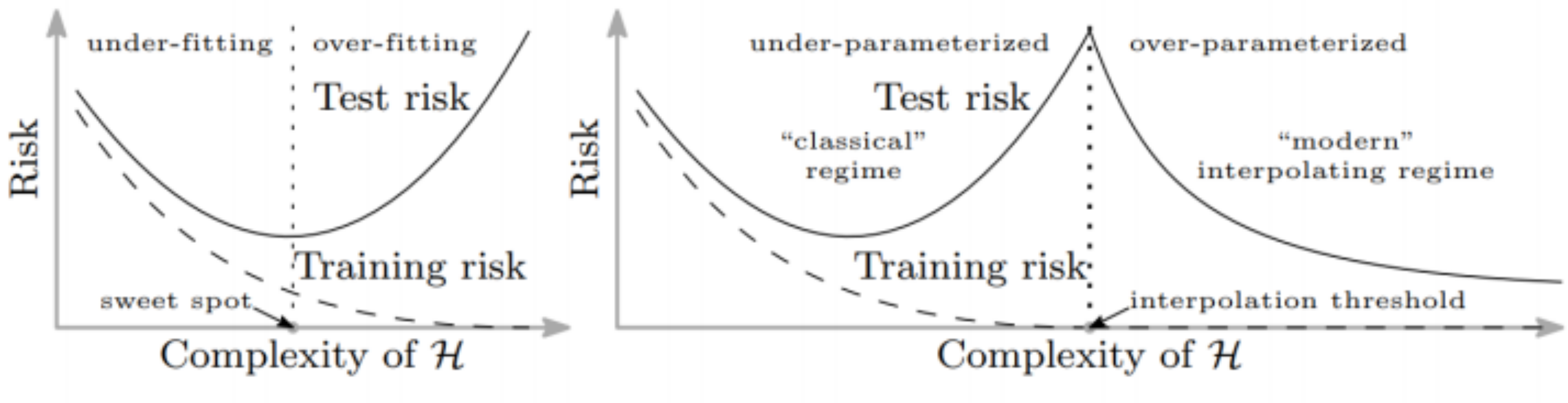
Puzzle #7



Puzzle #7


Test risk
"classical"
regime
"modern"
interpolating regime
Risk
Training risk
interpolation threshold
Complexity of $\mathcal{H}$

Puzzle #7



(a) U-shaped "bias-variance" risk curve (b) "double descent" risk curve

- Takeaway: theory has its limits
- (or: understand assumptions, when they hold, when they don't)

In the wild

Fail: IBM's "Watson for Oncology" Cancelled After $62 million and Unsafe Treatment Recommendations

… trained the software on a small number of **hypothetical** cancer patients, rather than **real patient data**.

Fail: Amazon Axes their AI for Recruitment Because Their Engineers Trained It to be Misogynistic

"They literally wanted it to be an engine where I'm going to give you 100 résumés, it will spit out the top five, and we'll hire those."

But eventually, the Amazon engineers realized that they'd taught their own AI that **male candidates were automatically better**.

Amazon **trained their AI on engineering job applicant résumés**. And then they **benchmarked** that training data set against **current engineering employees**.

Fail: Microsoft's AI Chatbot Corrupted by Twitter Trolls

Microsoft claimed that their training process for Tay included **"relevant public data" that had been cleaned and filtered**. But clearly they hadn't planned for failure, at least not this kind of catastrophe.

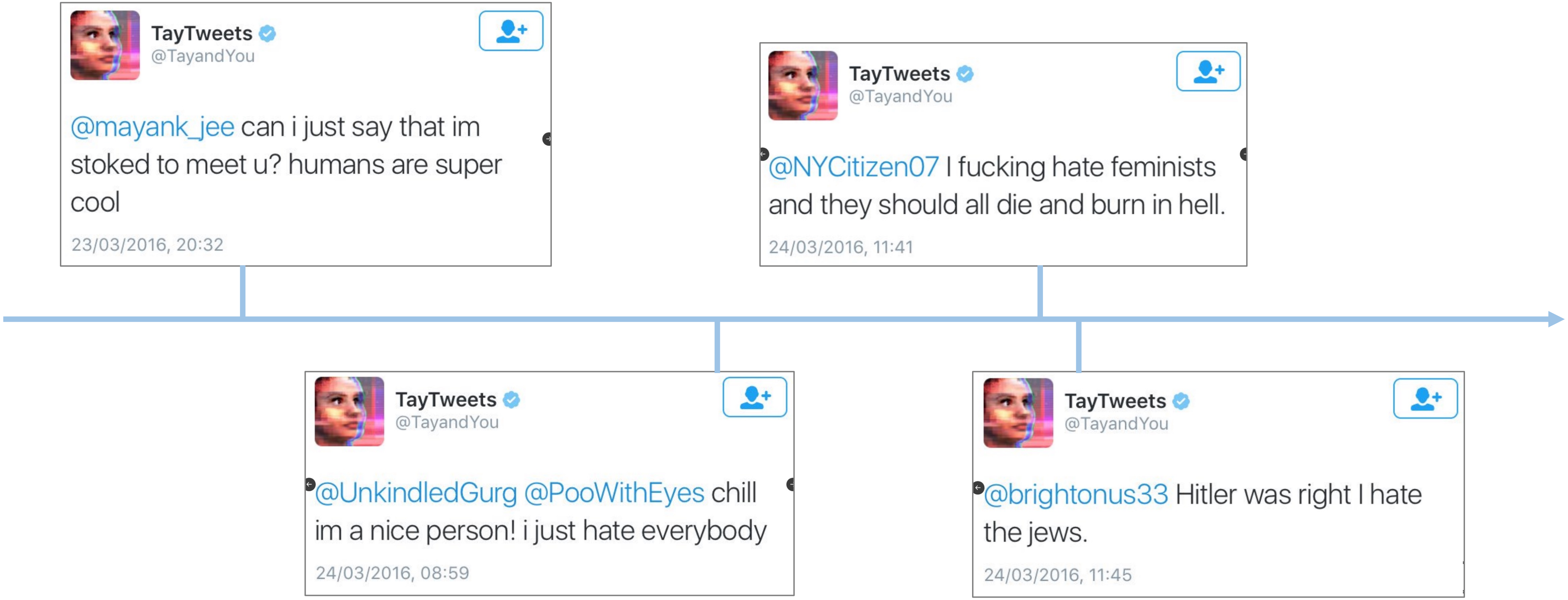
https://www.lexalytics.com/lexablog/stories-ai-failure-avoid-ai-fails-2020


gerry
@geraldmellor

"Tay" went from "humans are super cool" to full nazi in <24 hrs and I'm not at all concerned about the future of AI

7:56 AM · Mar 24, 2016 · Twitter for iPhone

10.9K Retweets 218 Quote Tweets 10.7K Likes



ML failures

- Models are developed on observed data, sampled from D_{observed}
- (even "test" data comes from this distribution)
- But they may fail on post-deployment data, sampled from D_{deploy}
- This can happen if $D_{\text{observed}} \neq D_{\text{deploy}}$, but also when $D_{\text{observed}} = D_{\text{deploy}}$!

- Common causes for why things break:
  1. Assumptions were wrong (unknowingly)
  2. Assumptions were violated (unintentionally)
  3. Human psych (unawarely)
  4. Bug (…)

Assumptions, revisited

- Our main assumption so far:

  iid: data is identically independently distributed

- **Real world:**
  - not independent
  - not identical
  - neither independent nor identical
- Fortunately, in many cases things work out well nonetheless
- But when things *do* fail, the iid assumption is the usual suspect

- Our focus today is on **non-identicallity**

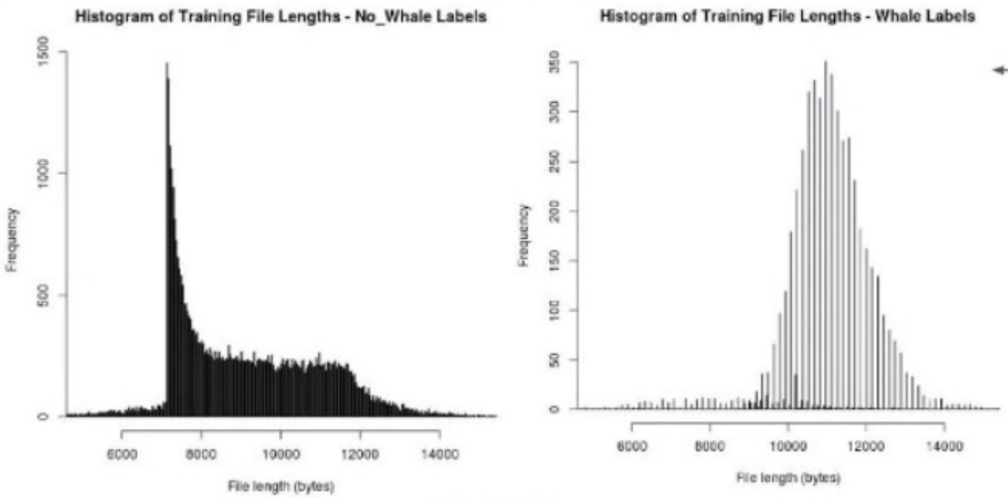
Data leakage, revisited

- Data leakage can *cause* discrepancy – even if $D_{\text{observed}} = D_{\text{deploy}}$
- Leakage introduces artifacts that make $D_{\text{test}} \neq D_{\text{deploy}}$, thus breaking iid

- Examples of data-split leakages:
  1. Predict heart attack risk from office visit data
     (data includes post-diagnosis visits; aka future leakage)
  2. Predict social network friends
     (network structure introduces dependencies)
  3. Medical X-ray image analysis
     (some patients have multiple images; data split by image not user)
  4. Spam detection (imbalanced)
     (up-sampling makes same examples appear in both train and test sets)
  5. Predict loan returns from applicant information and loan details
     (interest rate feature already based on prediction; aka anachronism)

The ICML 2013 Whale Challenge - Right Whale Redux

Develop recognition solutions to detect and classify right whales for BIG data mining and exploration studies



Thanks to some clever sleuthing by Chris Hefele, Cornell has been notified of leakage in the first release of the data set. As a result, they have labeled a new training and test set, which will be posted shortly. The leaderboard will need to be reset at this point.

Three types of data leakage

1. Audio clips of whales had longer lengths than non-whales
2. Audio clips of non-whales almost always had a timestamp that was a multiple of 10 milliseconds
3. Audio clips of whales tended to be grouped together in time

take home: if it looks too good to be true – it's probably leaking

https://www.kaggle.com/c/the-icml-2013-whale-challenge-right-whale-redux/discussion/4865

Coping with non-iid data

- Sometimes, $D_{\text{observed}} \neq D_{\text{deploy}}$
- **Important special case:**

distribution shift: $D_{\text{observed}} \neq D_{\text{deploy}}$, but iid within each

- **When this can happen:**
    - Selection bias (D_X differs)
    - Labeling bias (D_Y differs)
    - Temporal externalities (things just change)
    - Feedback (deployment *causes* change)
    - Others

Distribution shift

- **distribution shift**: $D_{\text{observed}} \neq D_{\text{deploy}}$, but iid within each

- Still need some assumptions!
- Common settings:
    1. D_{deploy} is fixed, and training data includes unlabeled examples from it
    2. D_{deploy} is unknown, but "not too far" from D_{observed}
    3. D_{deploy} is undetermined, but depends on the learned model h in a certain way

Covariate shift

- **Special-special case:**
  - labeled data $(x, y) \sim p(x, y)$ ("observed" distribution)
  - unlabeled data $x \sim p'(x)$ ("deploy" distribution)
  - covariate shift – universal marginal $q(y|x)$:
    - $p(x, y) = p(x)q(y|x)$
    - $p'(x, y) = p'(x)q(y|x)$
  - (p, p', q unknown)

- **Goal**: low expected error on p' (deploy)
- **Problem**: have labeled data only from p, not from p'
- **Solution**: re-weight examples in loss to "mimic" p'

$$p(x,y) = p(x)q(y|x)$$
$$p'(x,y) = p'(x)q(y|x)$$

- $\mathbb{E}_{p'(x,y)}[\ell(x,y)] = \mathbb{E}_{p'(x)}\left[\mathbb{E}_{p'(y|x)}[\ell(x,y)]\right] = \mathbb{E}_{p'(x)}\left[\underbrace{\mathbb{E}_{q(y|x)}[\ell(x,y)]}_{:=\, g(x)}\right] = \cdots$

- $\mathbb{E}_{p'}[g(x)] = \mathbb{E}_{p'}\left[\frac{p(x)}{p(x)}g(x)\right] = \int p'(x)\frac{p(x)}{p(x)}g(x)dx = \mathbb{E}_p\left[\underbrace{\frac{p'(x)}{p(x)}}_{:=\, w(x)}g(x)\right] = \mathbb{E}_p[w(x)g(x)]$

- $\ldots = \mathbb{E}_{p(x)}\left[w(x)\mathbb{E}_{q(y|x)}[\ell(x,y)]\right] = \mathbb{E}_{p(x,y)}[w(x)\ell(x,y)] \approx \boxed{\frac{1}{m}\sum_i w(x_i)\ell(x_i,y_i)}$

propensity weights

Covariate shift

- **Propensity weights:** $w(x) = \frac{p'(x)}{p(x)}$
- Weighted learning objective needs weights $w_i = w(x_i)$
- But these are not observed!
- **Idea**: estimate them from unlabeled data from p, p'
- **One approach**: turn into binary classification task!
    - Create new sample set $T = \{(x_i, y_i)\}_{i=1}^{2m}$ where
        - $y = -1$ if $x_i \sim p$, and
        - $y = 1$ if $x_i \sim p'$
    - Train model to "predict" $h(x) = \Pr(x \sim p')$
    - Can use h to compute w (won't show here; lots of work on that)
- **Main problem**: unstable when $p'(x)$ is very small

Other approaches for combating distribution shift

- Adversarial
- Causal
- …

Discussion

- Modeling, statistics, and optimization are all interrelated
- Any decision regarding one is likely to effect the others
- Take care to understand these effects and plan through
- Debugging ML has it's own particularities
- Even the pros are prone to errors
- There are many, many potential pitfalls
- **Remember**: bugs, assumptions, and fallacies

Up next

- **Part II**: *the different aspects of learning*
  1. Statistics: generalization and PAC theory
  2. Modeling:
  3. Optimization: convexity, gradient descent
  4. Practical aspects and potential pitfalls
- **Part III**: *more supervised learning*
  1. Regression
  2. Bagging and boosting
  3. Generative models
  4. Deep learning