Supervised Learning Review

Data and task

The **data**:

Set of pairs (x, y)

The **task**:

Having x predict y

- Classification: y is a discrete label
- Regression: y is a numerical value

The model

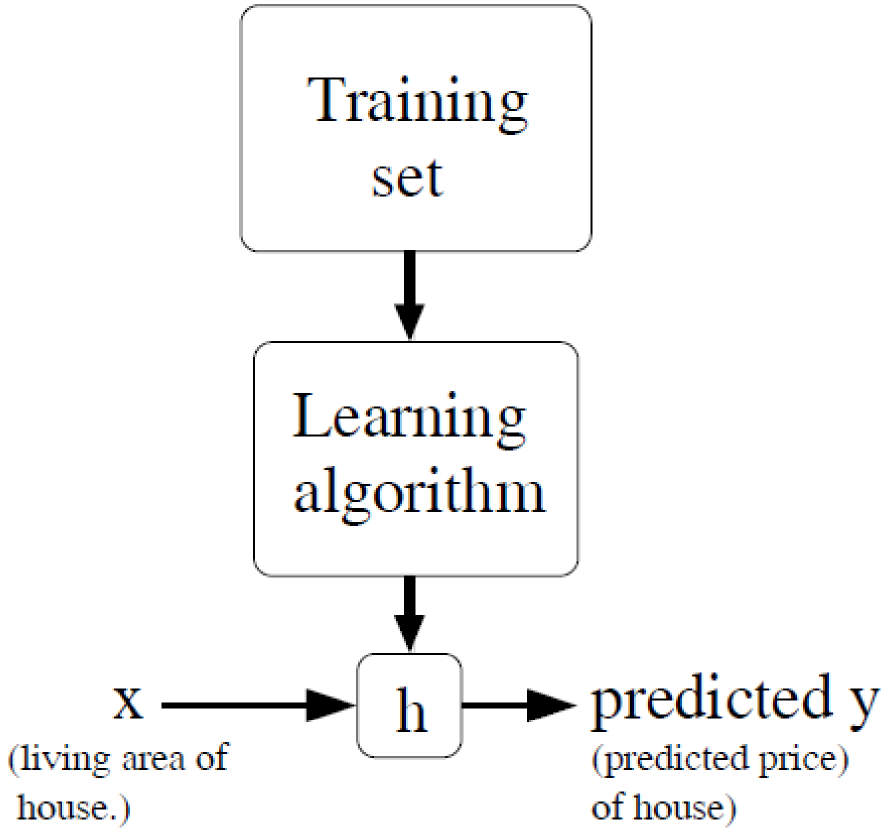
A parametric family of functions f that approximate

$$y \approx f(x, \theta)$$

θ: **parameters** (moving parts set to fit the data)

To get different fs we set **hyperparameters** (regularization constants, design choices)

Learning


Training set
Learning algorithm
x
(living area of house.)
h
predicted y
(predicted price) of house)

MODELS THAT WE KNOW

Nearest Neighbors

Predict y for a new x based on k nearest neighbors in the dataset.

Parameters:

- None, has to store the whole training set

Hyperparameters:

- k

More info: Homework 1

Linear Regression

$$y \approx f(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots$$

Parameters:

- $\theta_0, \theta_1, \theta_2, \ldots$
- We set them to:

$$\theta = \arg\min_{\Theta} \frac{1}{2} \sum_i \left(y^{(i)} - f\left(x^{(i)}\right)\right)^2 + \frac{\lambda}{2} \sum_j \theta_j^2$$

Hyperparameters:

- λ (larger λ yields small Θ)

More info: Homework 3, Lecture 6

Parameteric vs Nonparametric

Parametric models (e.g. linear regression)**:**

[in statistics]: a family of probability distributions that has a finite number of parameters

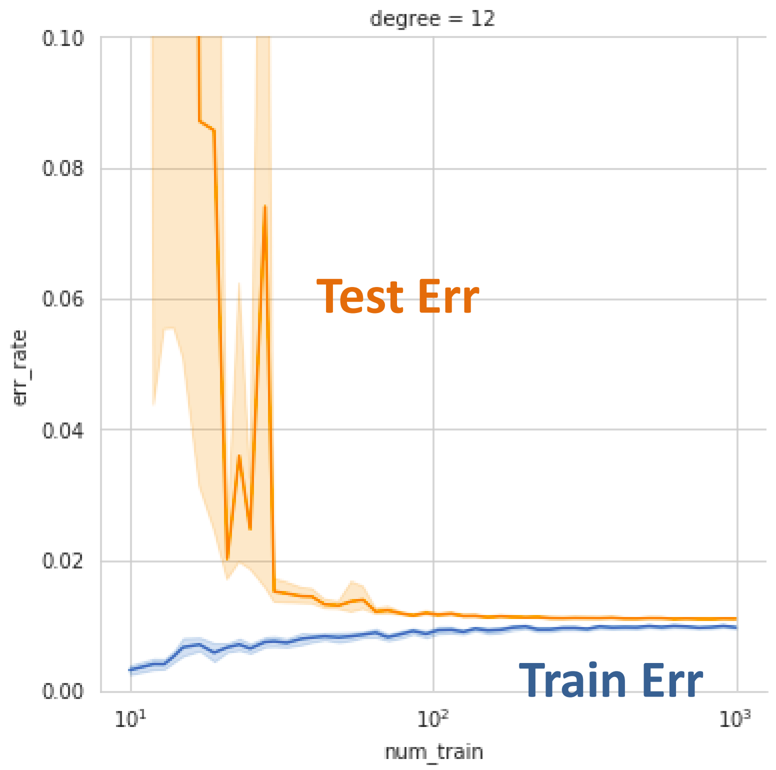
[in ML]: a model whose size DOES NOT grow with amount of data

Non-parametric models (e.g. k-NN)**:**

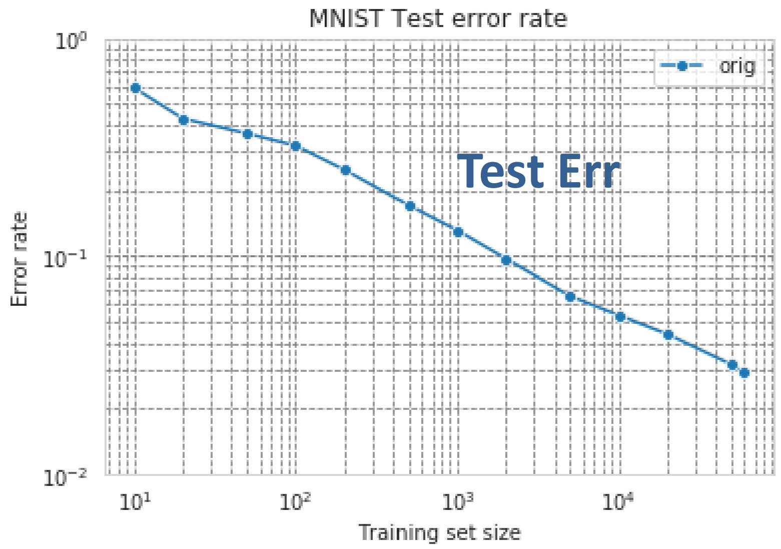
A model whose size DOES grow with amount of data

Model performance vs dataset size

Parametric



Non-parametric



Generative models

E.g.: Naive Bayes, Gaussian Discriminant analysis

Model the data generation process $p(x|y)$

Predict using the Bayes theorem

$$p(y|x) = \frac{1}{Z} p(x|y)p(y)$$

More info: NB, GD

Naive Bayes

y: discrete class labels

x: typically binary (e.g. presence/absence of a word)

Data generation process:

$p(y = k) = \pi_k$

$p(x|y) \approx \prod_i p(x_i|y)$

$p(x_i = 1|y = k) = p_{ik}$

Parameters:

π_k, p_{ik}

Parameter estimation: max likelihood, take observed counts from data

Hyperparameters:

Laplace smoothing (psuedocounts)

GDA

y is binary
x is a real-valued vector

Data-generation model

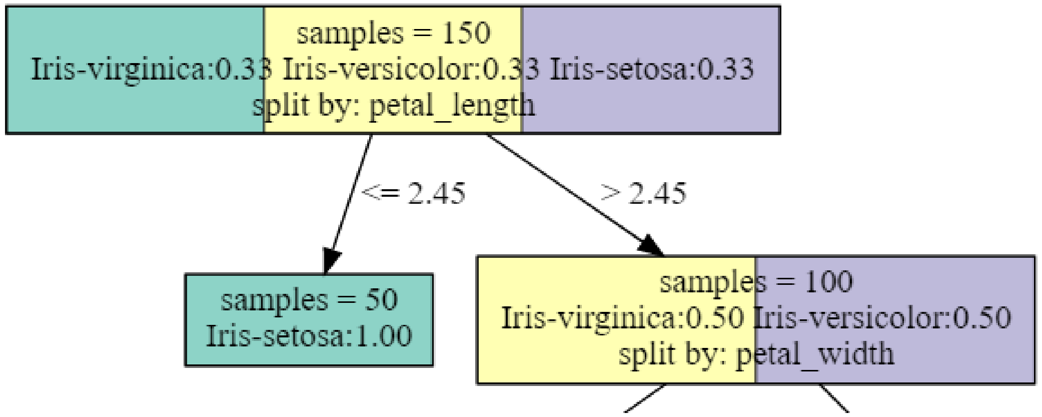
$$\begin{aligned} y &\sim \text{Bernoulli}(\phi) \\ x|y=0 &\sim \mathcal{N}(\mu_0, \Sigma_0) \\ x|y=1 &\sim \mathcal{N}(\mu_1, \Sigma_1) \end{aligned}$$

Parameters:

$$\phi, \mu_0, \Sigma_0, \mu_1, \Sigma_1$$

Parameters set using max-likelihood by observind data counts, means, and correlations

Decision tree



- Each node implemets a test
- Subtrees contain fractions of data
- Leaf nodes give classification details
- Parameters: structure, tests, class counts in leaves
- [unprunned] tree grows with data -> nonparametric

Tree building

Greedy node splitting:

- Go over all possible tests, compute their purity
- Choose the maximally pure one
- Split the data and recurse into subtrees

Pruning:

- Build a full tree
- Evaluate accuracy in all nodes using:
    - The upper-confidence bound heuristic (C4.5)
    - Cross-validation (fix tree structure, fit class counts in leaves and classify the test set)
- Remove nodes that are less accurate than their parent

Ensembles

Average predictions of many models.

For best results:

- Each model should be strong
- Model errors should be uncorrelated

Note:

strong models are correlated (they agree on samples correctly classified)

Bagging: Bootstrap Aggregation

Decorrelate the models by varying training data

- Draw a bootstrap training sample
- Train the model

Final model simply averages predictions!

Bonus: OOB (Out Of Bag) error estimate:

For each model, record its predictions on the data not in the bootstrap sample. Aggregate the predictions across all models.

Random forest

Average many decision trees.

Core ideas:

1. Bagging: train each tree on a **random** subset of train data (decorrelate trees)
2. Random tree: select each choice from a few **randomly sampled** features (decorrelate trees)
3. Unpruned trees: pruning increases train errs, correlates trees

Random forest

Practical aspects:

- OOB error estimates
- Attribute importance metrics
- Few hyperparams to tune

2nd best all-around classifier after boosted trees

Boosted Trees

Core idea:
Combine many weak (shallow) trees into one strong classifier.

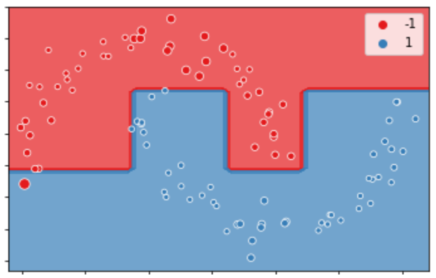
Algorithm loop:

1. Train a tree
2. Reweight the dataset to boost misclassified data

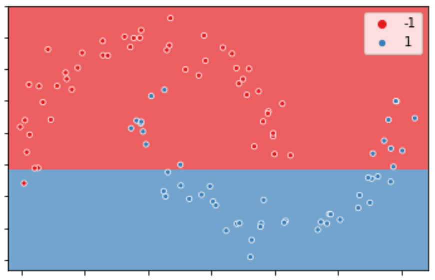
Finally:
Combine all trees

Note:
Trees are decorrelated.
Each tree fixes errors of previous ones!

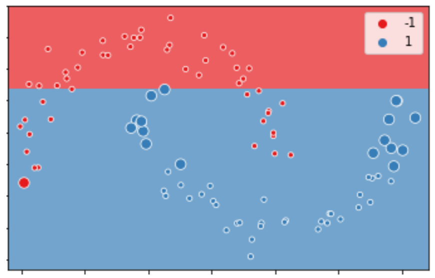
Adaboost



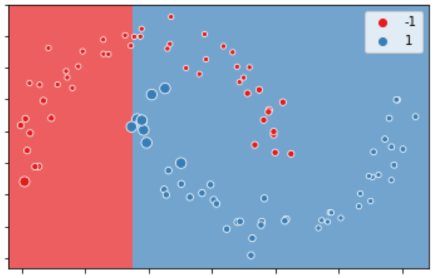
$= \alpha_1$



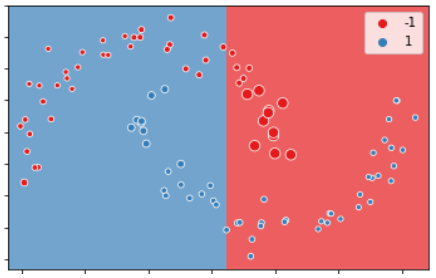
$+\alpha_2$



$+\alpha_3$



$+\alpha_4$



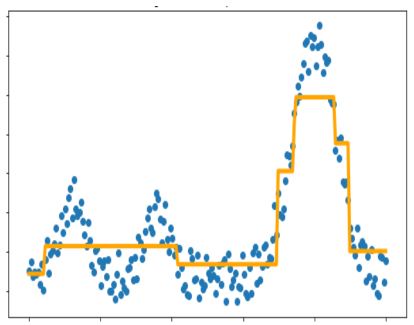
```
$w_i = 1/N$;
for $m = 1 : M$ do
    Fit a classifier $\phi_m(\mathbf{x})$ to the training set using weights $\mathbf{w}$;
    Compute $\text{err}_m = \frac{\sum_{i=1}^N w_{i,m}\mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))}{\sum_{i=1}^N w_{i,m})}$;
    Compute $\alpha_m = \log[(1 - \text{err}_m)/\text{err}_m]$;
    Set $w_i \leftarrow w_i \exp[\alpha_m \mathbb{I}(\tilde{y}_i \neq \phi_m(\mathbf{x}_i))]$;
Return $f(\mathbf{x}) = \text{sgn}\left[\sum_{m=1}^M \alpha_m \phi_m(\mathbf{x})\right]$;
```

Gradient boosting

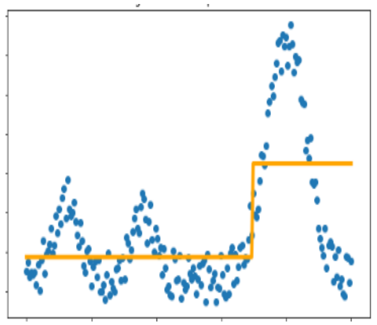
Generalize Adaboost to other tasks

$$f(x) = \alpha_1 \phi(x, \gamma_1) + \alpha_2 \phi(x, \gamma_2) + \alpha_3 \phi(x, \gamma_3) + \cdots$$

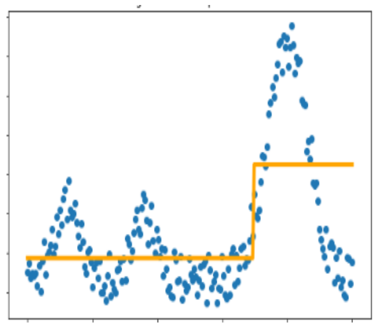
where ϕ is a model with parameters γ_i



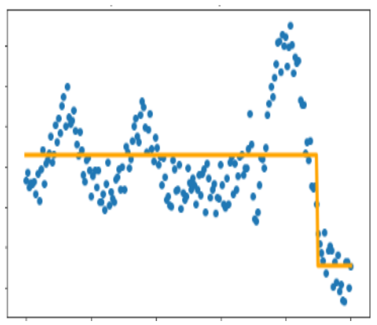
$= \alpha_1$



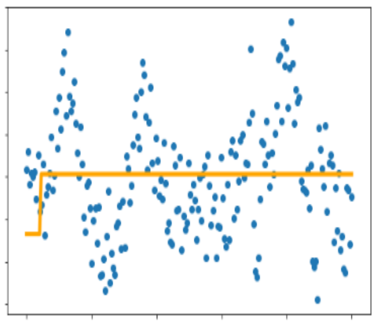
$+\alpha_2$



$+\alpha_3$



$+\alpha_4$



$+\ldots$

Gradient boosting

Generalize Adaboost to other tasks

$$f(x) = \alpha_1 \phi(x, \gamma_1) + \alpha_2 \phi(x, \gamma_2) + \alpha_3 \phi(x, \gamma_3) + \cdots$$

where ϕ is a model with parameters γ_i

Core intuitions:

Gradient descent in the space of functions

(Repeat adding models that correlate with gradient of the loss)

Weights α are sparse: from all functions (all possible parameters γ) we select only a few.

Gradient boosting

Generalize Adaboost to other tasks

$$f(x) = \alpha_1 \phi(x, \gamma_1) + \alpha_2 \phi(x, \gamma_2) + \alpha_3 \phi(x, \gamma_3) + \cdots$$

where ϕ is a model with parameters γ_i

```
Initialize $f_0(\mathbf{x}) = \text{argmin}_{\gamma} \sum_{i=1}^{N} L(y_i, \phi(\mathbf{x}_i; \boldsymbol{\gamma}))$;
for $m = 1 : M$ do
    Compute the gradient residual using $r_{im} = -\left[\frac{\partial L(y_i, f(\mathbf{x}_i))}{\partial f(\mathbf{x}_i)}\right]_{f(\mathbf{x}_i)=f_{m-1}(\mathbf{x}_i)}$;
    Use the weak learner to compute $\boldsymbol{\gamma}_m$ which minimizes $\sum_{i=1}^{N}(r_{im} - \phi(\mathbf{x}_i; \boldsymbol{\gamma}_m))^2$;
    Update $f_m(\mathbf{x}) = f_{m-1}(\mathbf{x}) + \nu\phi(\mathbf{x}; \boldsymbol{\gamma}_m)$;
Return $f(\mathbf{x}) = f_M(\mathbf{x})$
```

Linear and Logistic regression

Linear regression

y is continuous

$$p(y|x) = \mathcal{N}\left(\mu = \Theta^{\mathrm{T}}\mathrm{x}, \Sigma\right)$$

Logistic regression

y is binary

$$p(y = 1|x) = \sigma(\Theta^T x)$$
$$= \frac{1}{1 + e^{-\Theta^T x}}$$

Both

Parameters: Θ

Train to minimize negative log-lileklihood with regularization

$$\Theta = \arg\min_{\Theta} \sum_i - \log p(y = y^{(i)}|x^{(i)}) \ + \frac{\lambda}{2} \sum_j \theta_j^2$$

Practical aspects of regression training

Linear and logstic regression take numbers and are scale sensitive.

Normalize inputs:

- Map each input to $[-1,1]$ range
- Or scale each input to have mean 0, variance 1
- Even better: use PCA to de-correlate the inputs
- For discrete inputs:
  - 1-of-N: $'a' \rightarrow [1,0,0], 'b' \rightarrow [0,1,0], 'c' \rightarrow [0,0,1]$
    please note: this turns the matrix multiplication into a look-up table. You get a score for each category.
- Specific cases:
  - Thermometer : $1 \rightarrow [1,0,0], 2 \rightarrow [1,1,0], 3 \rightarrow [1,1,1]$
  - Transform angles using trig. functions, e.g.
    $\alpha \rightarrow [\sin(\alpha), \sin(\alpha + 120°), \sin(\alpha + 240°)]$
  - Similar approach possible for other periodic inputs!

Probabilistic view of regularization

- As we have seen, too „flexible” models are prone to overtraining.
- We need to prefer some hypotheses over others
  - Examples:
    - Linear models are simpler than polynomial
    - Small neural net is simpler than a large one
- Regularization serves to express our preferences about model simplicity
- Typically, we assign a **prior probability** to our models:

$$P(\mathbf{\Theta}) = \prod_{i=1}^{n} \mathcal{N}(\mathbf{\Theta}_i; \mu = 0, \sigma = \lambda)$$

Bayes theorem

$$\overset{\text{posterior}}{p(A|B)} = \frac{\overset{\text{support of B for A}}{p(B|A)}\,\overset{\text{Prior}}{p(A)}}{p(B)}$$

Interpretation: how our estimate of A changes after seeing B.

Why?

$$p(A, B) = p(A|B)p(B) = p(B|A)p(A)$$

Then divide by $p(B)$

Bayesian approach to ML

- What is the model probability after seeing the data $\mathcal{D}$?

$$p(\Theta|\mathcal{D}) = \frac{p(\mathcal{D}|\Theta)p(\Theta)}{p(\mathcal{D})}$$

How to make predictions? Integrate over all models:

$$p(y|x,\mathcal{D}) = \int_{\Theta} p(y|x,\Theta)p(\Theta|\mathcal{D})d\Theta$$

Then

$$E[y|x,\mathcal{D}] = \int_{y} yp(y|x,\mathcal{D})dy$$

But computing $p(y|x,\mathcal{D})$ is often intractable :(

Maximum-a-posteriori

- Instead of integrating over all Θ
- Use the maximally probable Θ:

$$\Theta_{MAP} = \arg\max_{\Theta} p(\Theta|\mathcal{D})$$

$$= \arg\max_{\Theta} \left(\prod_{i=1}^{m} p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) \right) p(\Theta)$$

- It's like Max. Likelihood with the extra term (which is the regularization).

Gaussian model MAP

$$\arg\max_{\Theta} \prod_{i=1}^{m} p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) p(\Theta) =$$

$$\arg\max_{\Theta} \sum_{i=1}^{m} \log p\left(y^{(i)} \middle| x^{(i)}, \Theta\right) + \log\left(p(\Theta)\right)$$

Now if Θ_j are Gaussian with zero-mean,

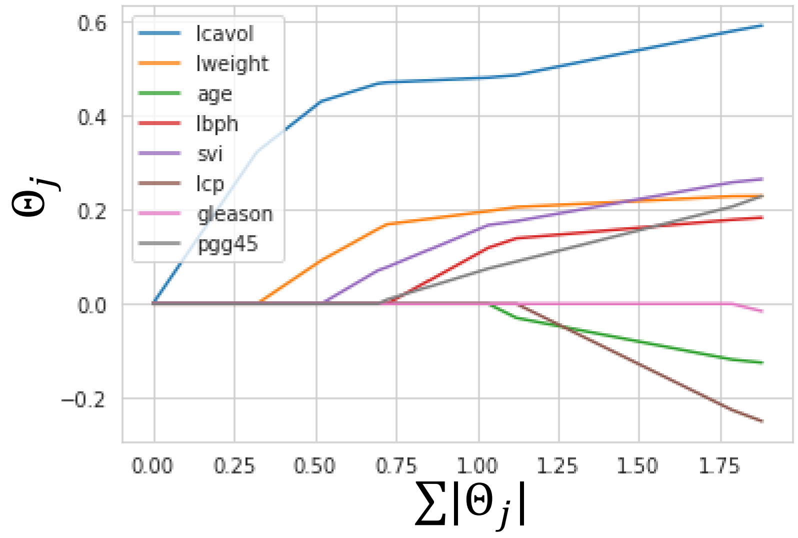
$$\log\left(p(\Theta)\right) \propto \sum_{j=1}^{n} \left(\Theta_j\right)^2$$

Other priors are possible

Assume

$$p(\Theta_j) \propto e^{-\lambda|\Theta_j|}$$

Then $-\log p(\Theta_j) = \lambda|\Theta_j|$

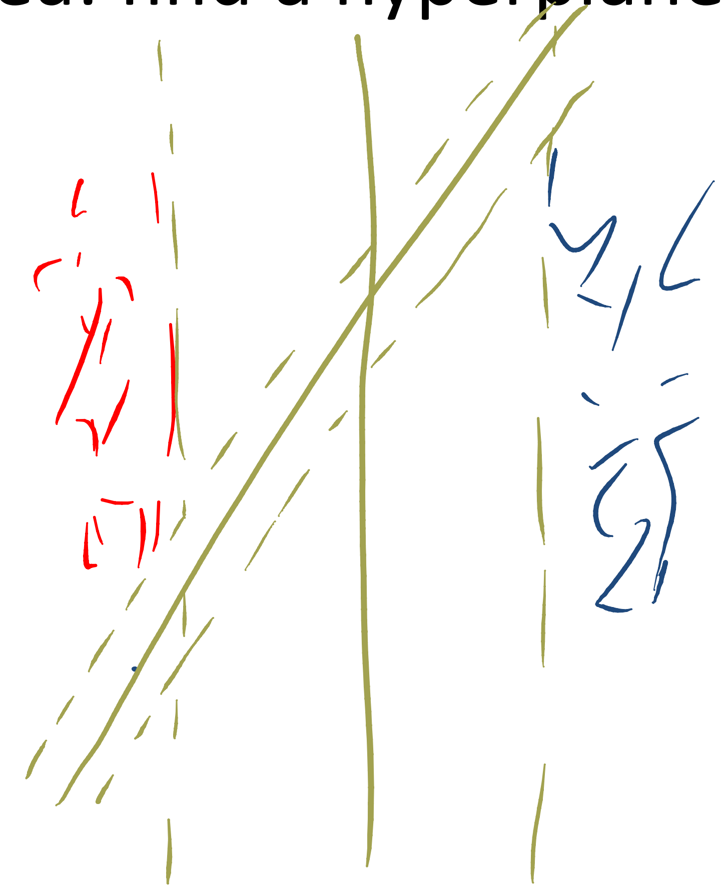


This yields LASSO Regression

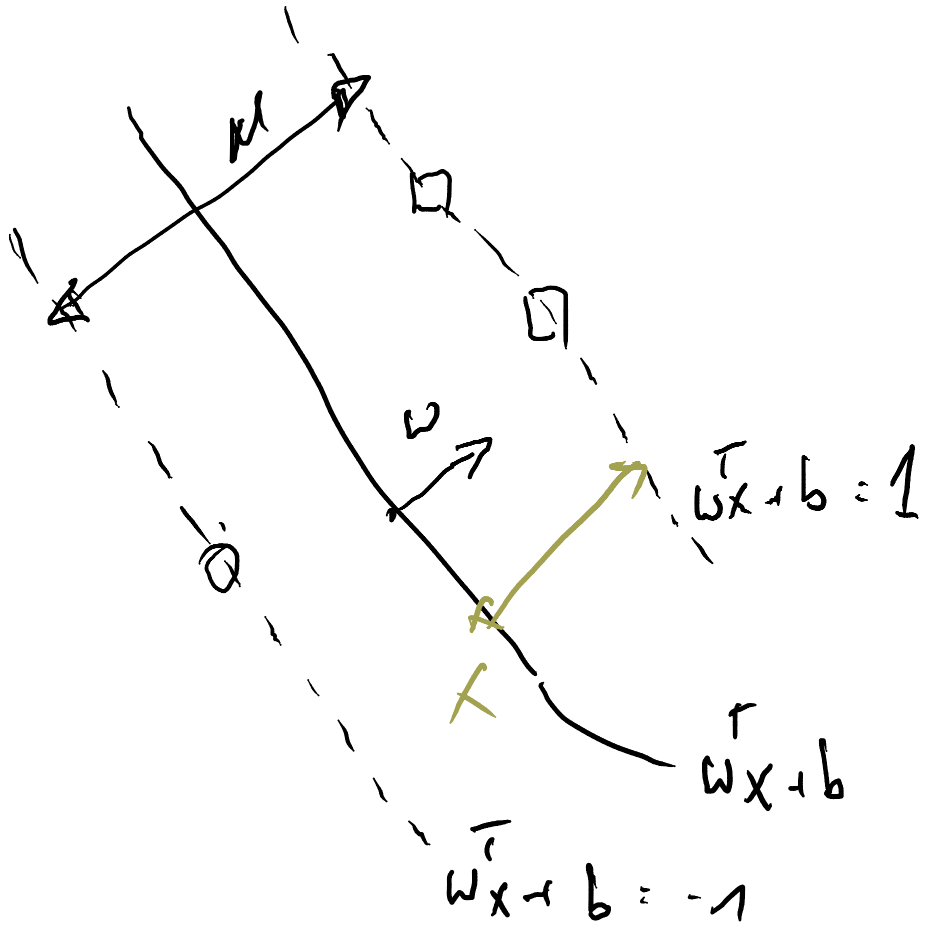
$$\Theta = \arg\min_{\Theta} \sum_i -\log p(y = y^{(i)}|x^{(i)}) + \lambda \sum_j |\Theta_j|$$

Longer example: SVM

- Task: 2-class classification
- Idea: find a hyperplane yielding max margin



The margin



$$w^T\left(x + \frac{Mw}{2\|w\|}\right) + b =$$

$$= w^t x + b + \frac{M\|w\|^2}{2\|w\|} =$$

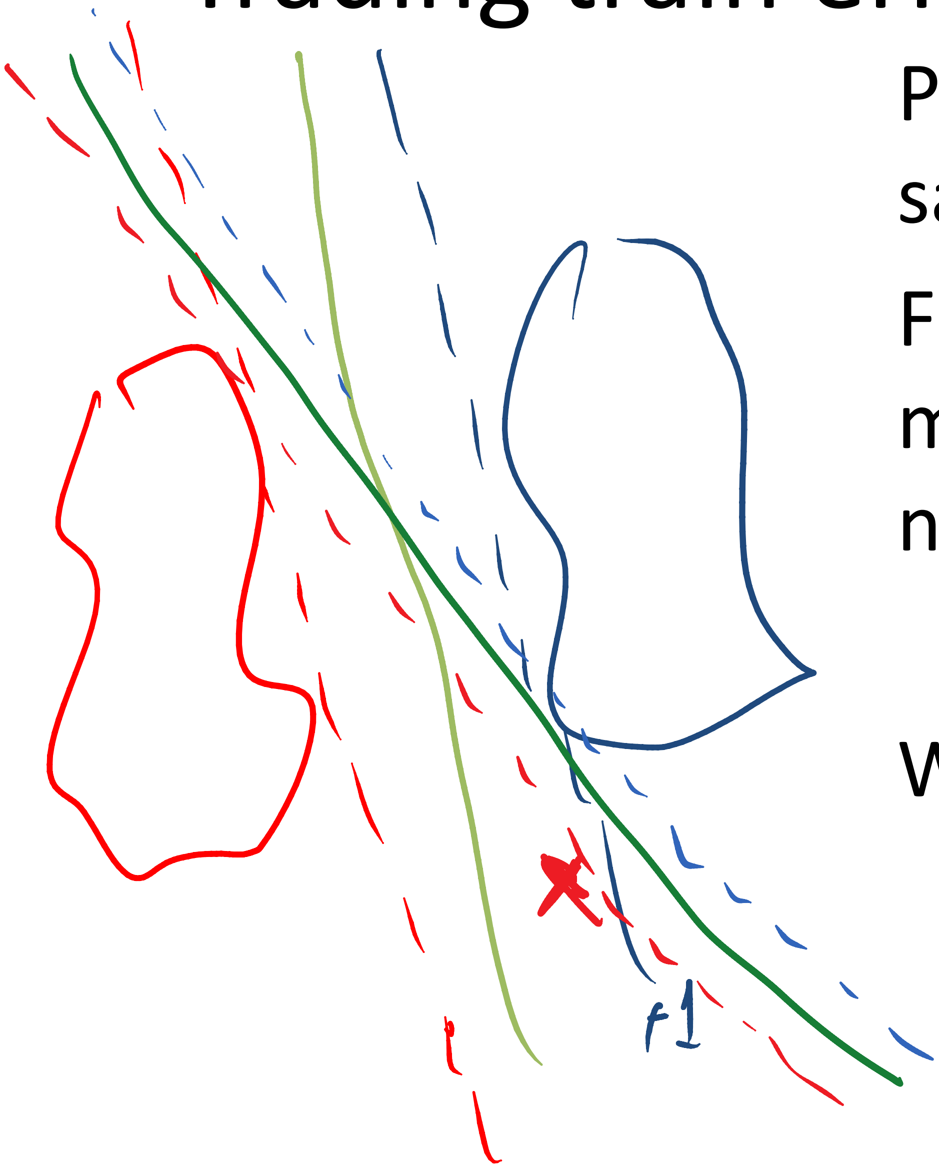
$$= \frac{M}{2}\|w\| = 1$$

Thus:

$$M = \frac{2}{\|w\|}$$

Maximum margin => minimum weights!

Trading train error for margin



Penalize errors and samples inside the margin

Find a tradeoff between margin width and number of errors!

We want:

$$y^{(i)}\left(\boldsymbol{w}^T \boldsymbol{x}^{(i)} + b\right) \geq 1 - \xi_i$$

Soft-Margin SVM

The SVM finds weights such to minimize

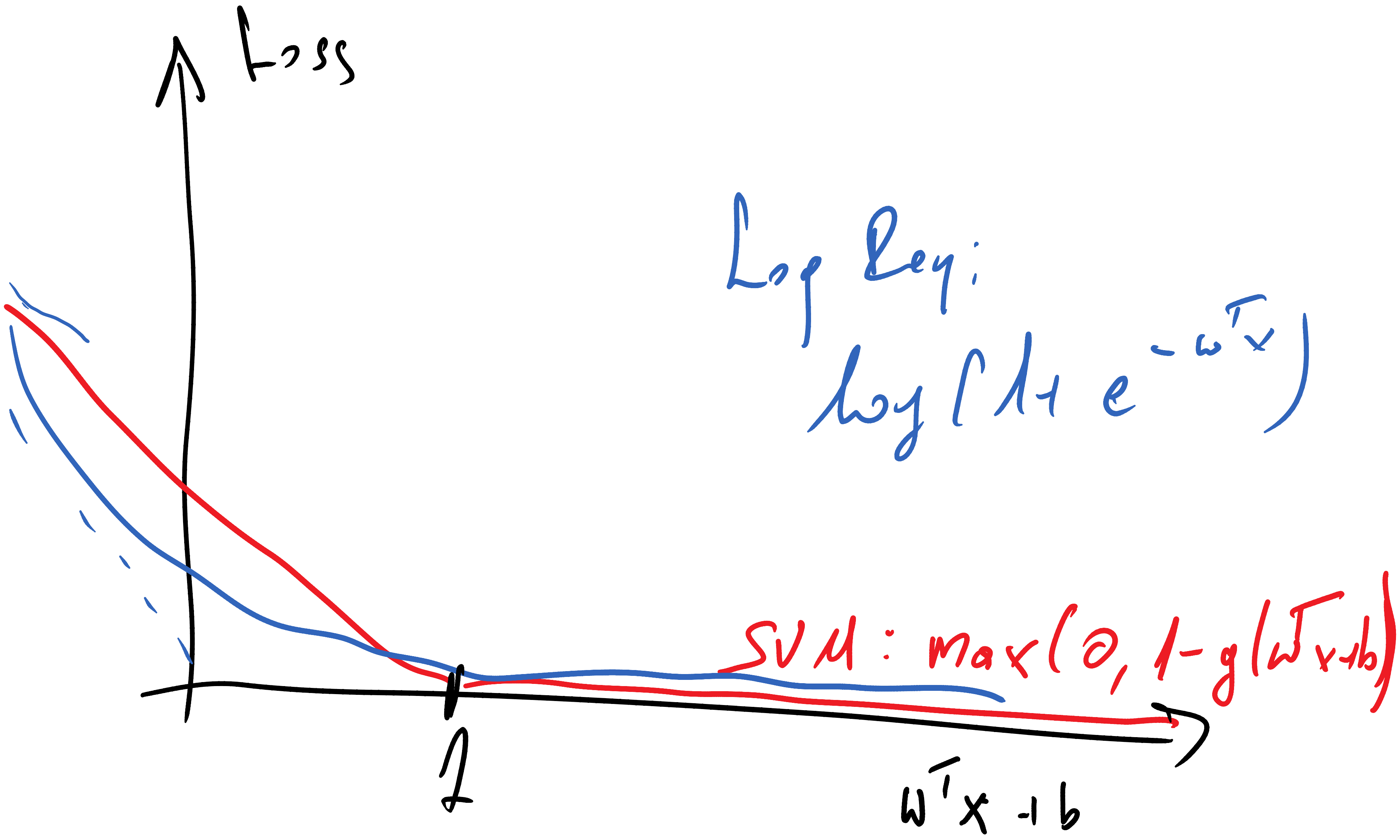
$$\frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_i \xi_i$$

$$\text{s.t.: } y^{(i)}\left(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b\right) \geq 1 - \xi_i \text{ and } \xi_i \geq 0 \;\forall i$$

Alternative formulation

$$\frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_i \max\left(0, 1 - y^{(i)}\left(\boldsymbol{w}^T\boldsymbol{x}^{(i)} + b\right)\right)$$

Note: Soft-Margin SVM and LogReg



Kernels: smart data transformations

For SVM, linear and logistic regression we can express weights as a linear combination of training samples:

$$\boldsymbol{w} = \sum_i \alpha_i \boldsymbol{x}^{(i)}$$

Technical note: α_i are the Lagrange multipliers of constraints

Kernels: observation

Assume

$$\boldsymbol{w} = \sum_i \alpha_i \boldsymbol{x}^{(i)}$$

Map (nonlinearly) $x \to \phi(x)$

$$\boldsymbol{w} = \sum_i \alpha_i \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$f(\boldsymbol{x}) = w^T \phi(\boldsymbol{x}) + b = \sum_i \alpha_i \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi(\boldsymbol{x}) + b$$

We only need dot-products in the feature $\boldsymbol{\phi}(\cdot)$ space.

Kernels: smart dot-products

Map (nonlinearly) $x \to \phi(x)$

We only need $\phi(\boldsymbol{x})^T \phi(\boldsymbol{y})$

Kernels compute this **in a smart way**:

$$K(\boldsymbol{x}, \boldsymbol{y}) = \phi(\boldsymbol{x})^T \phi(\boldsymbol{y})$$

K is the **kernel function**.

We never need to compute $\phi(\boldsymbol{x})$.

This often leads to speedups.

Exemplary Kernels

Gaussian: $K(\boldsymbol{x}, \boldsymbol{y}) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{y}\|}{2\sigma^2}\right)$

NB: this corresponds to an infinite feature space expansion!

Polynomial: $K(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}^T\boldsymbol{y} + c)^d$

SVM ♥ Kernels

$$\boldsymbol{w} = \sum_i \alpha_i \phi\left(\boldsymbol{x}^{(i)}\right)$$

$$f(\boldsymbol{x}) = w^T \phi(\boldsymbol{x}) + b = \sum_i \alpha_i \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi(\boldsymbol{x}) + b$$

For SVM, most of α_i are 0!

Only the Support Vectors have $\alpha_i > 0$!

Need to store ony a fraction of training data.

Kernels – further intuitions

Kernels take a **parametric** model
and make it **nonparametric.**

Each data sample has an α_i
more data-> more params.

Kernels can encode our background knowledge
– they are like a similarity measure.

PRACTICAL ASPECTS OF LEARNING

What’s the goal?

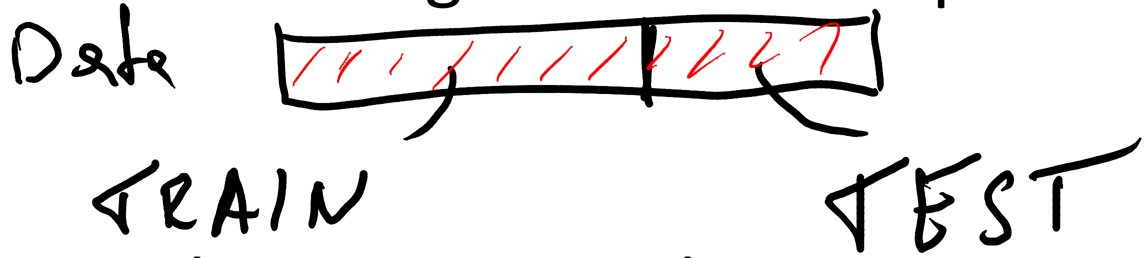
The **ultimate** goal of learning: do well on **new/unseen** data

ML projects are defined by **metrics** and **test sets.**

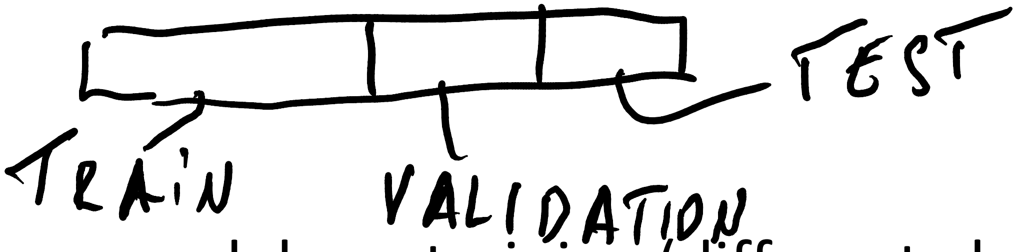
Honest estimates: Hold-out set

Large data case!!!

- Split the training data into two parts:



- Train only on training, then test on testing.
- Often we do a three-way split:



- Then:
  - Train many models on training (different algos, parameters)
  - Use validation to choose best model
  - Test on testing

Cross-validation

Small data case!!

- Hold-out set makes inefficient data use
- Idea:
  - Divide the data into k sets (~5,10)

  For i=1..k

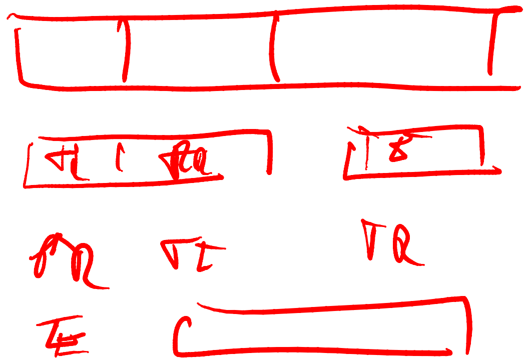
    Train on all but the i-th set

        may further split to choose the model…

    Test on the i-th set

  Finally:

  take the answers on the testing sets and use them to compute the performance measures
- Extreme case: leave-one-out (jackknife) – always use all but one sample to train!

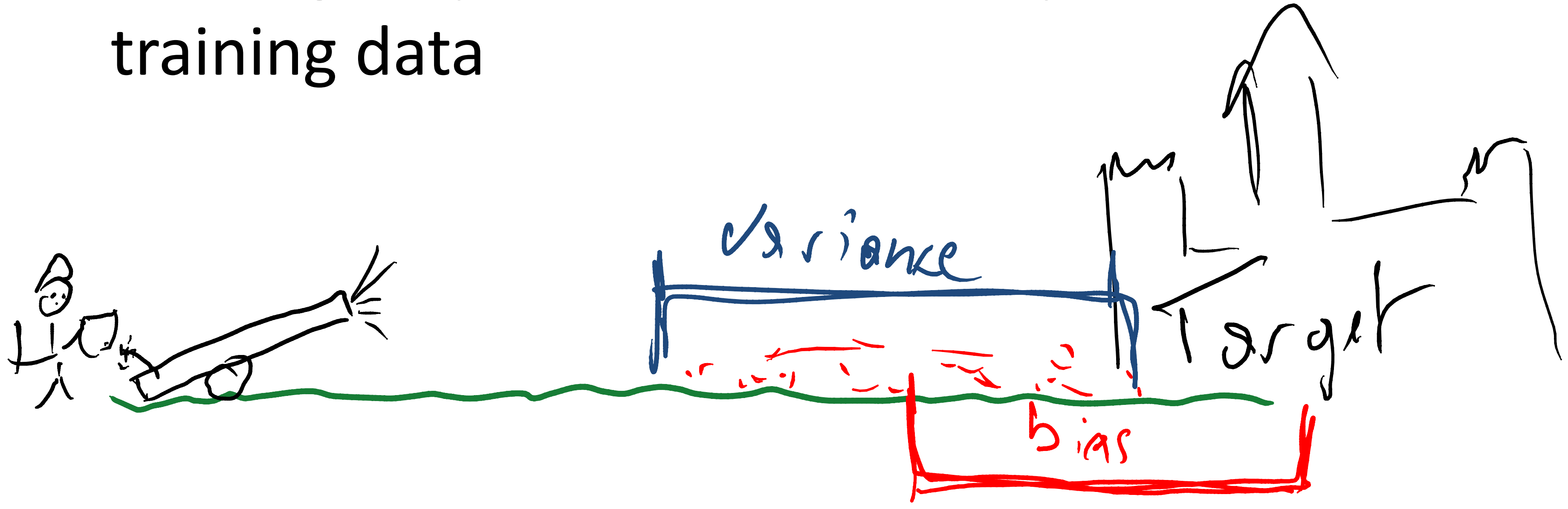


Bootstrap

- Small data case!!
- Sample with replacement m samples
  - About 37% will not be selected
- Train on the selected samples
- Test on the remaining ones
- Optionally repeat.

Bias-Variance: two sources of error!

- The **bias** captures how well our family of functions (hypothesis space) matches the data.
- The **variance** captures how the results of training vary with different samples from the training data



How to lower the bias?

- Choose more powerful/better models:
  - Understand the data and choose a matching model
  - Describe the data with more attributes
  - Expand the data, e.g.:
$$x \rightarrow [1, x, x^2, x^3, \dots]$$

- This usually increases the Hypothesis space

How to lower variance?

- Get more data (or generate synthetic, e.g. rotate and shear pictures)
- Select only the most important inputs
- **Constrain the models:**
  - Simpler models
  - Regularize the models:
    Assign a probability distribution to the models and choose the most probable ones
- **Average the models**
  - Very powerful
  - Also called “ensemble learning”, boosting, bagging
  - Requires that the models make uncorrelated errors

Approximations we take

- We want accuracy on UNKNOWN TEST DATA
- Approximation: Cross-Validation, hold-out set
- Can’t directly optimize acc (non-differentiable, NP-hard…)
- Proxy: optimize a loss function
- Often impossible exactly –use some greedy algo

Errors can come at all stages

- Data:
  - Is it representative of the problem
  - Does it cover all possible variations (e.g. in France “z” is ʒ)
  - Can you get more of it? Generate? Transform?

  z
- Prior beliefs:
  - Does the architecture you choose match the problem?
  - Maybe you know something (e.g. invariants, predominating probability distribution…)
- Loss function:
  - Does it make sense? Is it for classification/regression? Do smaller loss correspond to better performance?
- Training algorithm:
  - Do you reach the minimum of what you optimize?
  - Intentionally? How about early stopping?
- Performance measures:
  - do you separate train from test data?
  - How do train and test errors compare?

Example

Logistic regression classifier makes 10% errors

SVM with Gaussian Kernel makes 20% ☹

- Use the same loss – take linear and nonlinear SVM, which one has the lowest? (don't change training, just loss computation):
  - Linear classifier -> is the nonlinear SVM correct??
    - Maybe it is too regularized?
  - Nonlinear -> how is your train and test error, do you over-fit?
    - Do you use regularization? Can you increase it?
    - Can you get more training data?
    - Maybe the linear classifier is also over-fitting?