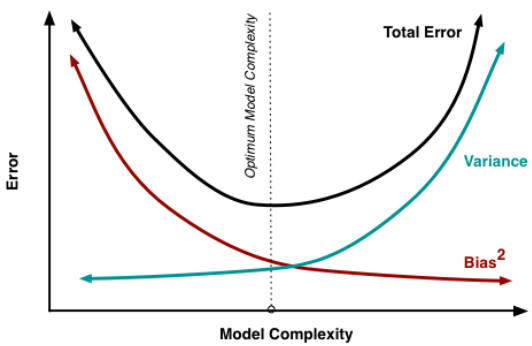
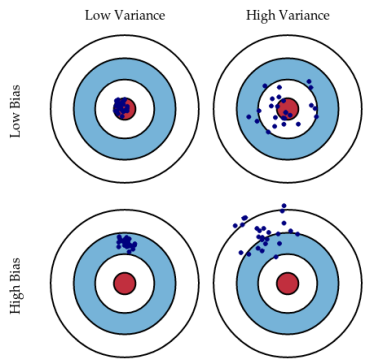
Machine Learning

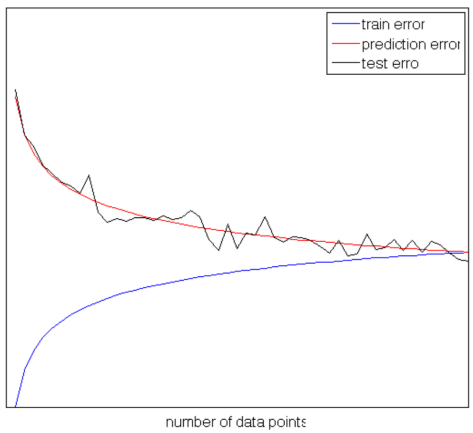
"No Free Lunch" Theorems
Bias-Variance and Model Selection

Marcello Restelli

April 21, 2020







Outline

1. "No Free Lunch" Theorems
2. Bias-Variance Tradeoff
3. Model Selection
   - Feature Selection
   - Regularization
   - Dimension Reduction
4. Model Ensembles
   - Bagging
   - Boosting

"No Free Lunch" Theorems

$$
\begin{aligned}
Acc_G(L) &= \textbf{Generalization accuracy} \text{ of learner } L \\
&= \text{Accuracy of } L \text{ on } \textbf{non-training} \text{ examples} \\
\mathcal{F} &= \text{set of all possible concepts, } y = f(\mathbf{x})
\end{aligned}
$$

Theorem

For any learner L, $\dfrac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} Acc_G(L) = \dfrac{1}{2}$

(given any distribution $\mathcal{P}$ *over* $\mathbf{x}$ *and training set size* N*)*

Proof Sketch.

- Given any training set $\mathcal{D}$
- For every concept f where $Acc_G(L) = 0.5 + \delta$
- There is a concept f' where $Acc_G(L) = 0.5 - \delta$
- $\forall \mathbf{x} \in \mathcal{D}$, $f'(\mathbf{x}) = f(\mathbf{x}) = y$; $\forall \mathbf{x} \notin \mathcal{D}$, $f'(\mathbf{x}) \neq f(\mathbf{x})$

"No Free Lunch" Theorems

$$
\begin{aligned}
Acc_G(L) &= \textbf{Generalization accuracy} \text{ of learner } L \\
&= \text{Accuracy of } L \text{ on } \textbf{non-training} \text{ examples} \\
\mathcal{F} &= \text{set of all possible concepts, } y = f(\mathbf{x})
\end{aligned}
$$

Theorem

For any learner L, $\dfrac{1}{|\mathcal{F}|}\sum_{\mathcal{F}} Acc_G(L) = \dfrac{1}{2}$

(given any distribution $\mathcal{P}$ *over* $\mathbf{x}$ *and training set size* N*)*

Corollary

- *For any two learners* L_1, L_2*:*
- ***If*** $\exists$ *learning problem s.t.* $Acc_G(L_1) > Acc_G(L_2)$
- ***then*** $\exists$ *learning problem s.t.* $Acc_G(L_2) > Acc_G(L_1)$

What Does This Means in Practice?

- Don't expect that your favorite learner to **always** be the best
- Try different approaches and **compare**
- But how could (say) a deep neural network be **less accurate** than a single-layer one?

Bias-Variance Decomposition

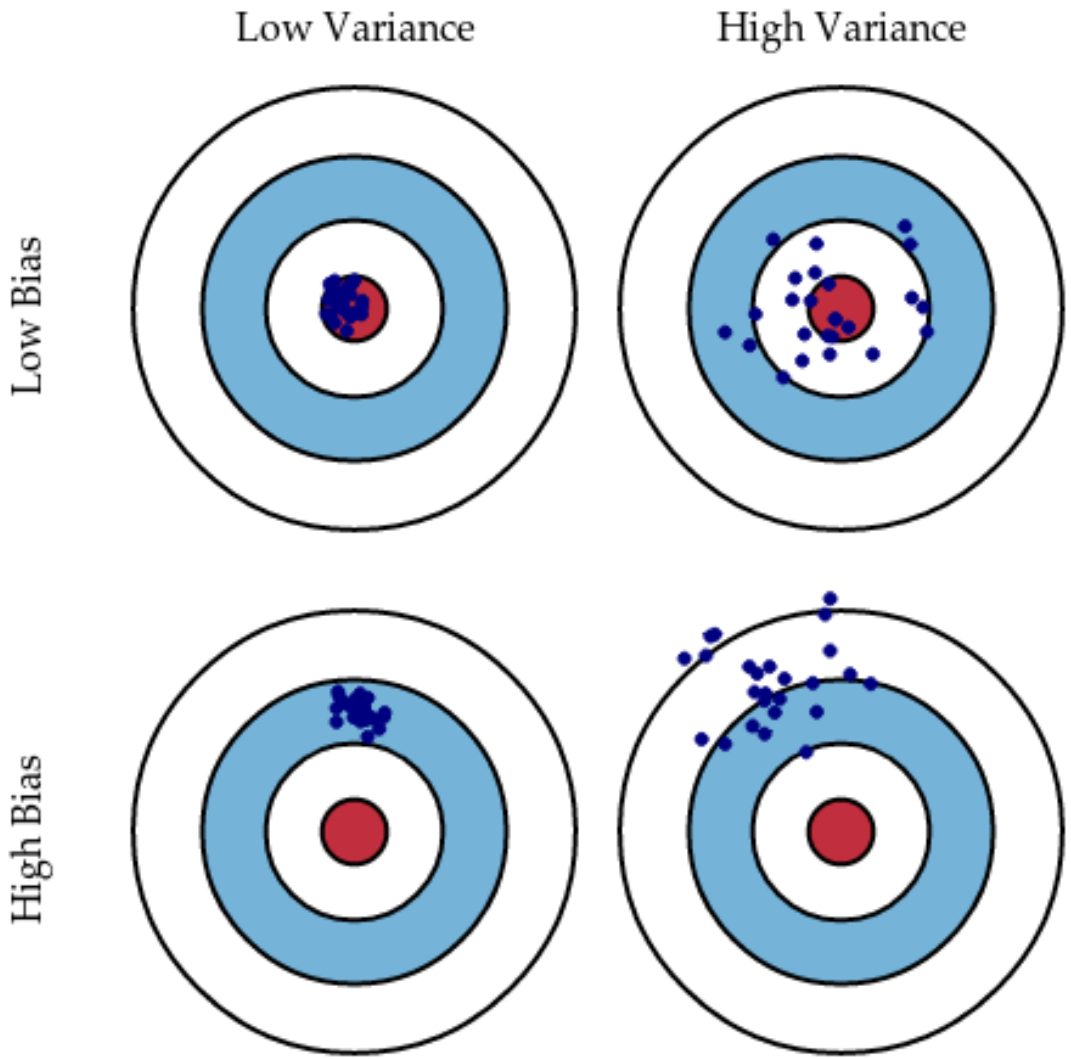
- Assume that we have a data set $\mathcal{D}$ with N samples obtained by a function $t_i = f(\mathbf{x}_i) + \epsilon$
- $\mathbb{E}[\epsilon] = 0$ and $Var[\epsilon] = \sigma^2$
- We want to find a model $y(\mathbf{x})$ that **approximates** f as well as possible
- Let's consider the **expected square error** on an unseen sample $\mathbf{x}$

$$
\begin{aligned}
\mathbb{E}[(t - y(\mathbf{x}))^2] &= \mathbb{E}[t^2 + y(\mathbf{x})^2 - 2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] + \mathbb{E}[y(\mathbf{x})^2] - \mathbb{E}[2ty(\mathbf{x})] \\
&= \mathbb{E}[t^2] \pm \mathbb{E}[t]^2 + \mathbb{E}[y(\mathbf{x})^2] \pm \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + \mathbb{E}[t]^2 + Var[y(\mathbf{x})] + \mathbb{E}[y(\mathbf{x})]^2 - 2f(\mathbf{x})\mathbb{E}[y(\mathbf{x})] \\
&= Var[t] + Var[y(\mathbf{x})] + (f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})])^2 \\
&= \underbrace{Var[t]}_{\sigma^2} + \underbrace{Var[y(\mathbf{x})]}_{\text{Variance}} + \underbrace{\mathbb{E}[f(\mathbf{x}) - y(\mathbf{x})]^2}_{\text{Bias}^2}
\end{aligned}
$$

- Expectation is performed over different realizations of the training set $\mathcal{D}$

Bias-Variance Decomposition

Graphical Definition



Bias

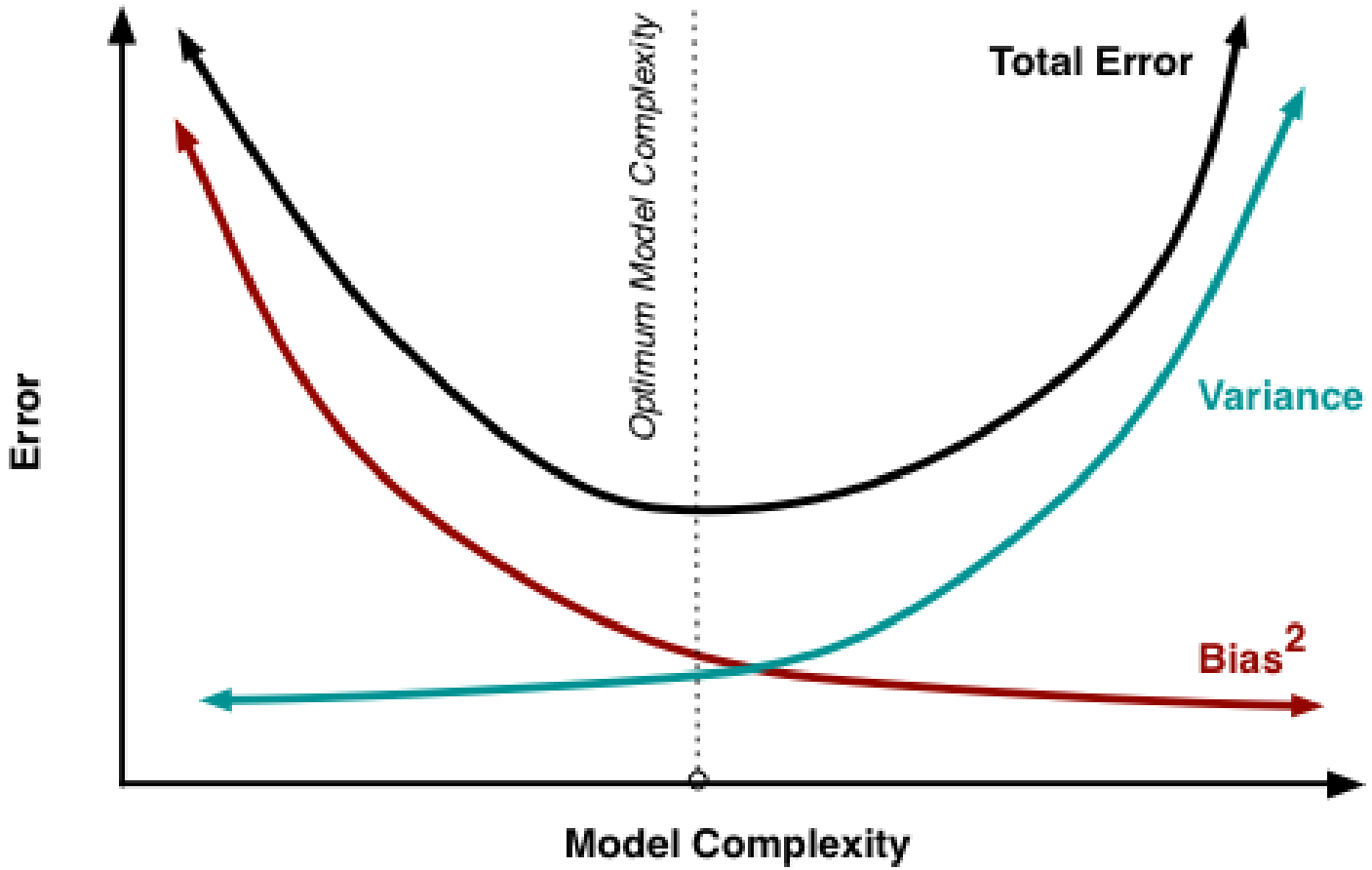
- If you sample a data set $\mathcal{D}$ **multiple times** you expect to learn a different $y(\mathbf{x})$
- **Expected hypothesis** is $\mathbb{E}[y(\mathbf{x})]$
- **Bias**: difference between the truth and what you expect to learn
  - $\text{bias}^2 = \int \left(f(\mathbf{x}) - \mathbb{E}[y(\mathbf{x})]\right)^2 p(\mathbf{x}) \mathrm{d}\mathbf{x}$
  - Decreases with **more complex models**

Variance

- **Variance**: difference between what you learn from a particular data set and what you expect to learn
  - variance $= \int \mathbb{E}\left[(y(\mathbf{x}) - \overline{y}(\mathbf{x}))^2\right] p(\mathbf{x}) \mathrm{d}\mathbf{x}$
  - where $\overline{y}(\mathbf{x}) = \mathbb{E}[y(\mathbf{x})]$
  - Decreases with **simpler models**
  - Decreases with **more samples**

Bias-Variance Decomposition

Graphical Definition



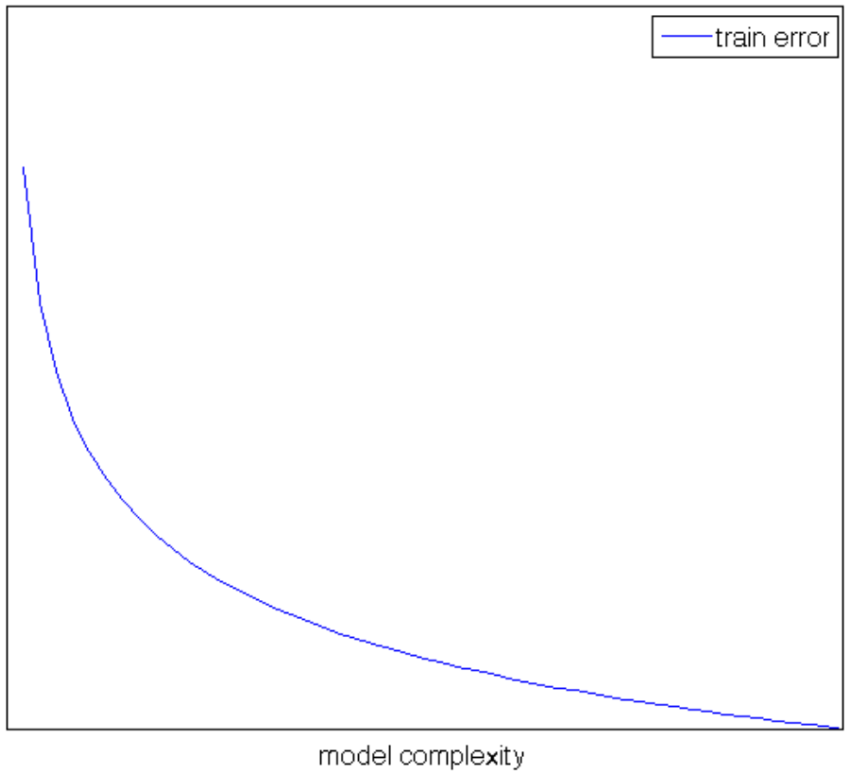
Training Error

- Given a data set $\mathcal{D}$ with N samples
- Chose a loss function (e.g., RSS)
- Training set error
  - Regression

$$L_{train} = \frac{1}{N} \sum_{n=1}^{N} (t_n - y(\mathbf{x}_n))^2$$

  - Classification

$$L_{train} = \frac{1}{N} \sum_{n=1}^{N} (I(t_n \neq y(\mathbf{x}_n)))$$



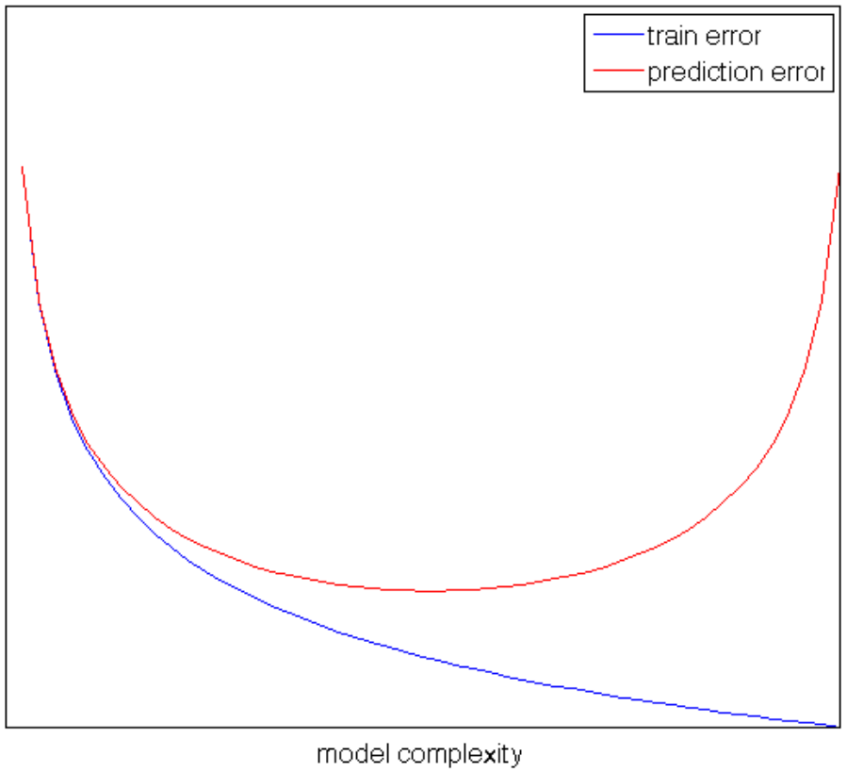
Prediction Error

- Training error is **not** necessary a good measure
- We care about the error over **all** input points
  - Regression

$$L_{true} = \int \left(f(\mathbf{x}) - y(\mathbf{x})\right)^2 p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

  - Classification

$$L_{true} = \int I\left(f(\mathbf{x}) \neq y(\mathbf{x})\right) p(\mathbf{x}) \mathrm{d}\mathbf{x}$$

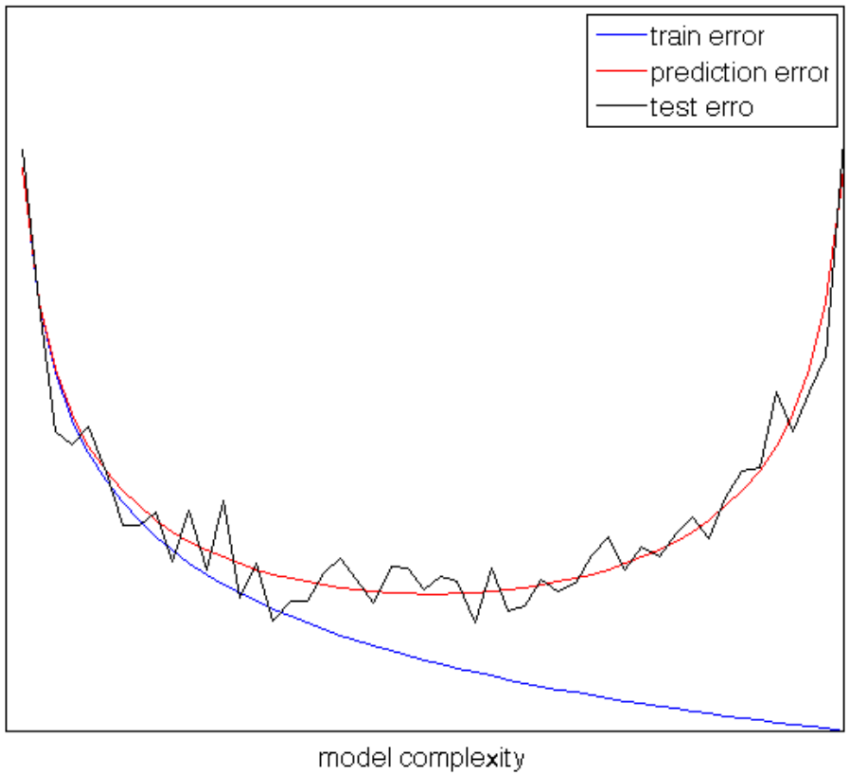


- Training error is an **optimistically biased** estimate of prediction error

Train-Test

- In practice
  - **Randomly** divide the data set into test and train
  - Use training data to **optimize parameters**
  - Use test data to **evaluate prediction error**

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

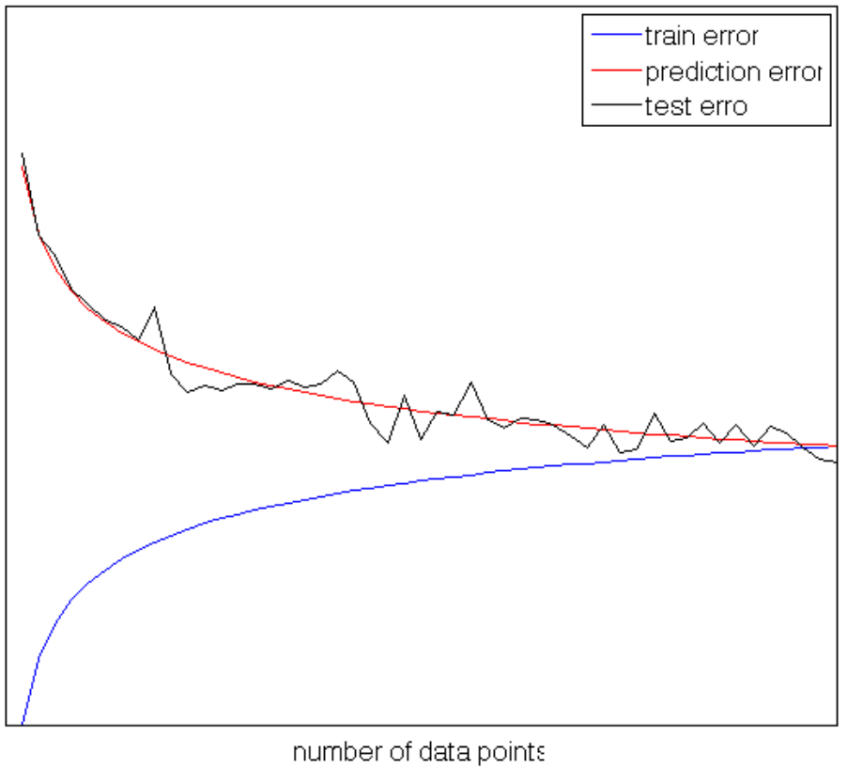


- Test set only unbiased if **no used for learning** (including parameter selection!)

Train-Test

- In practice
  - **Randomly** divide the data set into test and train
  - Use training data to **optimize parameters**
  - Use test data to **evaluate prediction error**

$$L_{test} = \frac{1}{N_{test}} \sum_{n=1}^{N_{test}} (t_n - y(\mathbf{x}_n))^2$$

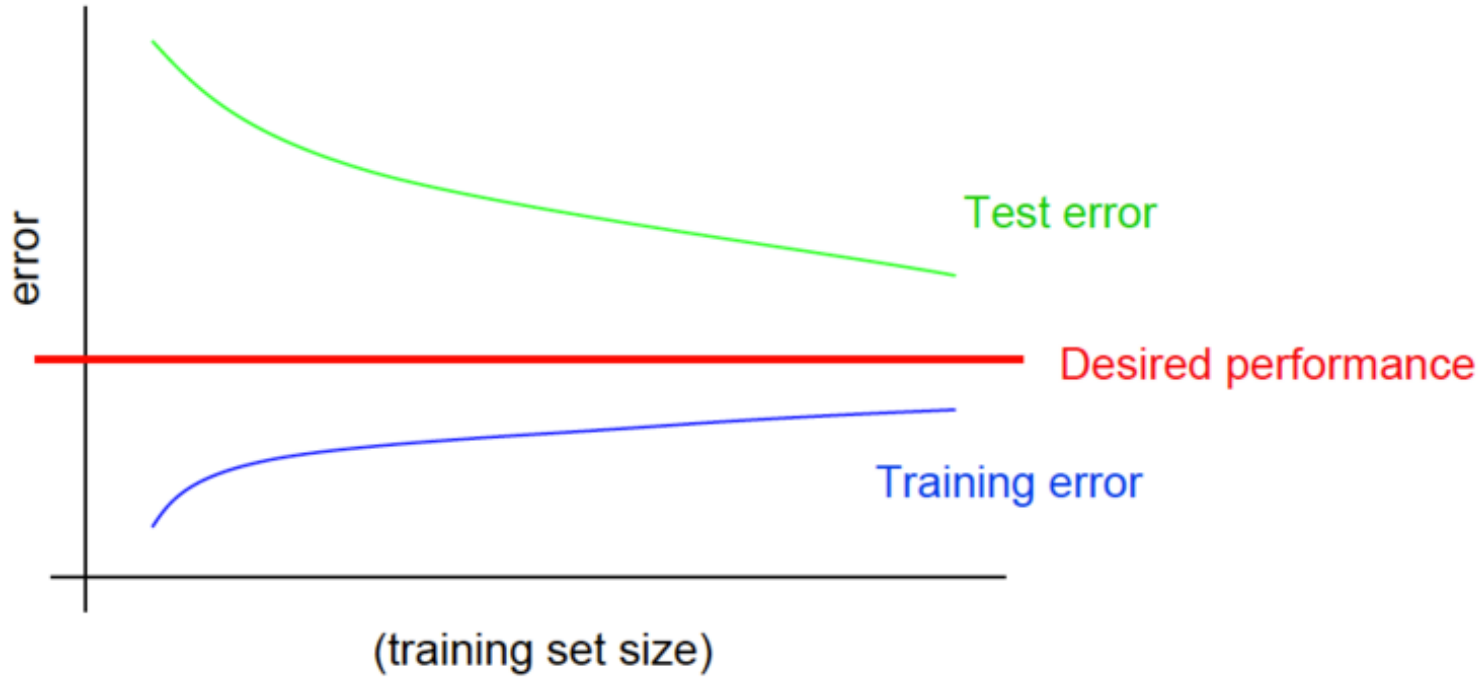


- Test set only unbiased if **no used for learning** (including parameter selection!)

Train-Test

Bias-Variance Tradeoff

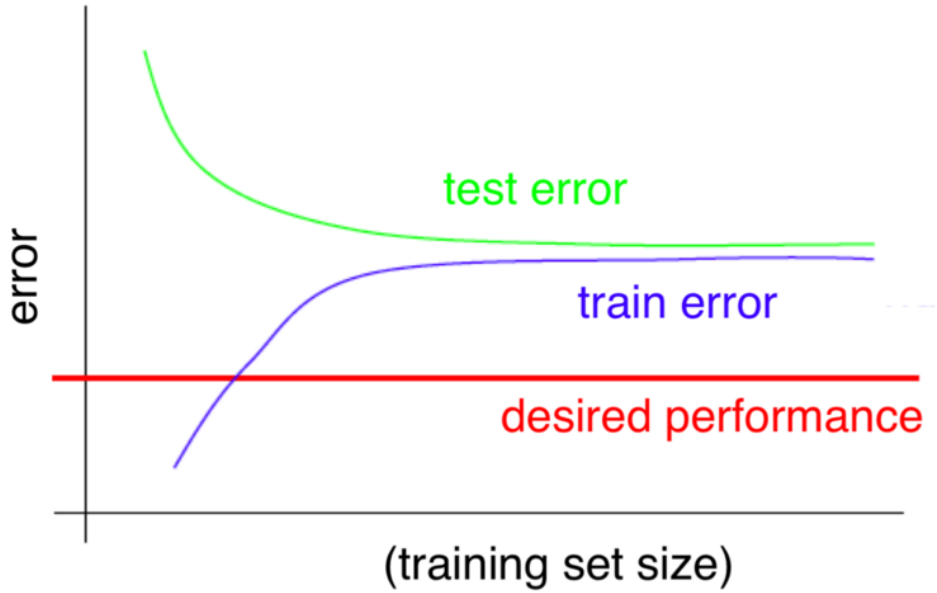
What is the problem? **High Variance**



Train-Test

Bias-Variance Tradeoff

What is the problem? **High Bias**



Managing the Bias-Variance Tradeoff

- The bias-variance tradeoff can be **managed** using different techniques
  - **Model selection**
    - Feature selection
    - Regularization
    - Dimension Reduction
  - **Model ensemble**
    - Bagging
    - Boosting

Curse of Dimensionality

- It is related to the **exponential increase in volume** associated with adding **extra dimensions** to the input space
- Working with high-dimensional data is **difficult**
  - large **variance** (overfitting)
  - needs **many samples**: N^d
  - high **computational cost**
- Common pitfall:
  - If we can't solve a problem with a few features, adding **more features** seems like a good idea
  - However the number of samples usually **stays the same**
  - The method with more features is likely to **perform worse** instead of expected better

Three classes of methods

Feature Selection : identifies a **subset of input features** that are most related to the output

Regularization : all the input features are used, but the estimated coefficients are **shrunken towards zero**, thus reducing the variance

Dimension Reduction : the input variables are **projected** into a lower-dimensional subspace

Best Subset Selection

1. Let $\mathcal{M}_0$ denote the **null model**, which contains no input feature (it simply predicts the sample mean for each observation)
2. For $k = 1, 2, \ldots, M$
   1. Fit all $\binom{M}{k}$ models that contain exactly k features
   2. Pick the best among these $\binom{M}{k}$ models and call it $\mathcal{M}_k$. Here **best** is defined as having the smallest RSS, or equivalently largest R^2
3. Select a **single best model** from among $\mathcal{M}_0, \ldots, \mathcal{M}_M$ using some criterion (cross-validation, AIC, BIC,...)

Feature Selection in Practice

- Best Subset Selection has problems when M is **large**
  - Computational cost
  - Over-fitting
- Three **metaheuristics**
  - **Filter**: rank the features (ignoring the classifier) and select the best ones
    - **Forward Step-wise Selection**: starts from an **empty model** and **adds** features **one-at-a-time**
    - **Backward Step-wise Selection**: starts with **all the features** and **removes** the least useful feature, **one-at-a-time**
  - **Embedded** (Built-in): the learning algorithm exploits its own variable selection technique
    - Lasso, Decision trees, Auto-encoding, etc.
  - **Wrapper**: evaluate only some subsets

Choosing the Optimal Model

- The model containing **all the features** will always have the **smallest training error**
- We wish to choose a model with **low test error**, not a low training error
- Therefore, RSS and R^2 are **not** suitable for selecting the best model among a collection of models with different numbers of features
- There are two approaches to **estimate the test error**
  - **Direct** estimation using a **validation** approach
  - Making an **adjustment** to the training error to account for **model complexity**

Direct estimation of Test Error

- Can we use the **train error**?
  - No
- Can we use the **test error**?
  - No, **never** use the test data for learning!

Validation Set

- So far, we have considered to **randomly** split the data set into **two** parts:
  - Training data
  - Test data
- But Test data must always remain **independent!**
  - **Never ever ever ever learn on test data**, including for model selection
- Given a dataset, **randomly** split into **three** parts
  - Training data
  - Validation data
  - Test data
- Use validation data for **tuning learning algorithm** (e.g., model selection)
  - Save test data for **very final evaluation**

Direct estimation of Test Error

- Can we use the **train error**?
  - No
- Can we use the **test error**?
  - No, never use the test data for learning!
- Can we use the **validation set error**?
  - No, **overfit** to validation set
- And so??? What is **the solution**?
  - **Cross validation!!!**

(LOO) Leave-One-Out Cross Validation

- Consider a **validation set with 1 example**:
  - Training data $\mathcal{D}$
  - $\mathcal{D} \setminus \{n\}$: training data with the n-th data point moved to validation set
- Learn model with $\mathcal{D} \setminus \{n\}$ data set
- Estimation error is based on the performance over **only one point**
- **LOO cross validation**: Average over all data points:

$$L_{LOO} = \frac{1}{N} \sum_{n=1}^{N} (t_n - y_{\mathcal{D}\setminus\{n\}}(\mathbf{x}_n))^2$$

- LOO is **almost unbiased**!
  - **slightly pessimistic**

Computational Cost of LOO

- Suppose you have 100,000 data points
- You implement a great version of your learning algorithm
  - Learns in only 1 second
- Computing LOO will take about 1 day!!!
  - If you have to do for each choice of basis functions, it will take **forever**!!!

Use k-fold Cross Validation

- **Randomly** divide training data into k equal parts: $\mathcal{D}_1, \ldots, \mathcal{D}_k$
- For each i
  - Learn model $y_{\mathcal{D}\setminus\mathcal{D}_i}$ using data points not in $\mathcal{D}_i$
  - Estimate error of $y_{\mathcal{D}\setminus\mathcal{D}_i}$ on validation set $\mathcal{D}_i$

$$L_{\mathcal{D}_i} = \frac{k}{N} \sum_{(\mathbf{x}_n, t_n) \in \mathcal{D}_i} (t_n - y_{\mathcal{D}\setminus\mathcal{D}_i}(\mathbf{x}_n))^2$$

  - k-fold cross validation error is average over data splits

$$L_{k-fold} = \frac{1}{k} \sum_{i=1}^{k} L_{\mathcal{D}_i}$$

- k-fold cross validation properties:
  - Much **faster** to compute than LOO
  - **More (pessimistically) biased**
  - Usually, $k = 10$;)

Adjustment Techniques

C_p : $C_p = \frac{1}{N}(RSS + 2d\tilde{\sigma}^2)$

where d is the total number of parameters, $\tilde{\sigma}^2$ is an estimate of the variance of the noise ϵ

AIC : $AIC = -2\log L + 2d$

where L is the maximized value of the likelihood function for the estimated model

BIC : $BIC = \frac{1}{N}(RSS + \log(N)d\tilde{\sigma}^2)$

BIC replaces the $2d\tilde{\sigma}^2$ of C_p with $\log(N)d\tilde{\sigma}^2$ term. Since $\log N > 2$ for any $n > 7$, BIC selects smaller models

Adjusted R^2 : $AdjustedR^2 = 1 - \frac{RSS/(N-d-1)}{TSS/(N-1)}$

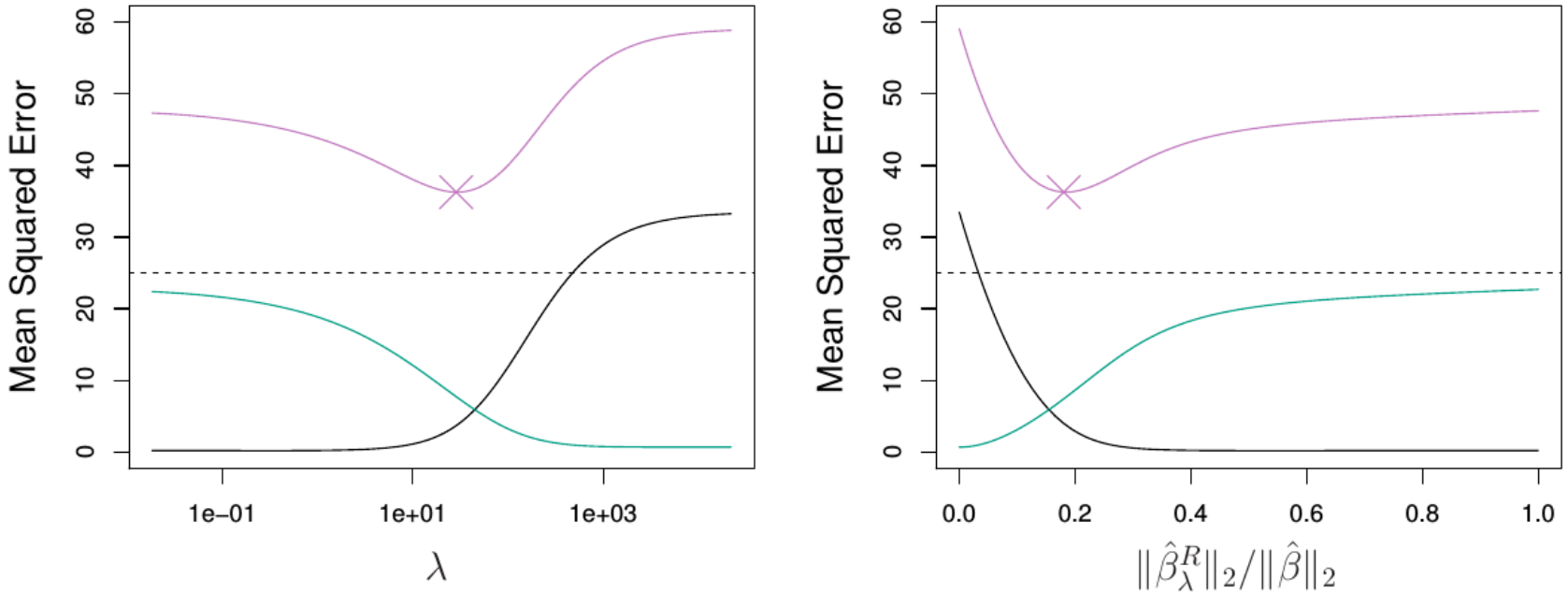
where TSS is the total sum of squares. Differently from the other criteria, here a **large value** indicates a model with a **small test error**.

Shrinkage Methods

- We have already seen regularization approaches applied to **linear models**:
  - Ridge regression
  - Lasso
- Such methods **shrink** the parameters towards **zero**
- It may not be immediately obvious why such a constraint should improve the fit, but it turns out that shrinking coefficient estimates can significantly **reduce the variance**

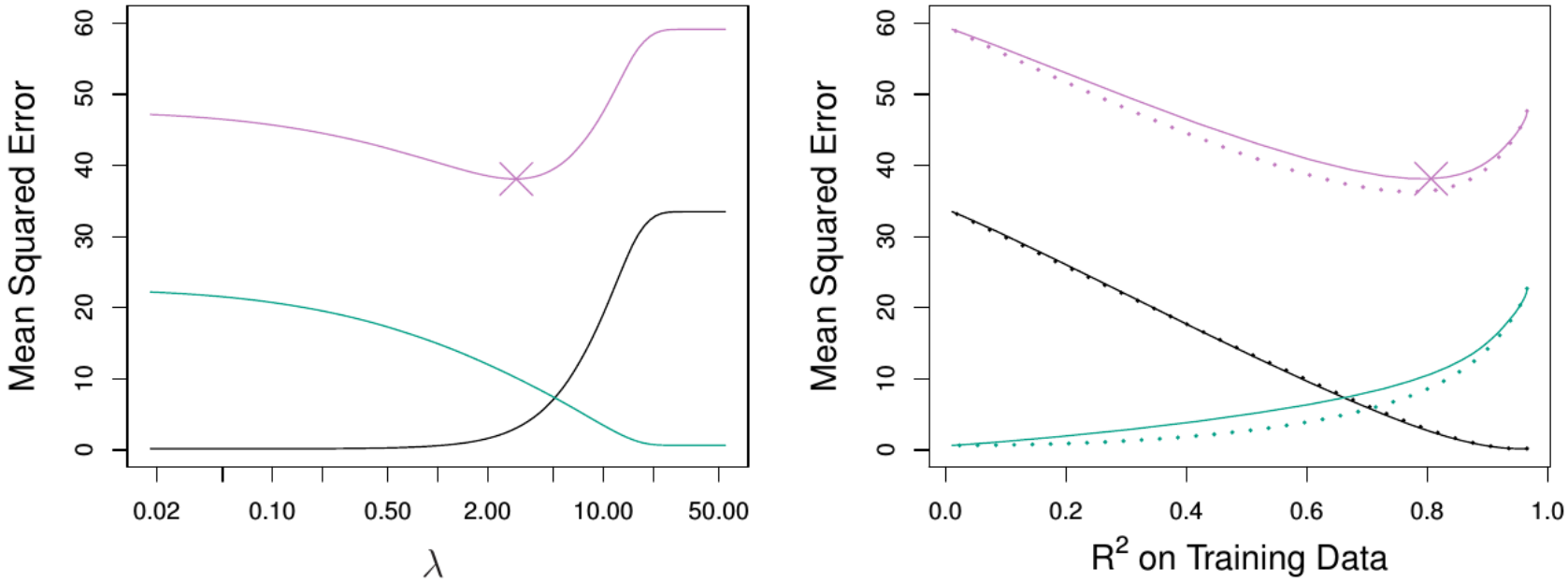
Why Does Ridge Regression Improve Over Least Squares?

Simulated data with $N = 50$ observations, $M = 45$ features, all having nonzero coefficients.



- Black: squared bias
- Green: variance
- Purple: MSE
- Dashed: the minimum possible MSE
- Purple crosses: ridge regression model with minimum MSE

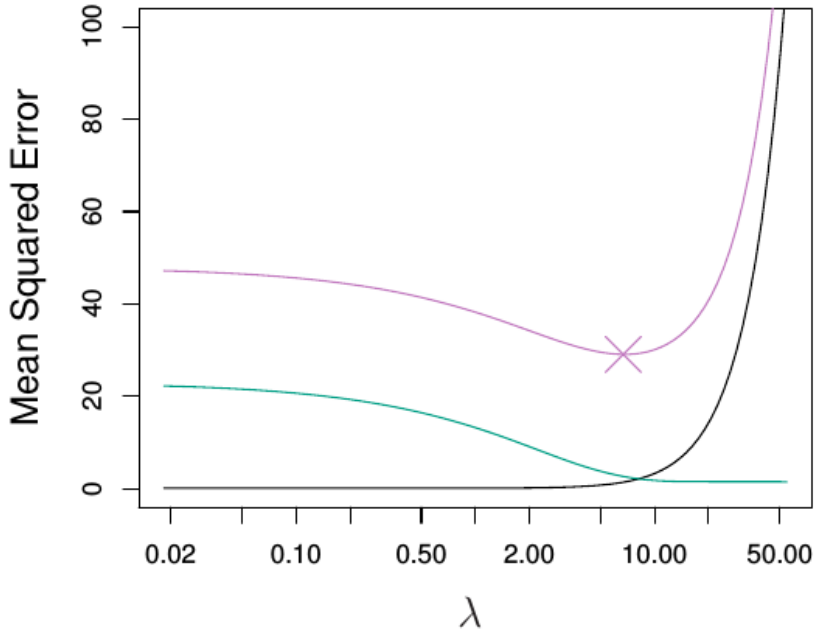
Comparing the Lasso and Ridge Regression

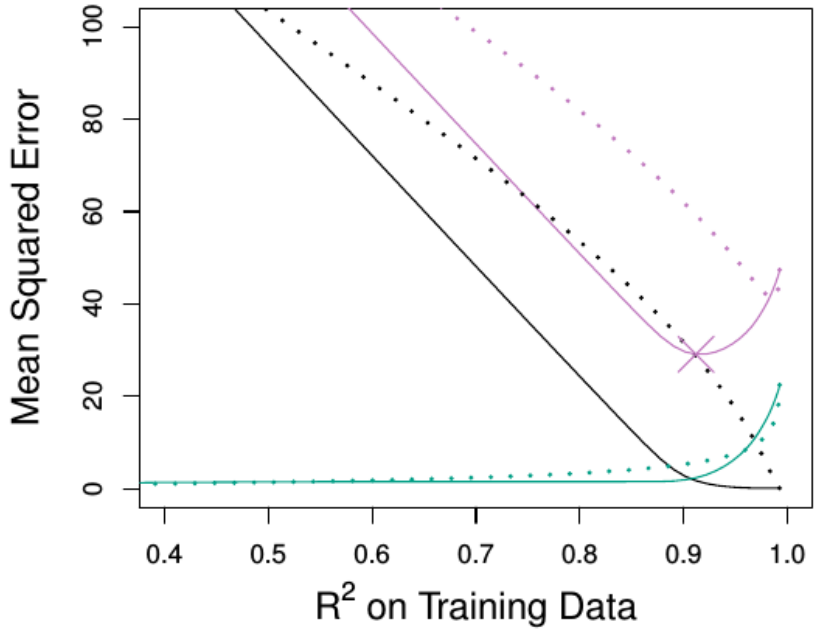


- On the left we have **Lasso**, on the right the comparison with **ridge**
- Black: squared bias
- Green: variance
- Purple: MSE
- Solid: Lasso
- Dashed: Ridge
- Purple crosses: Lasso model with minimum MSE

Comparing the Lasso and Ridge Regression

What happens if only 2 out of 45 features are relevant



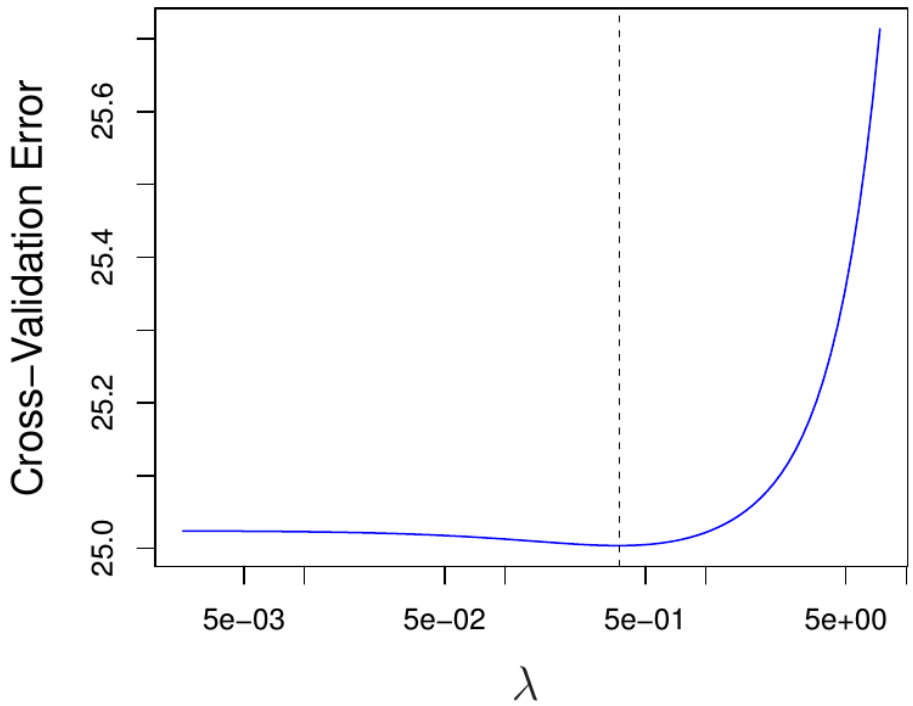


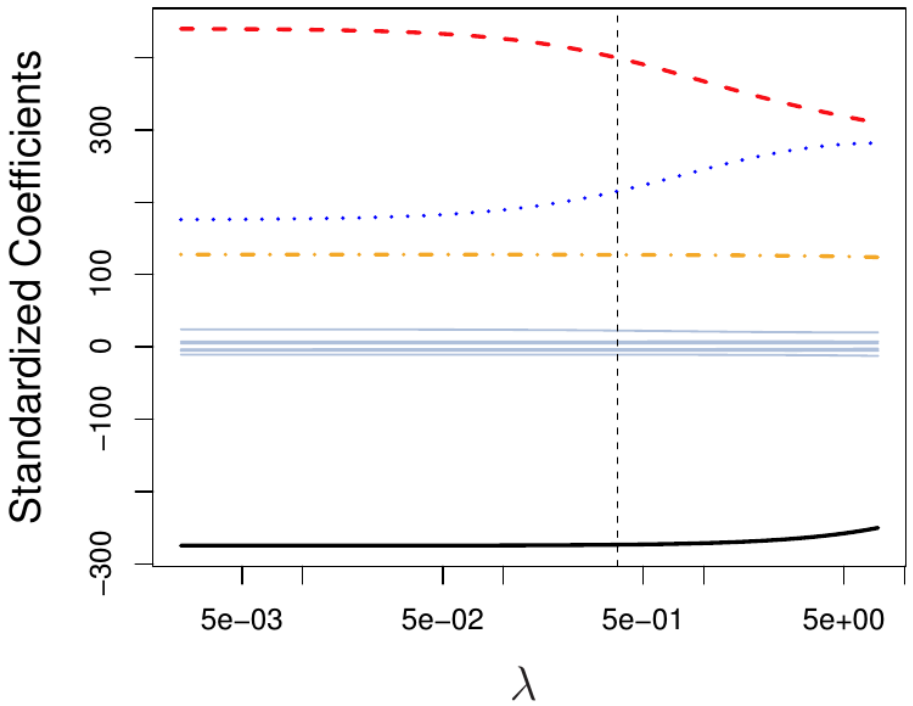
- Black: squared bias
- Green: variance
- Purple: MSE
- Solid: Lasso
- Dashed: Ridge
- Purple crosses: Lasso model with minimum MSE

Selecting the Tuning Parameter for Ridge and Lasso

- As for subset selection, for ridge regression and lasso we require a method to determine which of the models under consideration is **best**
- So, we need a method for selecting the tuning parameter λ
- **Cross-validation** provides a simple way to tackle this problem. We choose a grid of λ values, and compute the corss-validation error rate for each value of λ
- We then select the tuning parameter value for which the cross-validation error is **smallest**
- Finally, the model is **re-fit** using **all** of the available observations and the selected value of the tuning parameter

Cross-Validation with Ridge Regression



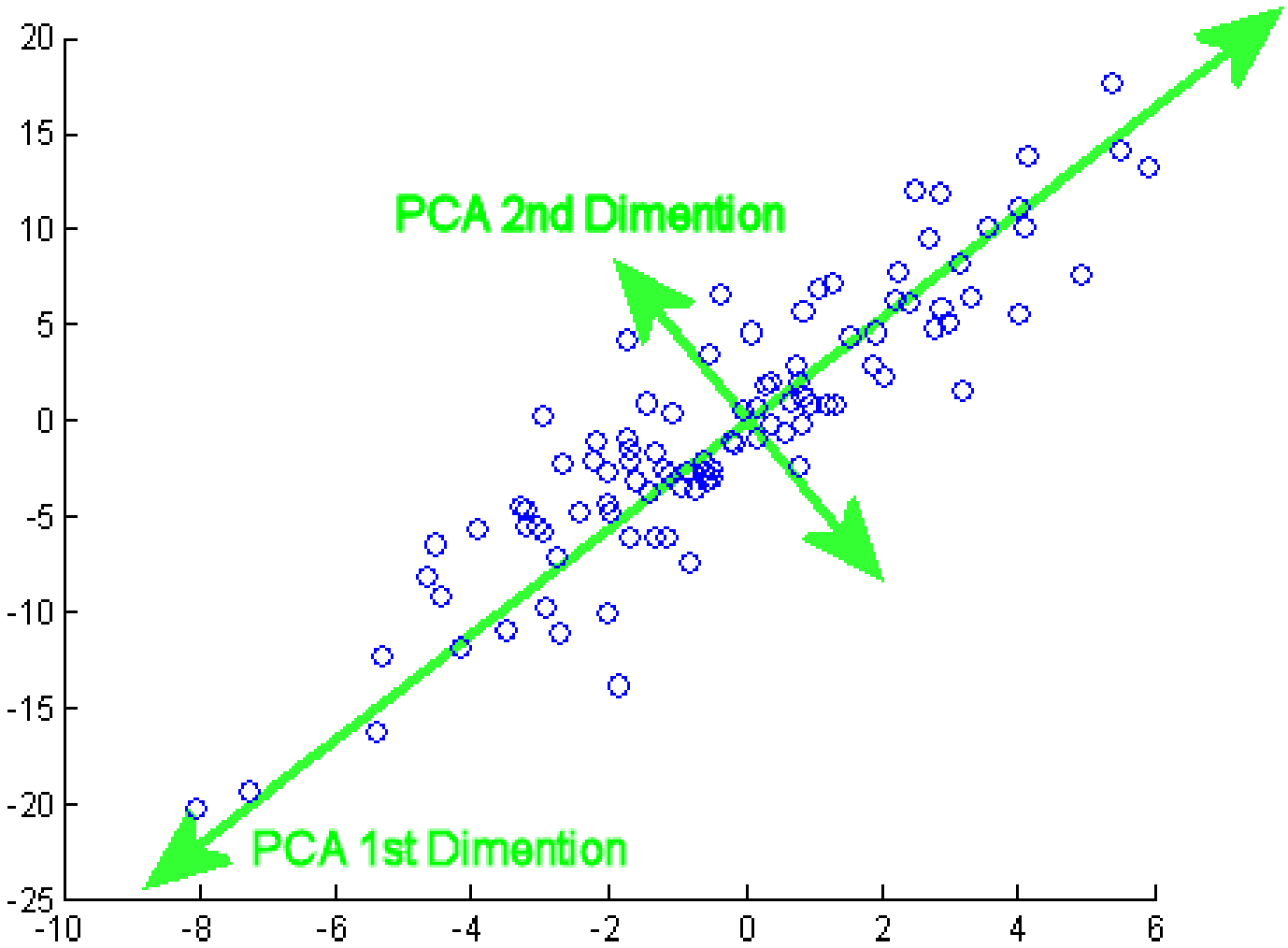


Dimension Reduction

- The previous approaches operate on the **original features**
- Dimension reduction methods **transform** the original features and then the model is learned on the **transformed variables**
- Dimension reduction is an **unsupervised learning** approach
- There are many techniques to perform dimensionality reduction:
  - **Principal Component Analysis (PCA)**
  - Independent Component Analysis (ICA)
  - Self-organizing Maps
  - Autoencoders
  - etc.

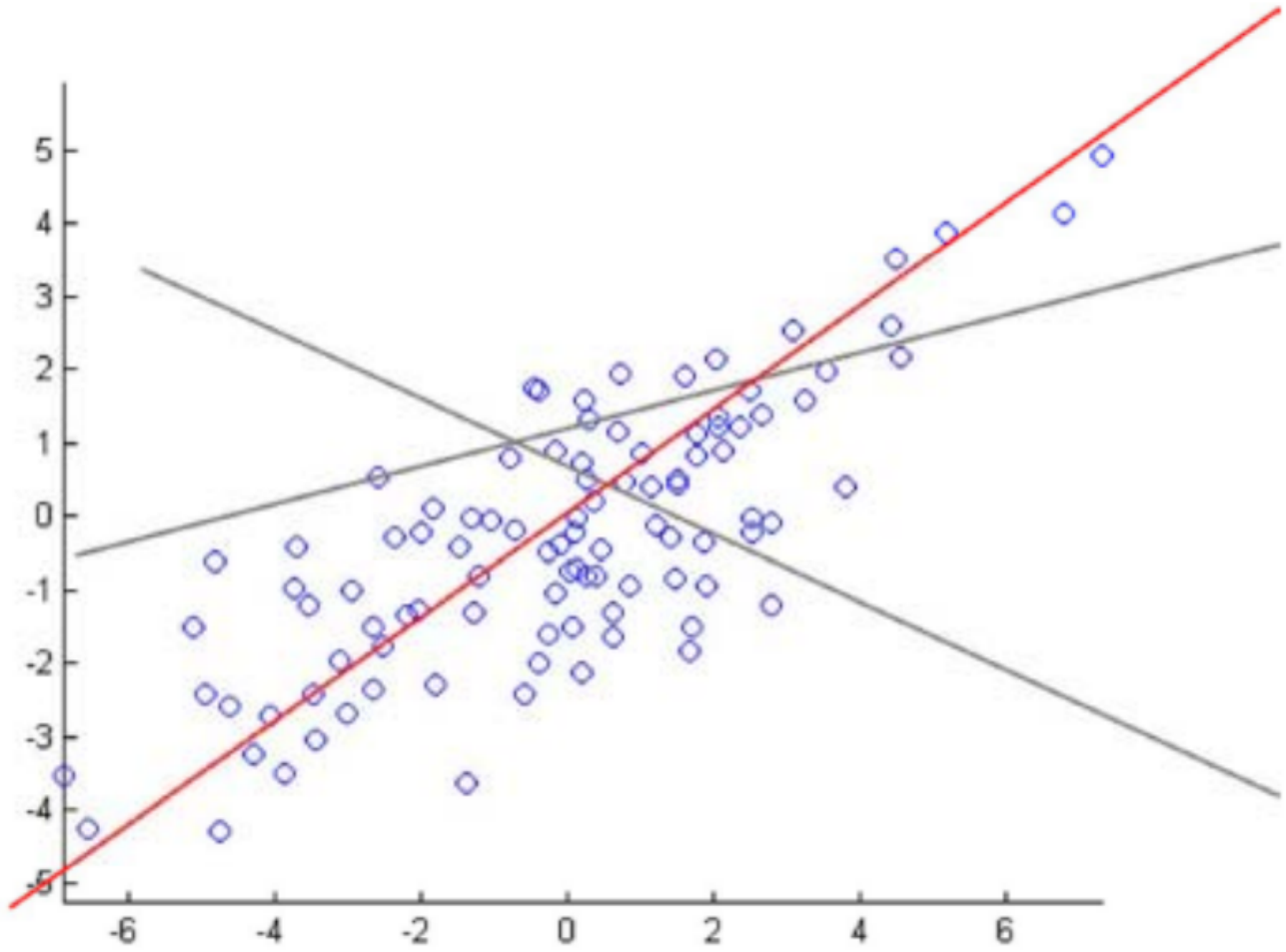
Principal Component Analysis (PCA)

The idea is to **project** onto the (input) subspace which accounts for most of the **variance**



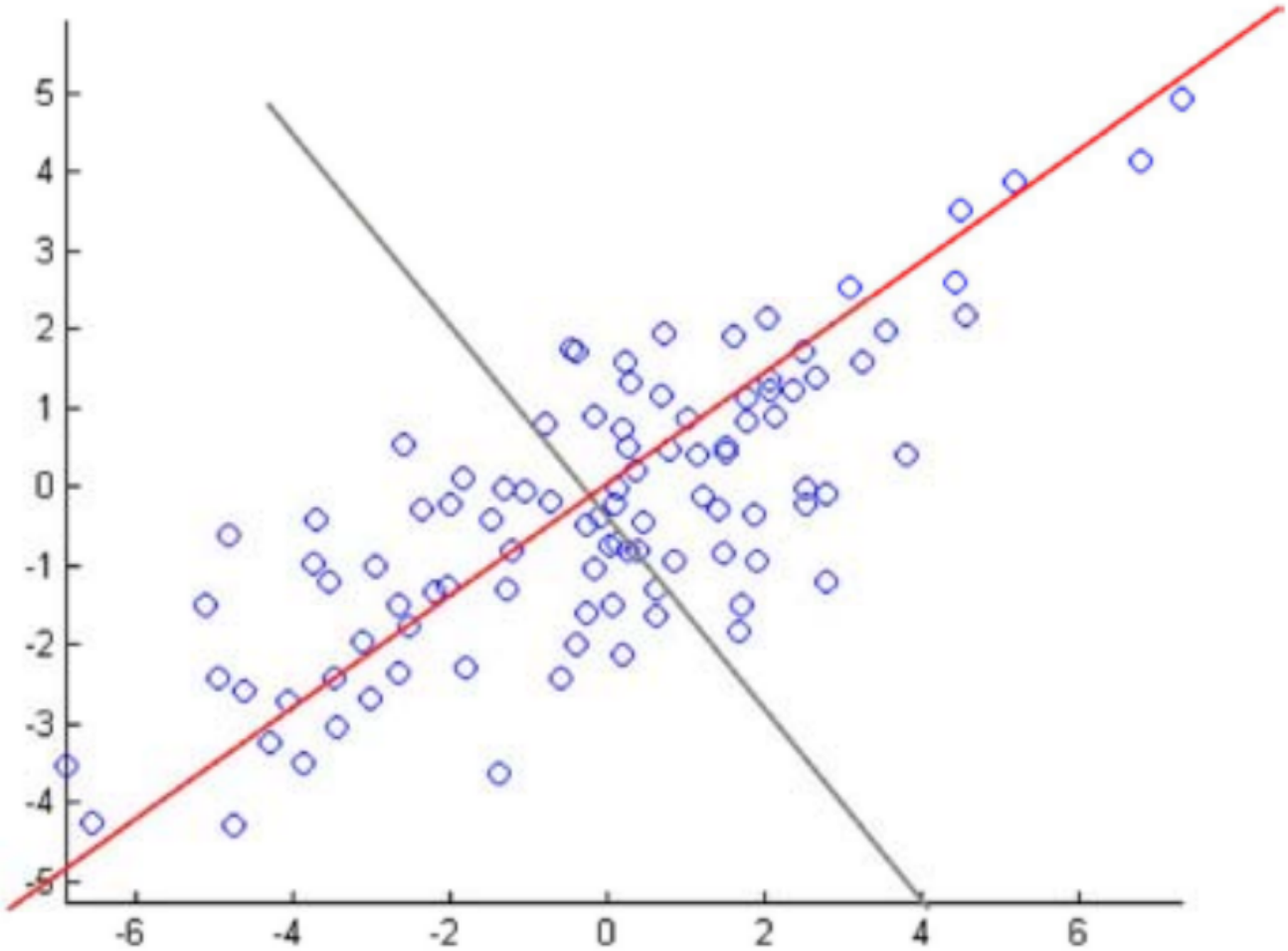
Conceptual Algorithm

Find a line such that when the data is projected onto that line, it has the **maximum variance**



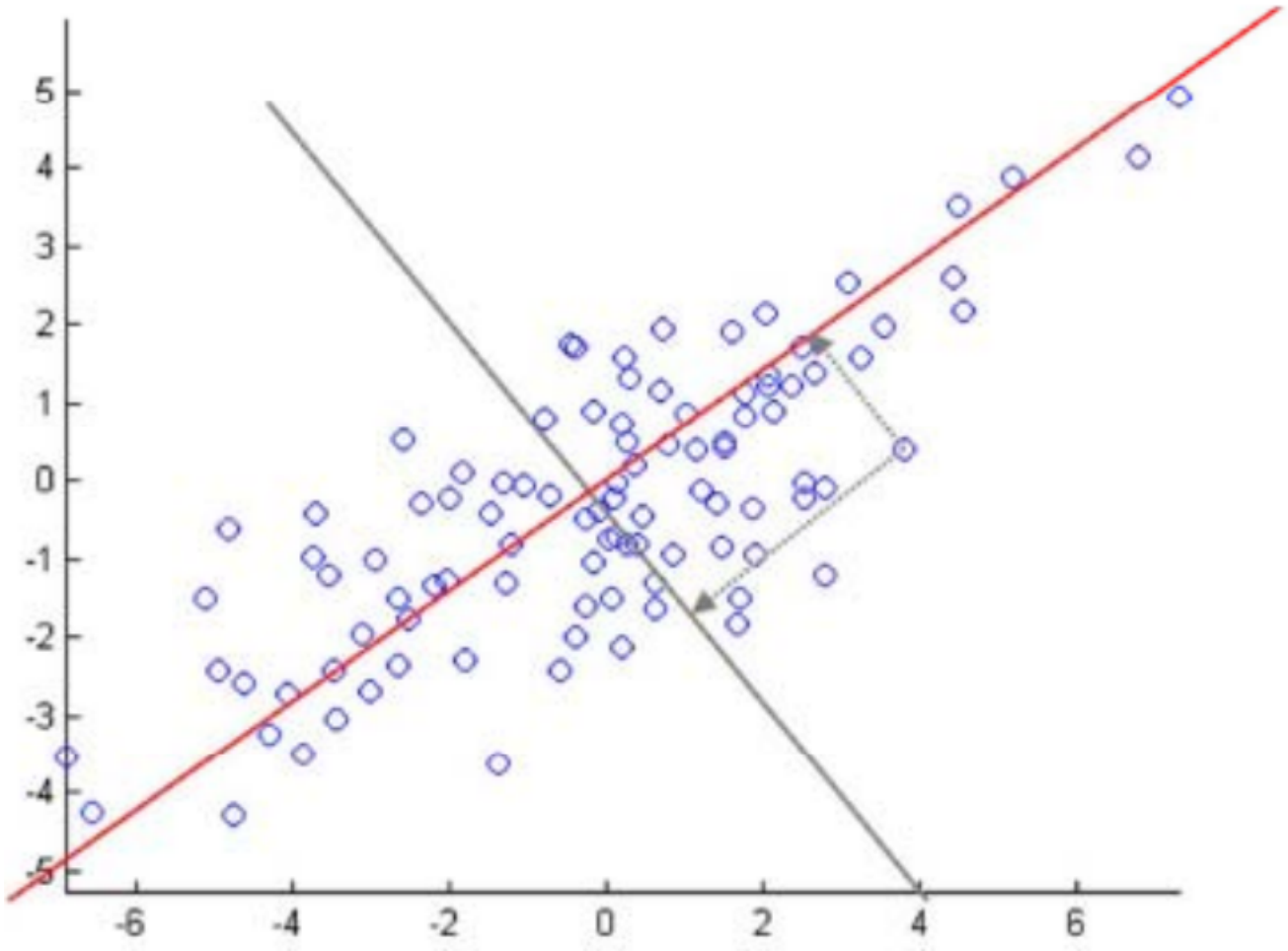
Conceptual Algorithm

Find a new line, **orthogonal** to the first one, that has maximum projected variance



Conceptual Algorithm

Repeat until m lines have been identified and project the points in the data set on these lines



Steps in PCA

- Mean center the data

$$\overline{\mathbf{x}} = \frac{1}{N}\sum_{n=1}^{N}\mathbf{x}_n$$

- Compute the covariance matrix $\mathbf{S}$
- Calculate eigenvalues and eigenvectors of $\mathbf{S}$

$$\mathbf{S} = \frac{1}{N-1}\sum_{n=1}^{N}(\mathbf{x}_n - \overline{\mathbf{x}})(\mathbf{x}_n - \overline{\mathbf{x}})^T$$

  - Eigenvector $\mathbf{e}_1$ with largest eigenvalue λ_1 is **first** principal component (PC)
  - Eigenvector $\mathbf{e}_k$ with k^{th} largest eigenvalues λ_k is k^{th} PC
  - $\lambda_k / \sum_i \lambda_i$ is the proportion of **variance** captured by k^{th} PC

Applying PCA

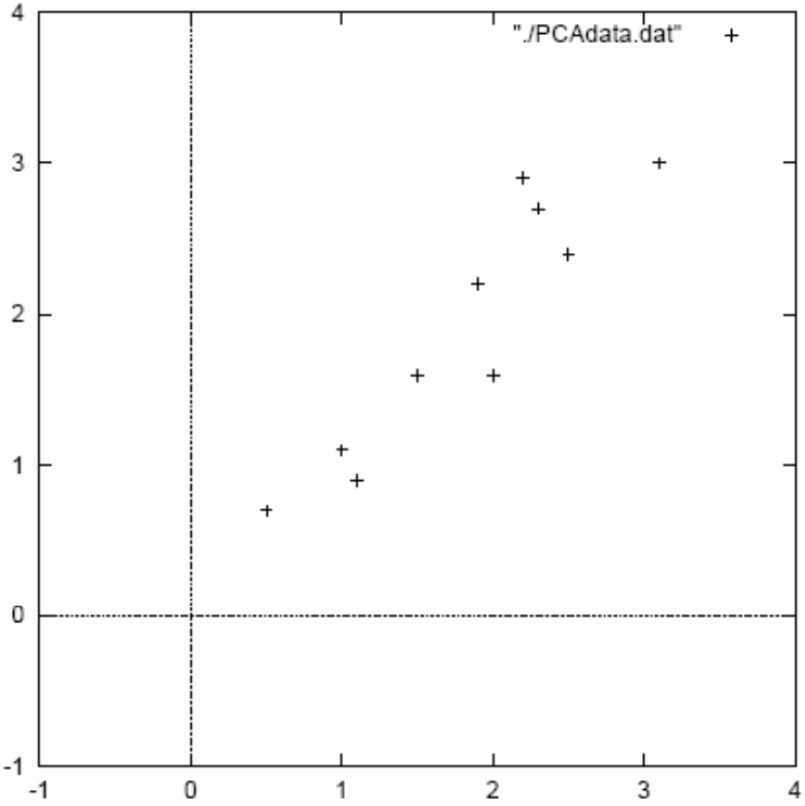
- Full set of PCs comprise a **new orthogonal basis** for feature space, whose axes are aligned with the maximum variances of original data
- Projection of original data onto first k PCs ($\mathbf{E}_k = (\mathbf{e}_1, \ldots, \mathbf{e}_k)$) gives a **reduced dimensionality representation** of the data

$$\mathbf{X}' = \mathbf{X}\mathbf{E}_k$$

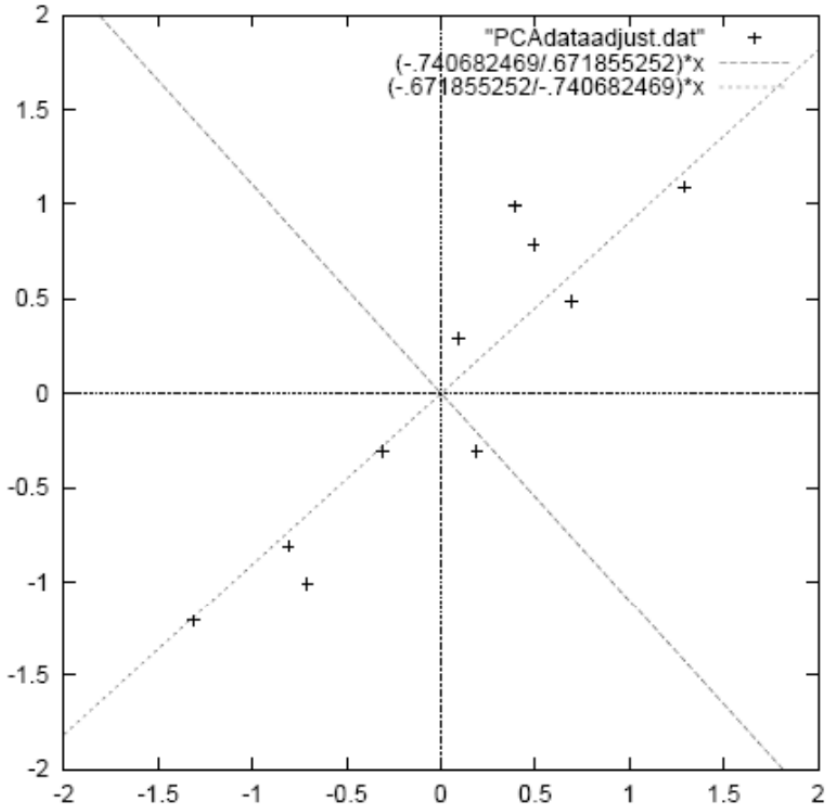
- Transforming reduced dimensionality projection back into original space gives a reduced dimensionality **reconstruction** of the original data
- Reconstruction will have **some error**, but it can be small and often is acceptable given the other benefits of dimensionality reduction

PCA Example

original data

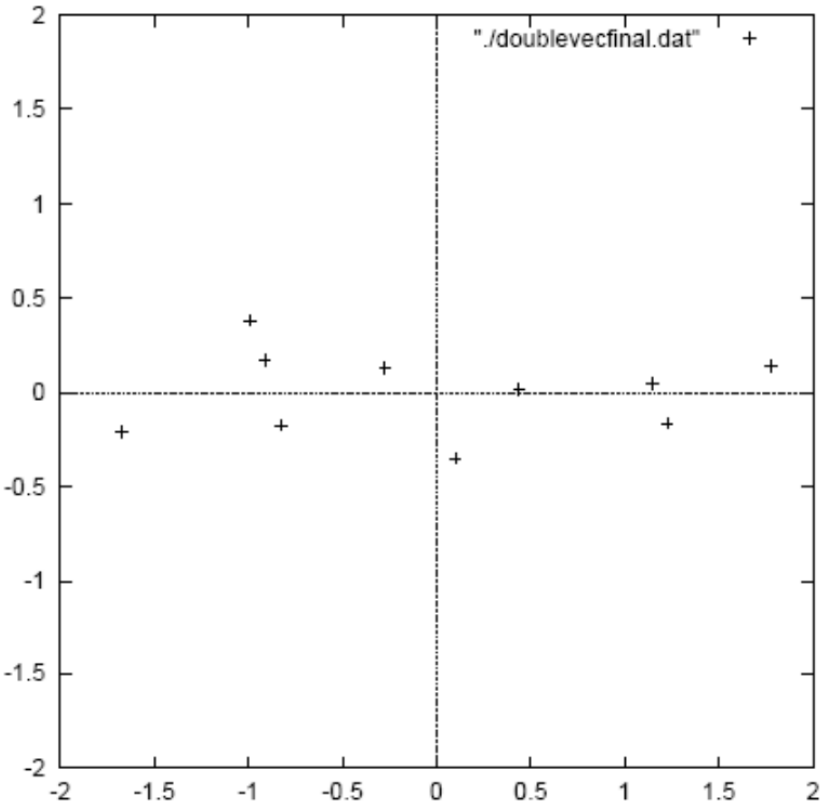


mean centered data with PCs overlayed

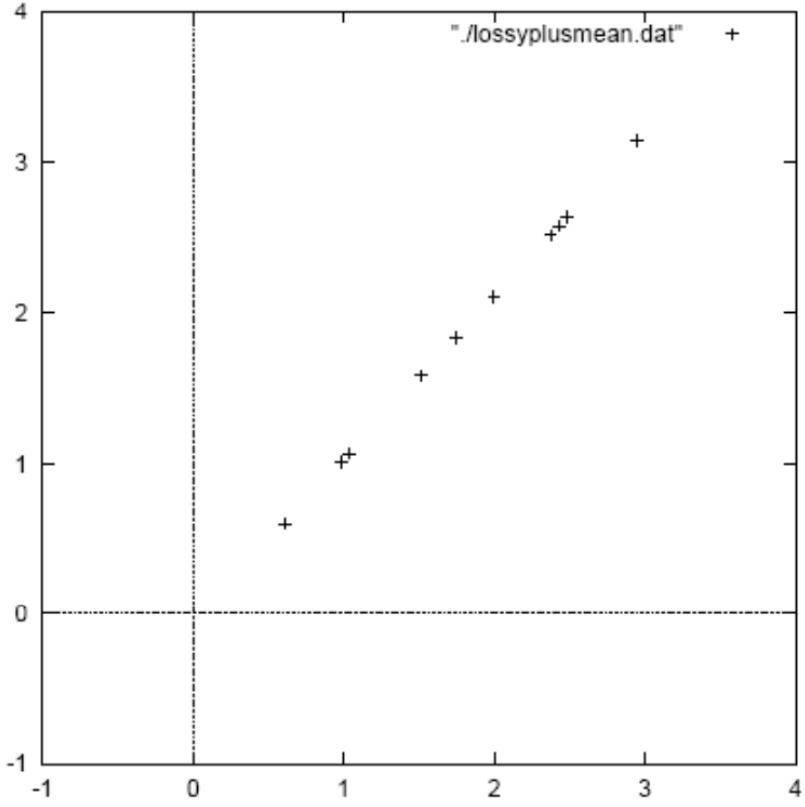


PCA Example

original data projected Into full PC space



original data reconstructed using only a single PC

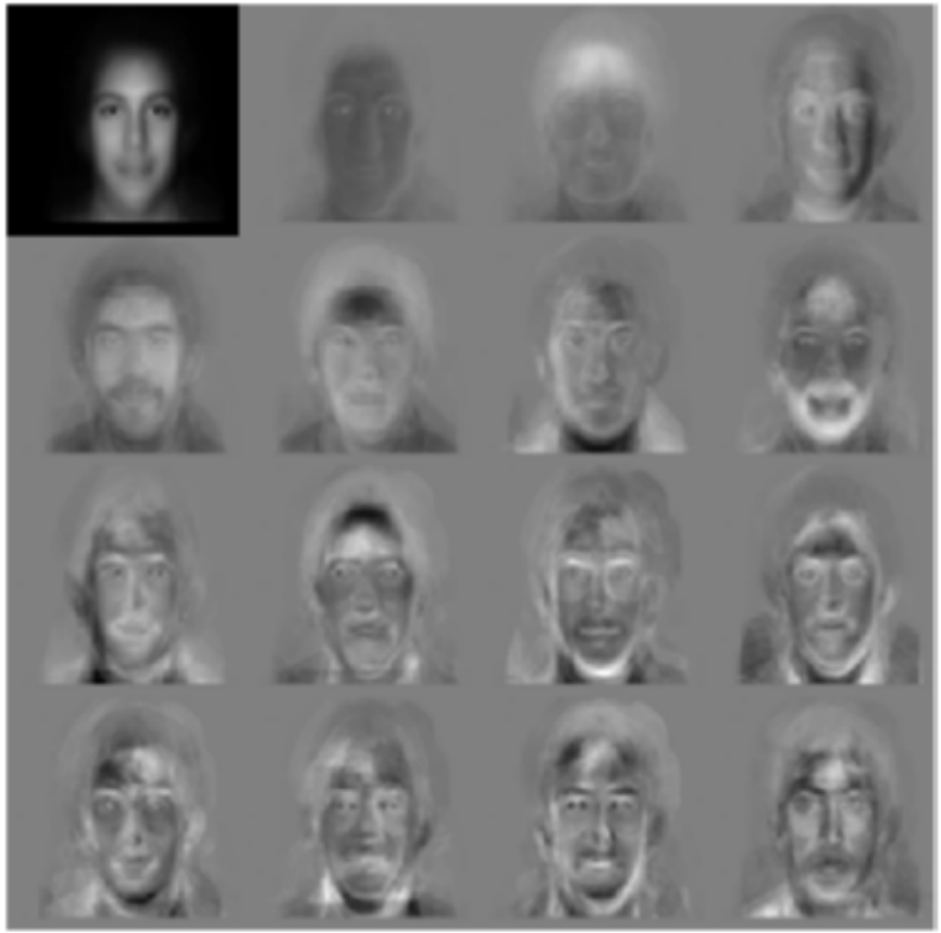


Example: Face Recognition

- A typical image of size 256×128 is described by $n = 256 \times 128 = 32768$ dimensions
- Each face image lies somewhere in this **high dimensional space**
- Images of faces are generally similar in overall configuration
  - They **cannot be randomly distributed** in this space
  - We should be able to describe them in a **much low-dimensional space**

PCA for Face Images: Eigenfaces

- Database of 128 carefully-aligned faces
- Here are the mean and the first 15 eigenvectors
- Each eigenvector can be shown as an image
- These images are face-like, thus called eigenface



PCA: a Useful Preprocessing Step

- Help reduce **computational complexity**
- Can help **supervised learning**
  - Reduced dimension $\Rightarrow$ simpler hypothesis space
  - Less risk of overfitting
- PCA can also be seen as **noise reduction**
- **Caveats**:
  - Fails when data consists of **multiple clusters**
  - Directions of greatest variance may **not** be most informative
  - Computational problems with many dimensions (SVD can help!)
  - PCA computes **linear** combination of features, but data often lies on a **nonlinear** manifold (see ISOMAP method)

Improving the Bias-Variance Tradeoff

- The methods seen so far can reduce bias by increasing variance or *vice versa*
- Is it possible to reduce variance **without increasing bias**?
  - Yes, **Bagging**!
- Is it possible to reduce also the bias?
  - Yes, **Boosting**!
- Bagging and Boosting are **meta-algorithms** (there are many other methods)
- The basic **idea** is: instead of learning one model, learn **several** and **combine** them
- Typically **improves accuracy**, often by a lot

Reducing Variance Without Increasing Bias

- **Averaging** reduces variance

$$Var(\overline{X}) = \frac{Var(X)}{N}$$

- Average models to **reduce variance**
- **One problem**
  - we have **only one** train set
  - where do **multiple** models come from?

Bagging: Bootstrap Aggregation

- Generate B **bootstrap samples** of the training data: **random sampling with replacement**
- **Train** a classifier or a regression function using each bootstrap sample
- **Prediction**
  - For classification: **majority vote** on the classification results
  - For regression: **average** on the predicted values
- **Reduce variation**
- Improves performance for **unstable learners** which vary significantly with small changes in the data set
- Works particularly well with **decision trees**

Analysis of Bagging

- The MSE is the sum of noise, squared bias and variance
- Bagging decreases **variance** due to **averaging**
- Bagging typically **helps**
  - when applied to an **overfitted** base model
  - **high dependency** on the training data
- It **does not help** much
  - When there is **high bias** (model robust to change in the training data)

Boosting

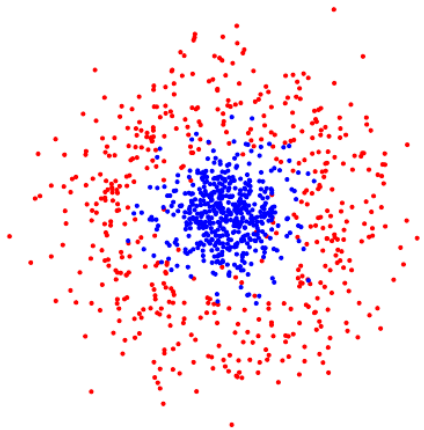
- Another (very different) technique of generating an **ensemble of models**
- The idea is to **sequentially** train **weak learners**
- A weak learner has a performance that on **any** train set is slightly better than chance prediction
- Intended to answer a **theoretical question**, not as a practical way of improve learning
  - Is a weak learner (low accuracy) **equivalent** to a strong learner (very high accuracy)?
  - The answer is **yes**! (if the weak learner is better than chance for **any** distribution)
  - How is it possible? **Boosting**!
  - **Many** weak classifiers turn into a strong classifier!

Boosting Algorithm

- **Weight** all train samples **equally**
- **Train** model on train set
- **Compute error** of model on train set
- **Increase weights** on train cases **where model gets wrong**
- **Train** new model on **re-weighted train set**
- **Re-compute errors** on **weighted** train set
- Increase weights again on cases model gets wrong
- Repeat until tired
- Final model: **weighted** prediction of each model

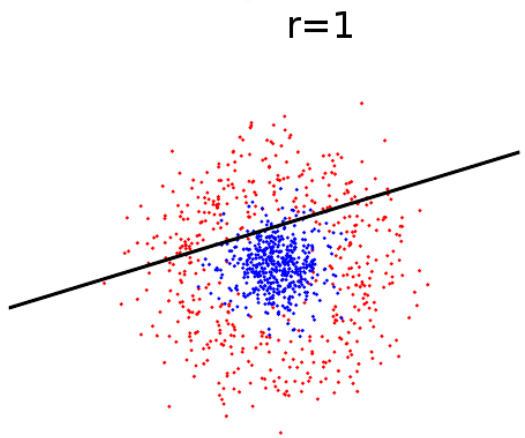
Boosting for Classification: AdaBoost

Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j)\mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i)\beta_r^{1-\mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r}\right) \mathbf{1}[y_r(x) = t]$



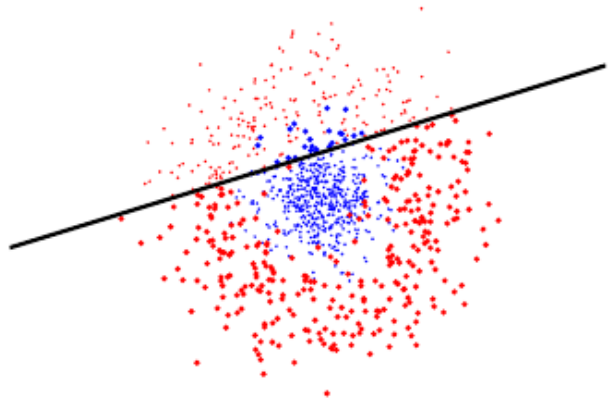
Boosting for Classification: AdaBoost

Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}$, weights$)$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$



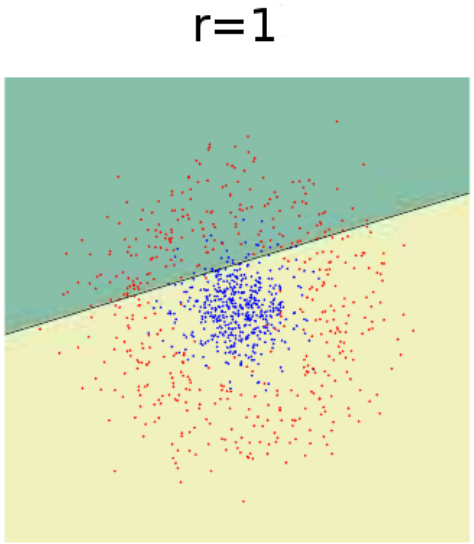
Boosting for Classification: AdaBoost

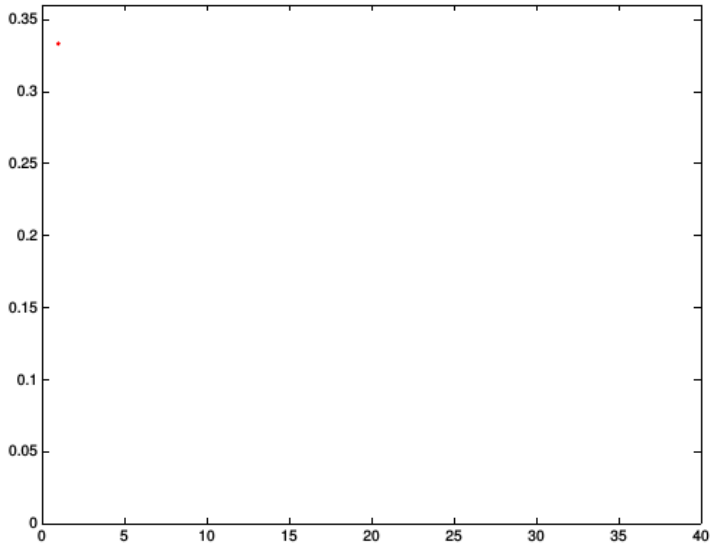
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$



Boosting for Classification: AdaBoost

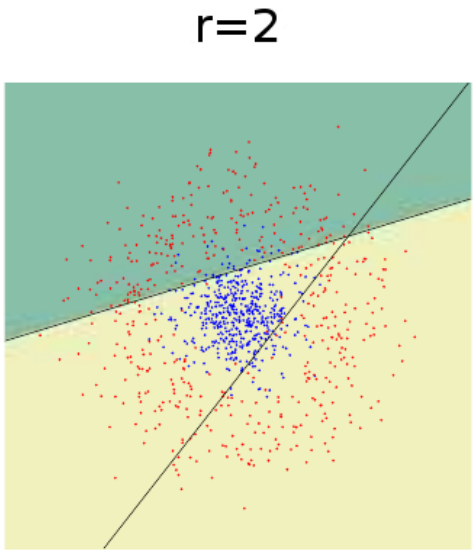
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$

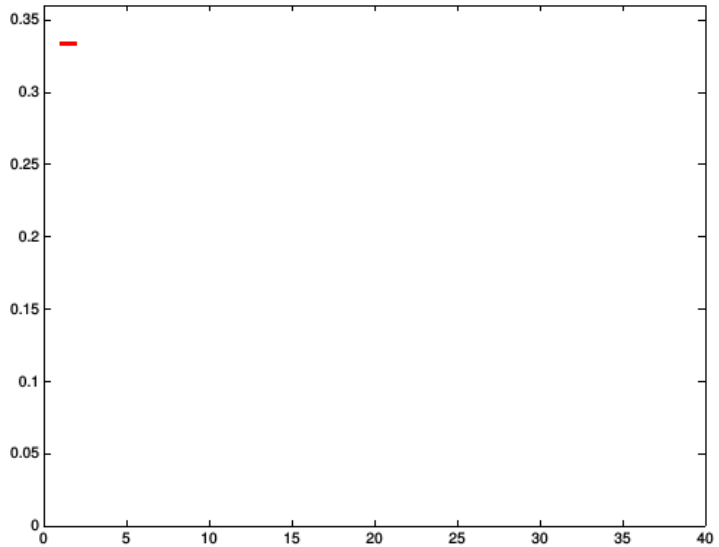




Boosting for Classification: AdaBoost

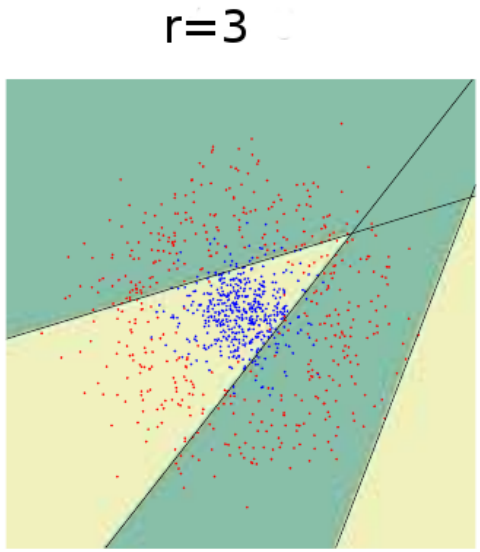
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j)\mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i)\beta_r^{1-\mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r}\right) \mathbf{1}[y_r(x) = t]$

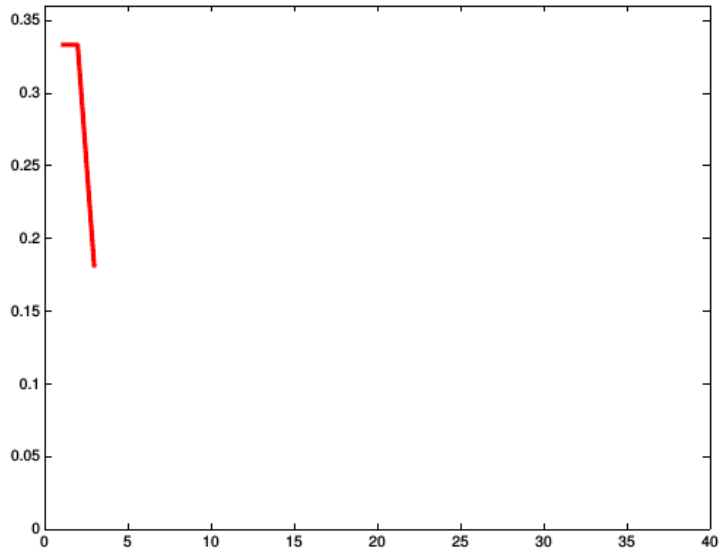




Boosting for Classification: AdaBoost

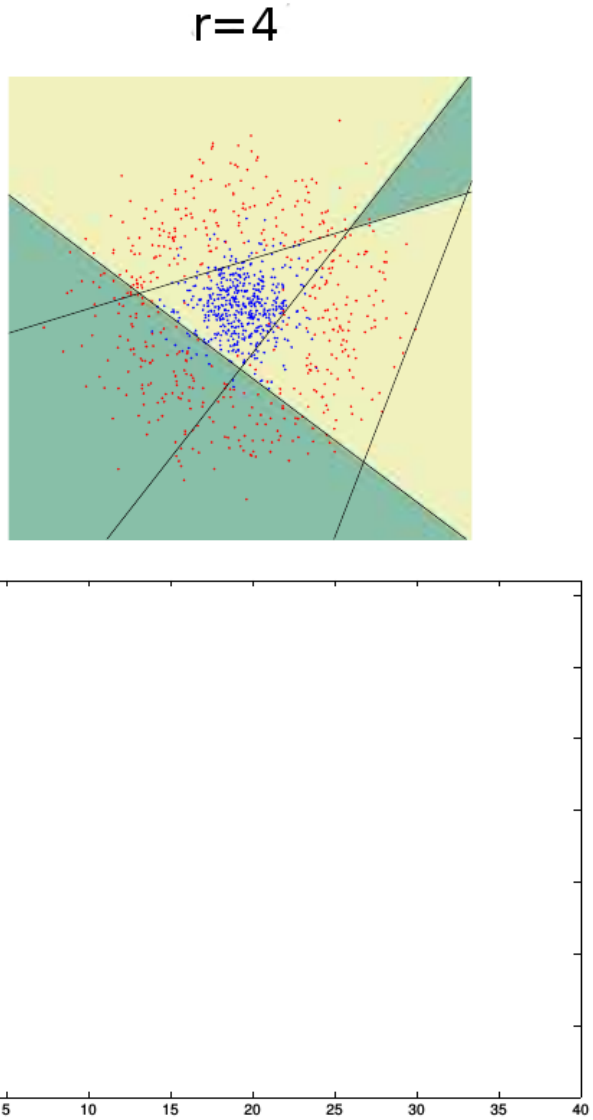
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$





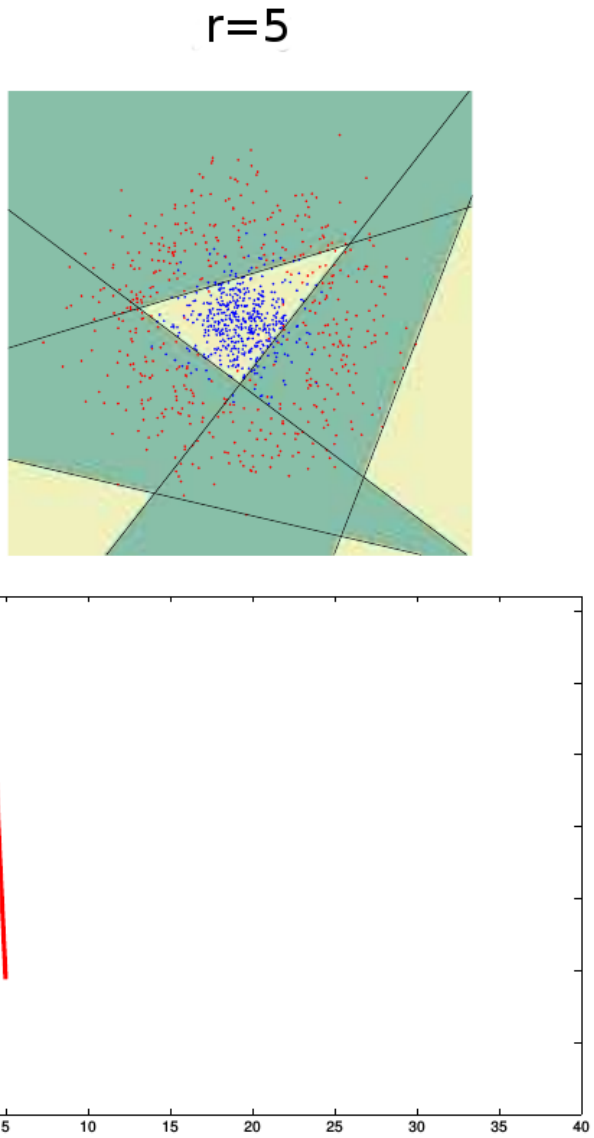
Boosting for Classification: AdaBoost

Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$



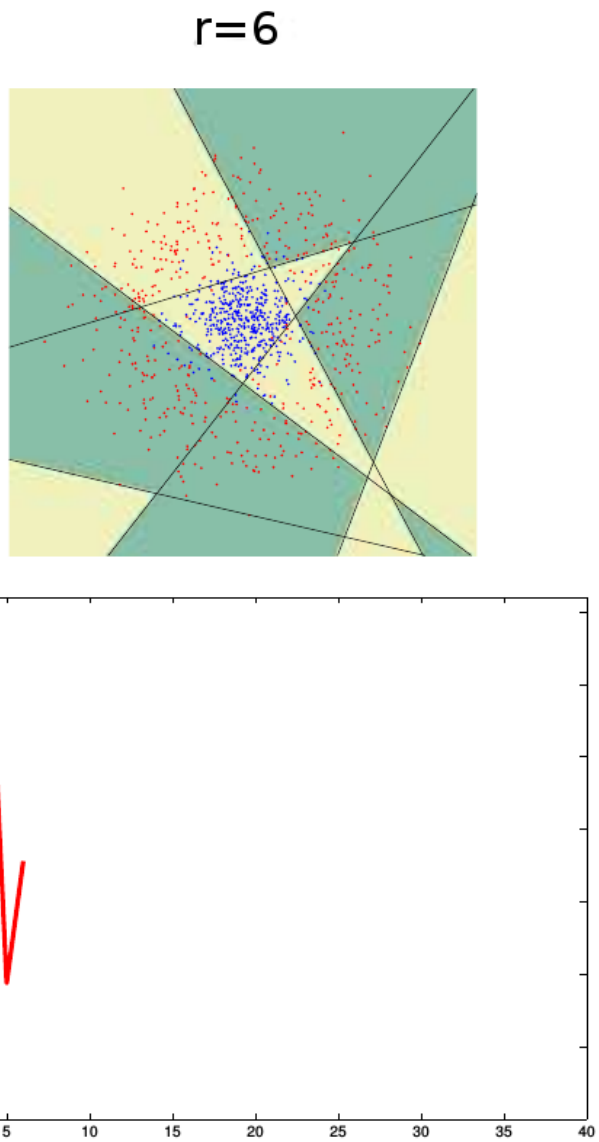
Boosting for Classification: AdaBoost

Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j)\mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i)\beta_r^{1-\mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r}\right) \mathbf{1}[y_r(x) = t]$



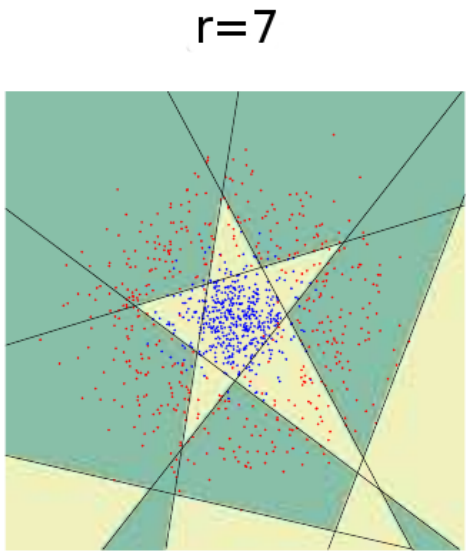
Boosting for Classification: AdaBoost

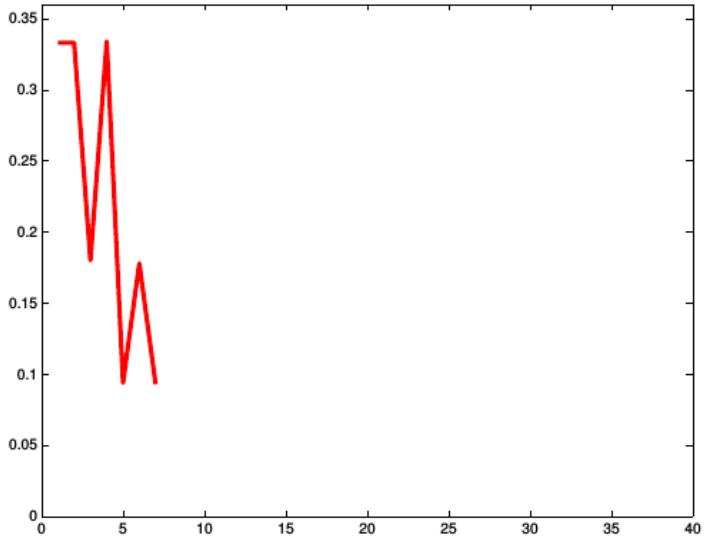
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$



Boosting for Classification: AdaBoost

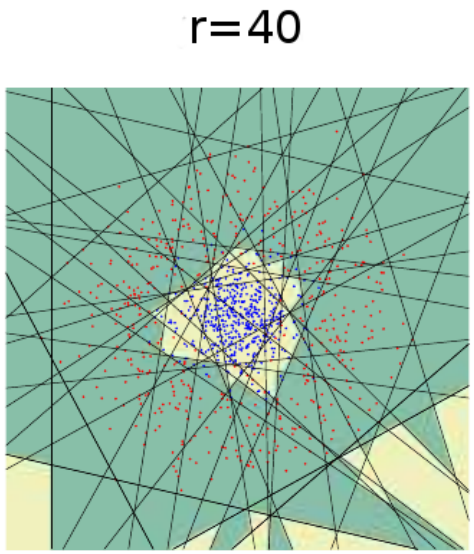
Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j) \mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i) \beta_r^{1 - \mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r} \right) \mathbf{1}[y_r(x) = t]$

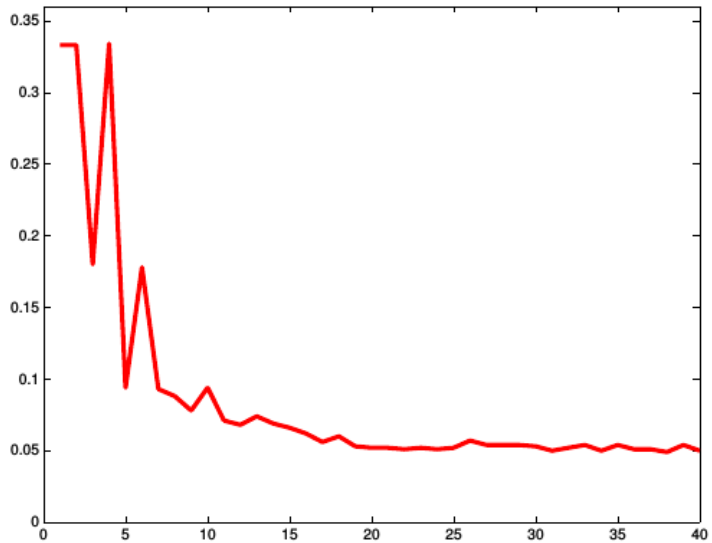




Boosting for Classification: AdaBoost

Require: Training set: $\mathcal{D} = \{(x_1, t_1), \ldots, (x_m, t_m)\}$
Require: Learner: $L(\mathcal{D}, \text{weights})$
Require: Number of rounds: T
  for all i in $\mathcal{D}$ **do**
    $w_1(i) = \frac{1}{N}$
  end for
  for $r = 1$ to T **do**
    for all i **do**
      $p_r(i) = \frac{w_r(i)}{\sum_j w_r(j)}$
    end for
    $y_r = L(\mathcal{D}, p_r)$
    $\epsilon_r = \sum_j p_r(j)\mathbf{1}[y_r(j) \neq t_j]$
    if $\epsilon_r > 0.5$ **then**
      $T = r - 1$
      Exit
    end if
    $\beta_r = \frac{\epsilon_r}{1 - \epsilon_r}$
    for all i **do**
      $w_{r+1}(i) = w_r(i)\beta_r^{1-\mathbf{1}[y_r(x_i) \neq t_i]}$
    end for
  end for
  return $y(x) = arg\max_t \sum_{r=1}^{T} \left(\log \frac{1}{\beta_r}\right) \mathbf{1}[y_r(x) = t]$





Bagging vs Boosting

- Bagging reduces **variance**
- Boosting reduces **bias**
- Bagging doesn't work so well with **stable models**. Boosting might still help
- Boosting might hurt performance on **noisy datasets**. Bagging doesn't have this problem
- In practice bagging almost always helps
- On average, boosting helps more than bagging, but it is also more common for boosting to hurt performance
- The weights grow **exponentially**
- Bagging is easier to **parallelize**