

Parameter estimation Optimization

Lecture 2d

Parametric functions

- General ML model $Y \sim \text{Distribution}(f_{\theta}(\mathbf{x}), \dots)$
 - Some of them: $Y \sim f_{\theta}(\mathbf{x}) + \epsilon, \epsilon \sim \text{Distribution}(\dots)$
 - Generalization of simple models can be done
- **Example:** logistic $Y \sim \text{Bernoulli}\left(\frac{1}{1+e^{-\theta^T \mathbf{x}}}\right)$
 - Generalization 1: (**basis function expansion**):
 $Y \sim \text{Bernoulli}\left(\frac{1}{1+e^{-\theta^T \phi(\mathbf{x})}}\right)$
 - Generalization 2: $Y \sim \text{Bernoulli}\left(\frac{1}{1+e^{-f_{\theta}(\mathbf{x})}}\right)$
 - $f_{\theta}(\mathbf{x}) = \|\mathbf{x} - \boldsymbol{\theta}\|^2$

Cost minimization

- Given training set T , we **want** to minimize

$$E_{new} = \int_{(\mathbf{x}_*, y_*)} E(y_*, \hat{y}(\mathbf{x}_*, T, \boldsymbol{\theta})) p(\mathbf{x}_*, y_*) d\mathbf{x}_* dy_*$$

- We **can** minimize cost function

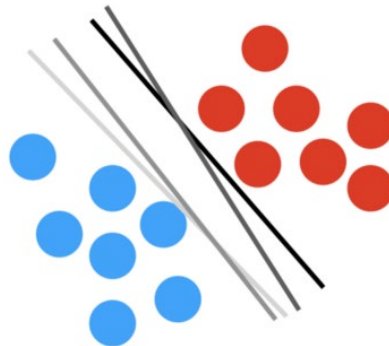
$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n L(y_i, \hat{y}(\mathbf{x}_i, \boldsymbol{\theta}))$$

$$E_{new} \approx J(\boldsymbol{\theta})?$$

- Optimizing $J(\boldsymbol{\theta})$ does not lead to optimizing E_{new}
 - Overfitting

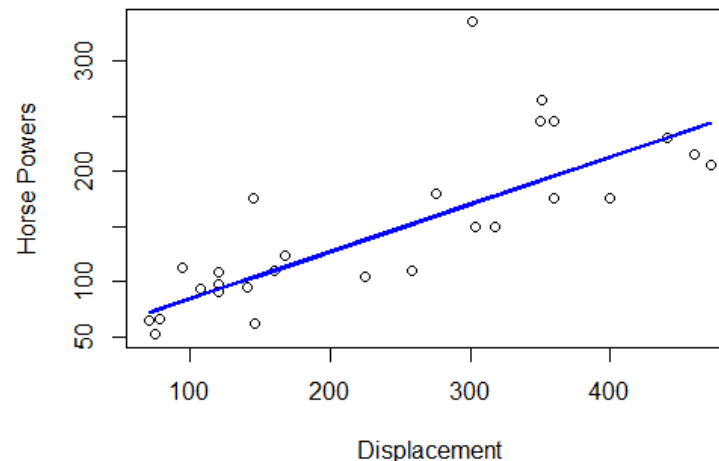
Cost minimization: comments

- Training a model with perfect accuracy unreasonable
 - Statistical noise for finite n
- Loss function can be different from error function
- Some loss functions are not good for training, for ex. misclassification rate.



Loss functions

- Assuming a distribution, **derive as minus log-likelihood**:
- $y \sim \text{Normal}(f_\theta(x), \sigma^2) \rightarrow L(y, f_\theta(x)) = (y - f_\theta(x))^2$
- Heavy outliers $y \sim \text{Laplace}(f_\theta(x), r) \rightarrow L(y, f_\theta(x)) = |y - f_\theta(x)|$

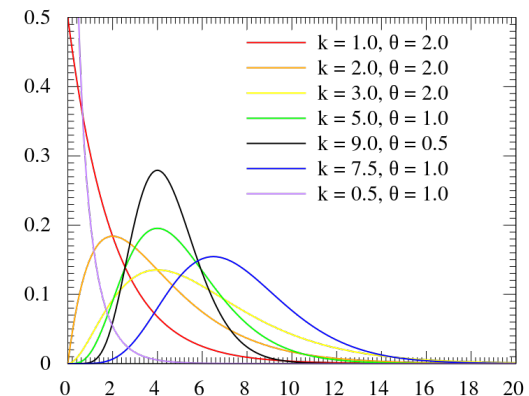
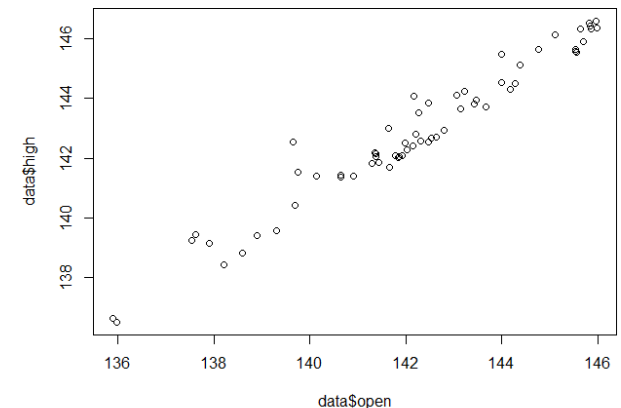
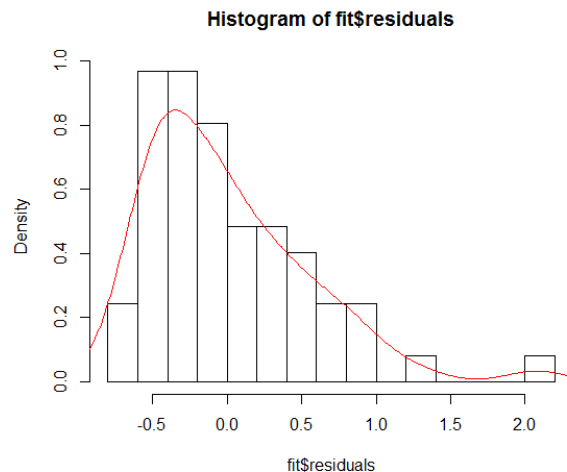


Loss functions

- Count data $y \sim \text{Poisson}(f_\theta(x))$

Example: Daily Stock prices NASDAQ

- Open
 - High (within day)
1. Try to fit usual linear regression, study histogram of residuals



Loss functions

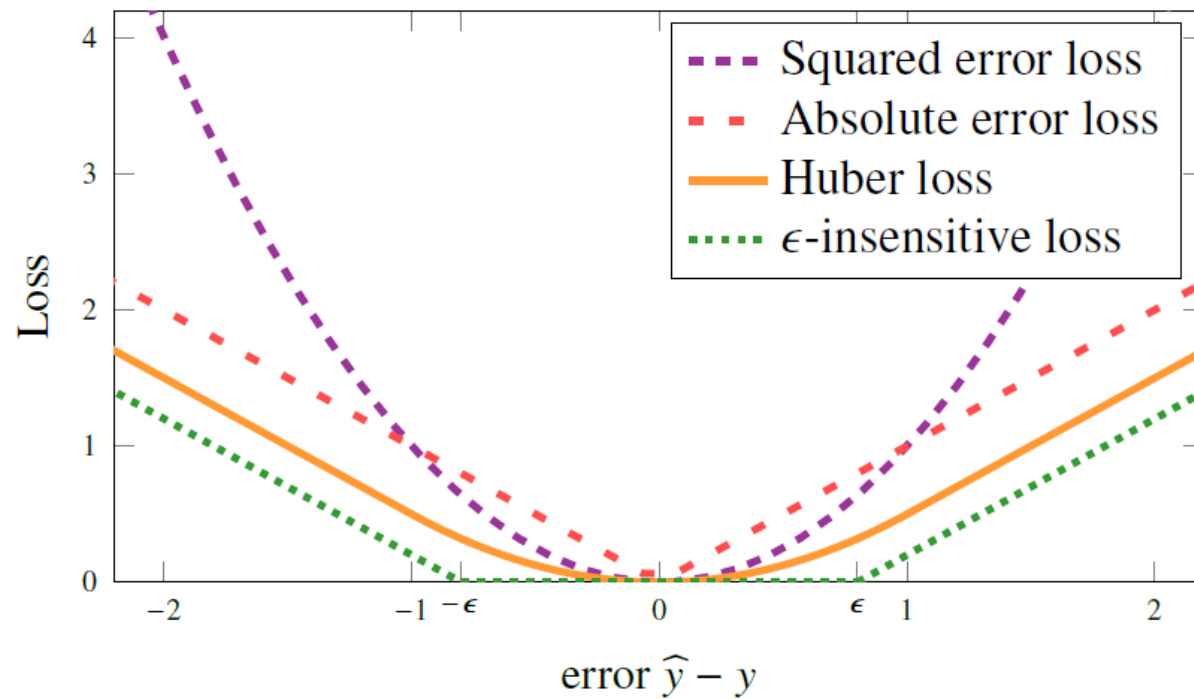
- If the distribution is difficult to assume / only some properties known → **ad-hoc loss functions**
- **Huber loss**: similar to quadratic but robust to outliers

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(\hat{y} - y)^2 & \text{if } |\hat{y} - y| < 1, \\ |\hat{y} - y| - \frac{1}{2} & \text{otherwise.} \end{cases}$$

- **E-insensitive loss**

$$L(y, \hat{y}) = \begin{cases} 0 & \text{if } |\hat{y} - y| < \epsilon, \\ |\hat{y} - y| - \epsilon & \text{otherwise,} \end{cases}$$

Loss functions



Loss functions: classification

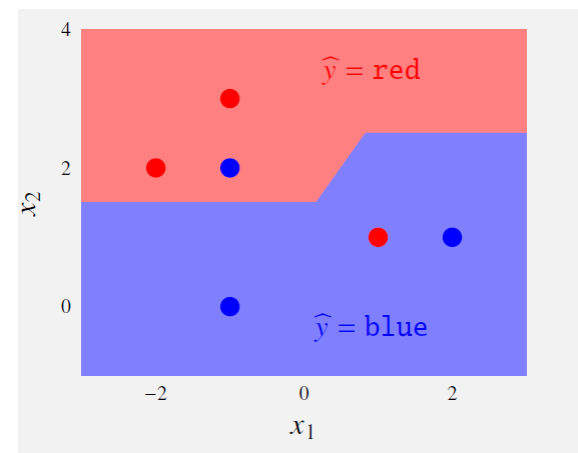
- **Cross-entropy** corresponds to minus log-likelihood:

$$J(y, \hat{p}(y)) = - \sum_{i=1}^n \sum_{m=1}^M I(y_i = C_m) \log \hat{p}(y_i = C_m)$$

- Ad-hoc loss functions binary classification $\mathcal{C} = \{-1, 1\}$
 - Assume model returns $f(\mathbf{x})$: $\hat{y} = \text{sign}(f(\mathbf{x}))$

- **Example:** logistic $f(\mathbf{x}) = \frac{1}{1+e^{-\theta^T \mathbf{x}}} - 0.5$

- **Note:** mistake when $yf(\mathbf{x}) < 0$



Loss functions: classification

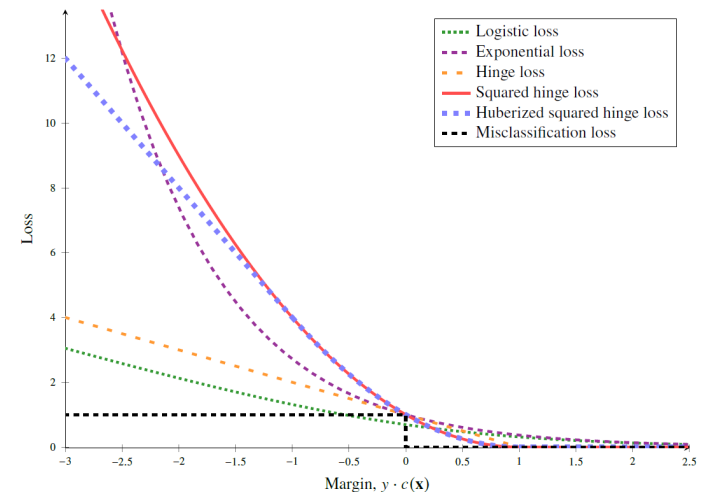
Ad-hoc loss functions binary classification

- Exponential loss**

$$L(y \cdot f(\mathbf{x})) = \exp(-y \cdot f(\mathbf{x}))$$

- Hinge loss**

$$L(y \cdot f(\mathbf{x})) = \begin{cases} 1 - y \cdot f(\mathbf{x}) & \text{for } y \cdot f(\mathbf{x}) \leq 1 \\ 0 & \text{otherwise.} \end{cases}$$



Loss functions: classification

Binary to multiclass

- **One versus one**: class C_i vs class C_j + majority voting from all classifiers
- **One versus rest**: class C_i vs not C_i + highest probability class
- **Comparison**: OVO needs less data to train one model but more models.

Regularization

- $E_{new} \approx J(\theta)$? – no
- Similar for (moderately) simple models, not similar for too complex model (overfitting).
- **Explicit regularization**: penalize complexity by changing cost function
- **Implicit regularization**: **early stopping**
 - If cost function optimized iteratively, don't let it decrease too much

Explicit regularization

- Penalize cost function

$$\min_{\theta} J(\theta) + \lambda R(\theta)$$

- $\lambda > 0$
- **L1 regularization:** $R(\theta) = \lambda \|\theta\|_1$
- **L2 regularization:** $R(\theta) = \lambda \|\theta\|_2$
- **Example:** Ridge regression

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^n (\mathbf{y}_i - \theta^T \mathbf{x}_i)^2 + \lambda \sum_{j=1}^p \theta_j^2, \quad \lambda > 0$$

Explicit regularization: ridge regression

Equivalent form

$$\hat{\theta}^{ridge} = \operatorname{argmin} \sum_{i=1}^N (y_i - \theta_0 - \theta_1 x_{1i} - \dots - \theta_p x_{pi})^2$$

subject to $\sum_{j=1}^p \theta_j^2 \leq s$

Solution

$$\theta^{ridge} = (X^T X + \lambda I)^{-1} X^T y$$

Ridge regression

Properties

- Extreme cases:
 - $\lambda = 0$ usual linear regression (no shrinkage)
 - $\lambda = +\infty$ fitting a constant ($\theta = 0$ except of θ_0)
- Degrees of freedom decrease when λ increases
 - $\lambda = 0 \rightarrow d.f. = p$
- $p > n$ is doable
 - Compare with linear regression
- How to estimate λ ?
 - cross-validation

Ridge regression

Example Computer Hardware Data Set : performance measured for various processors and also

- Cycle time
- Memory
- Channels
- ...

Build model predicting performance



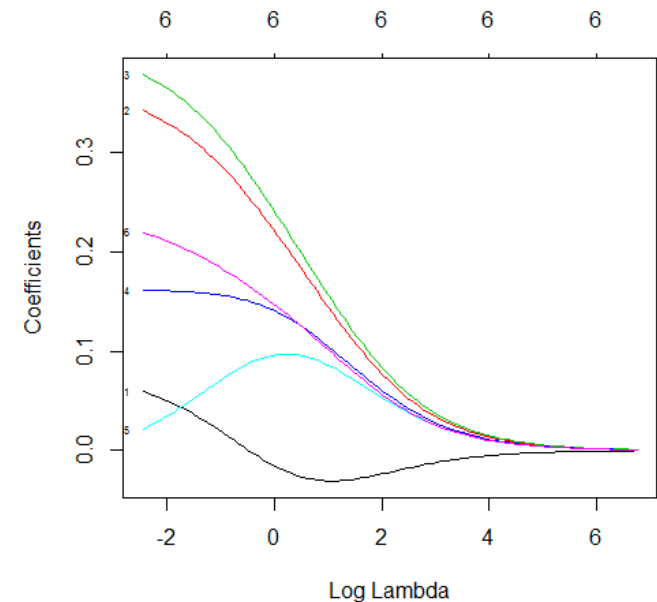
Ridge regression

- R code: use package **glmnet** with $\alpha=0$ (Ridge regression)
- Seeing how Ridge converges

```
data=read.csv("machine.csv", header=F)
```

```
library(caret)  
library(glmnet)  
scaler=preProcess(data)  
data1=predict(scaler, data)  
covariates=data1[,3:8]  
response=data1[, 9]
```

```
model0=glmnet(as.matrix(covariates),  
response, alpha=0,family="gaussian")  
plot(model0, xvar="lambda", label=TRUE)
```



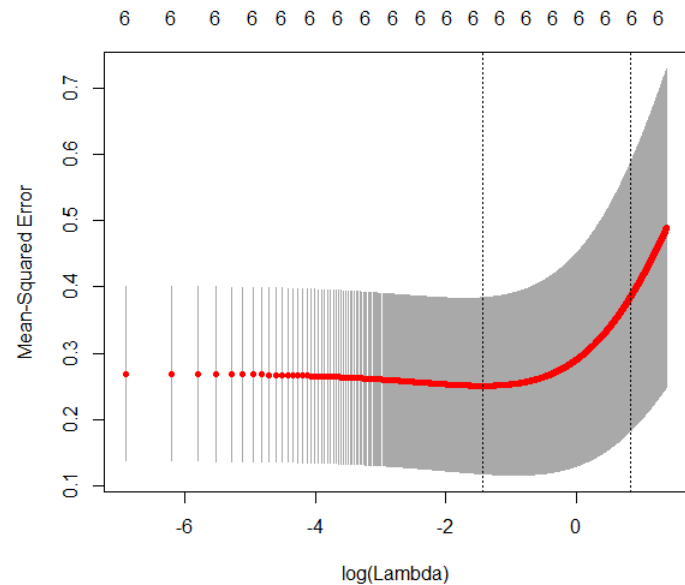
Ridge regression

- Choosing the best model by cross-validation:

```
model=cv.glmnet(as.matrix(covariates),
response, alpha=0,family="gaussian")
model$lambda.min
plot(model)
coef(model, s="lambda.min")
```

```
> coef(model, s="lambda.min")
7 x 1 sparse Matrix of class "dgCMatrix"

      1
(Intercept) -4.530442e-17
v3          3.420739e-02
v4          3.085696e-01
v5          3.403839e-01
v6          1.593470e-01
v7          5.489116e-02
v8          1.970982e-01
```



```
> model$lambda.min
[1] 0.046
```

Ridge regression

- How good is this model in prediction?

```
covariates=train[,1:6]
response=train[, 7]
model=cv.glmnet(as.matrix(covariates), response, alpha=0,family="gaussian",
lambda=seq(0,1,0.001))
y=test[,7]
ynew=predict(model, newx=as.matrix(test[, 1:6]), type="response")
```

```
#Coefficient of determination
sum((ynew-mean(y))^2)/sum((y-mean(y))^2)
```

```
sum((ynew-y)^2)
```

Note that data are so small so numbers
change much for other train/test

```
> sum((ynew-mean(y))^2)/sum((y-mean(y))^2)
[1] 0.5438148
> sum((ynew-y)^2)
[1] 18.04988
> |
```


LASSO

- Add l_1 regularization term

$$\hat{\theta}^{lasso} = \operatorname{argmin} \left\{ \frac{1}{n} \sum_{i=1}^n (y_i - \theta_0 - \theta_1 x_{1i} - \dots - \theta_p x_{pi})^2 + \lambda \sum_{j=1}^p |\theta_j| \right\}$$

- $\lambda > 0$ is **penalty factor**
- Equivalent formulation

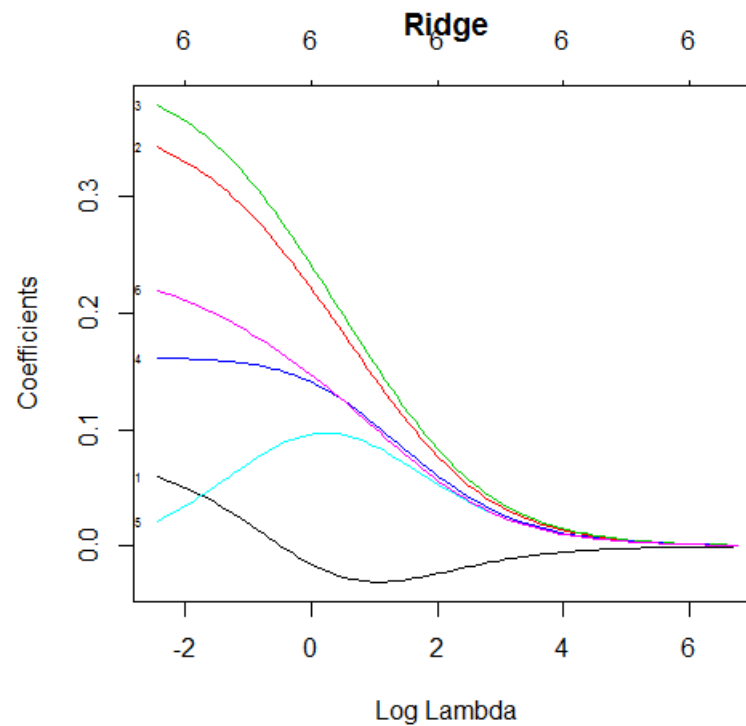
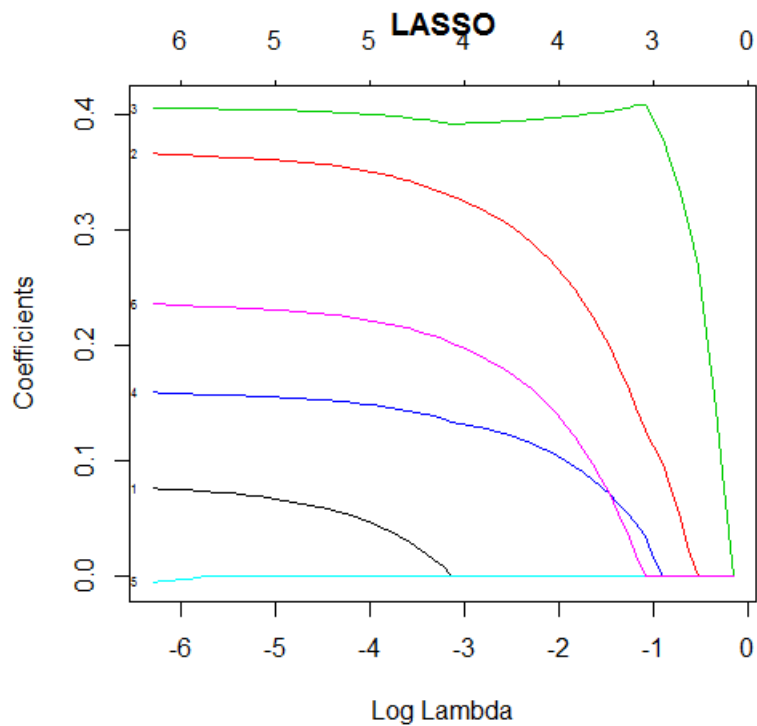
$$\begin{aligned} \hat{\theta}^{lasso} &= \operatorname{argmin} \sum_{i=1}^n (y_i - \theta_0 - \theta_1 x_{1i} - \dots - \theta_p x_{pi})^2 \\ \text{subject to } &\sum_{j=1}^p |\theta_j| \leq s \end{aligned}$$



LASSO vs Ridge

- **LASSO yields sparse solutions!**

Example Computer hardware data



LASSO vs Ridge

- In R, use glmnet with **alpha=1**
- Only 5 variables selected by LASSO

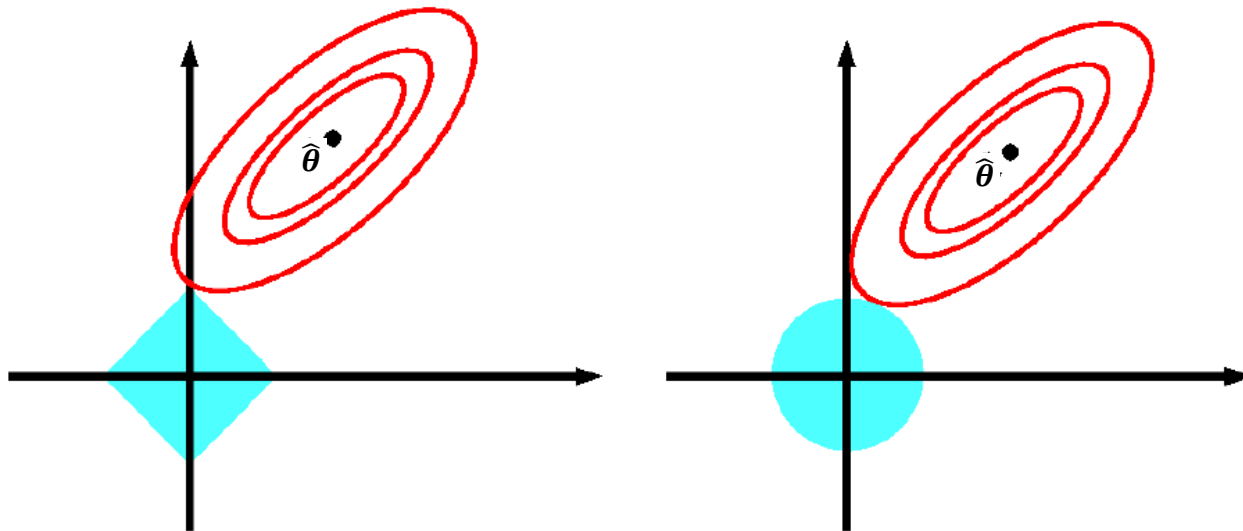
```
> coef(model, s="lambda.min")  
7 x 1 sparse Matrix of class "dgCMatrix"
```

```
      1  
(Intercept) -5.091825e-17  
v3           6.350488e-02  
v4           3.578607e-01  
v5           4.033670e-01  
v6           1.541329e-01  
v7           .  
v8           2.287134e-01  
> l
```

```
> sum((ynew-mean(y))^2)/sum((y-mean(y))^2)  
[1] 0.5826904  
> sum((ynew-y)^2)  
[1] 16.63756
```

LASSO vs Ridge

- Why Lasso leads to sparse solutions?
 - Feasible area for Ridge is a circle (2D)
 - Feasible area for LASSO is a polygon (2D)



LASSO properties

- **Lasso is widely used when $p \gg n$**
 - Linear regression breaks down when $p > n$
 - Application: DNA sequence analysis, Text Prediction
- No explicit formula for $\hat{\theta}^{lasso}$
 - Optimization algorithms used

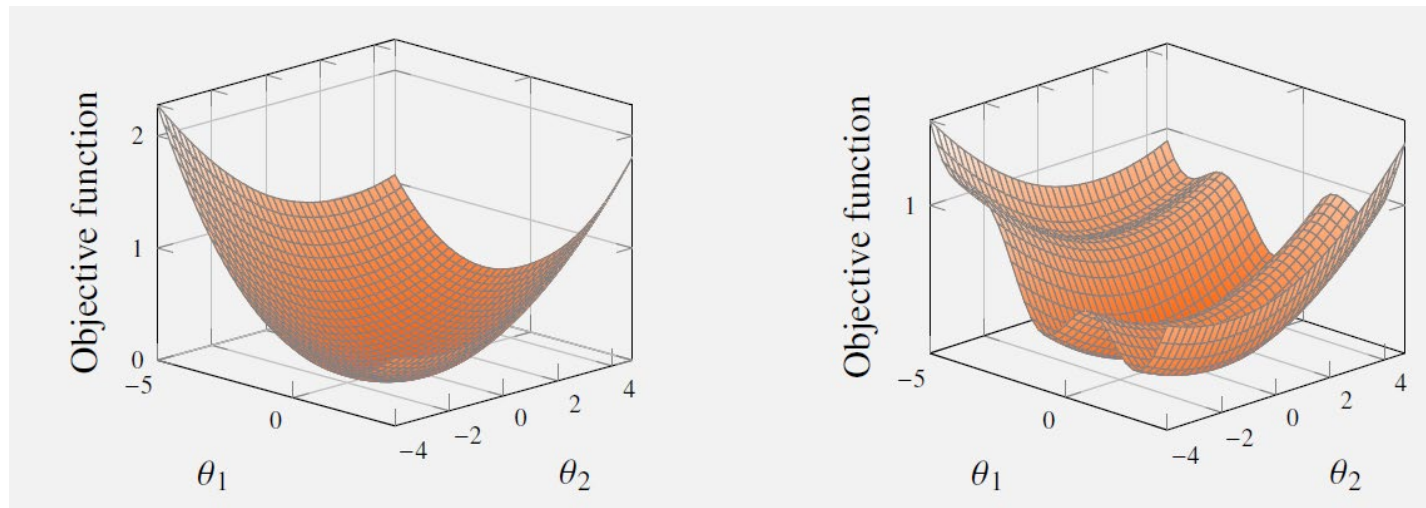
Optimization methods

- Numerical optimization often needed

$$\min_{\theta} J(\theta)$$

$$\min_{\lambda} E_{hold-out}(\lambda)$$

- If not convex objective, more than one local optimum



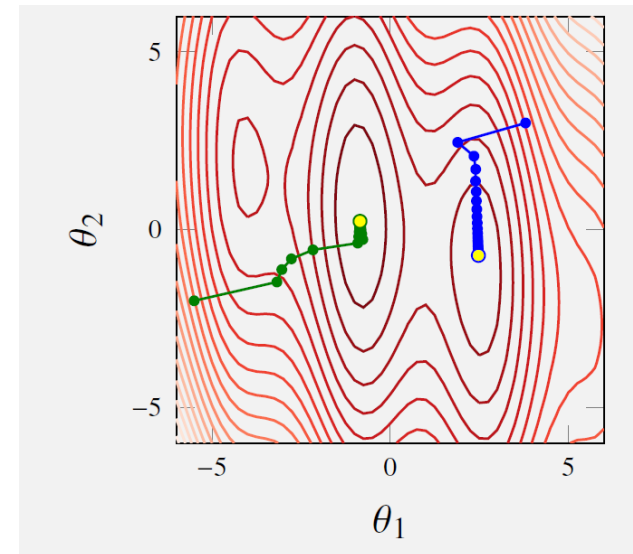
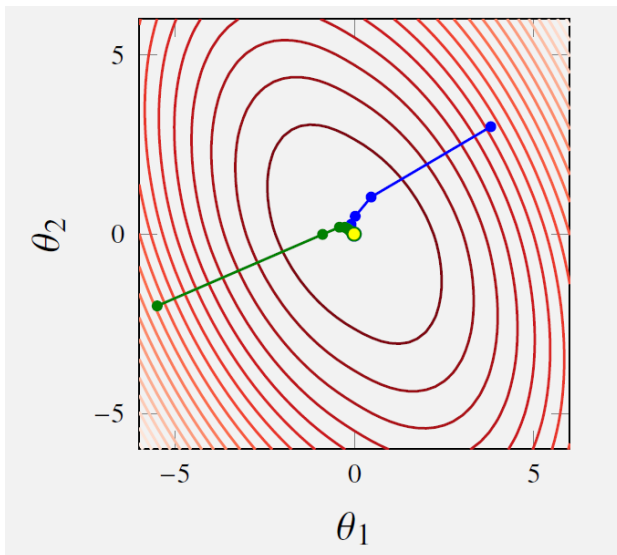
Optimization methods

- **Gradient descent method**

$$\hat{\theta} = \arg \min_{\theta} J(\theta)$$

- Basic idea:

- Start from some point θ_0
- Move to the next point along **descent direction** $-\nabla_{\theta} J(\theta)$



Gradient descent

Algorithm 5.1: Gradient descent

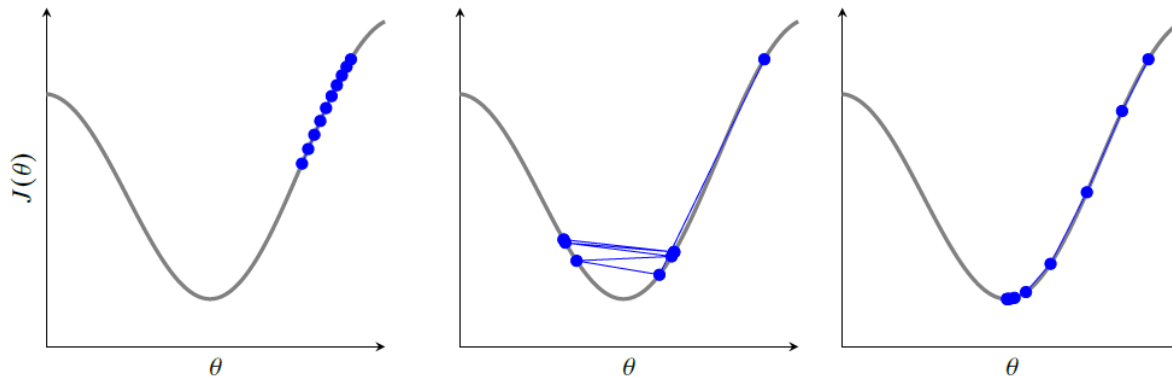
Input: Objective function $J(\theta)$, initial $\theta^{(0)}$, learning rate γ

Result: $\hat{\theta}$

- 1 Set $t \leftarrow 0$
 - 2 **while** $\|\theta^{(t)} - \theta^{(t-1)}\|$ *not small enough* **do**
 - 3 Update $\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma \nabla_{\theta} J(\theta^{(t)})$
 - 4 Update $t \leftarrow t + 1$
 - 5 **end**
 - 6 **return** $\hat{\theta} \leftarrow \theta^{(t-1)}$
-

Gradient descent

- Influence of γ

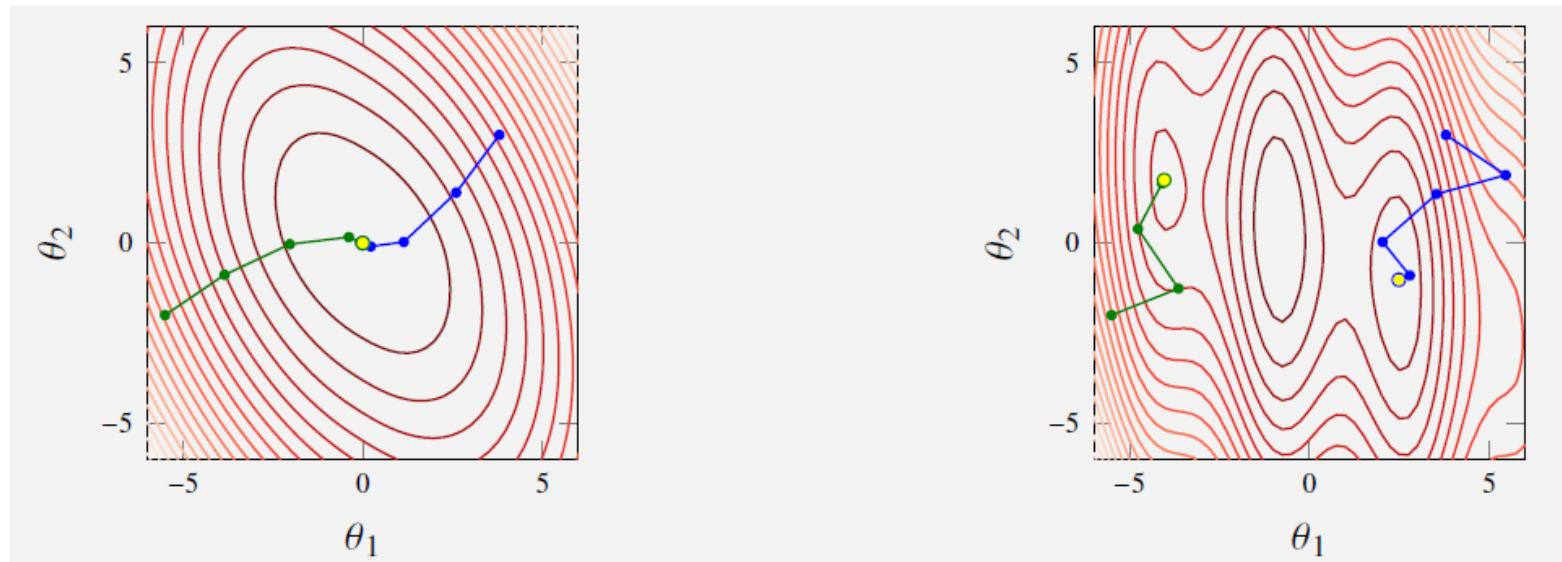


- Trace $J(\theta^{(t)})$ vs t
 - High oscillation \rightarrow decrease γ
 - Slow changes \rightarrow increase γ
- Try with different $\theta^{(0)}$ if possible

Newton's method

- Assume $J(\boldsymbol{\theta})$ is "locally" quadratic
- Newton's method: move along the best direction

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta [\nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}^{(t)})]^{-1} [\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]$$



Newton's method

- Properties
 - No convergence guarantees
 - **Advantage**: if $J(\theta)$ is quadratic and $\eta = 1 \rightarrow$ convergence in one iteration
 - **Disadvantage 1**: Hessian must be invertable
 - **Disadvantage 2**: Hessian is computationally heavy
- Solution: quasi-Newton methods (ex. **BFGS**)
 - Approximate inverse Hessian at each step numerically

Newton's method

Algorithm 5.2: Trust-region Newton's method

Input: Objective function $J(\boldsymbol{\theta})$, initial $\boldsymbol{\theta}^{(0)}$, trust region radius D

Result: $\hat{\boldsymbol{\theta}}$

- 1 Set $t \leftarrow 0$
 - 2 **while** $\|\boldsymbol{\theta}^{(t)} - \boldsymbol{\theta}^{(t-1)}\|$ *not small enough* **do**
 - 3 Compute $\mathbf{v} \leftarrow [\nabla_{\boldsymbol{\theta}}^2 J(\boldsymbol{\theta}^{(t)})]^{-1} [\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})]$
 - 4 Compute $\eta \leftarrow \frac{D}{\max(\|\mathbf{v}\|, D)}$
 - 5 Update $\boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \mathbf{v}$
 - 6 Update $t \leftarrow t + 1$
 - 7 **end**
 - 8 **return** $\hat{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta}^{(t-1)}$
-

Optimization methods in R

- In R, use `optim(par, fn, gr, method,...)`
 - `par`: initial parameter vector
 - `fn`: function to optimize
 - `gr`: gradient function
 - `method`

Example: trace plot for $y = (x_1 - 2)^4 + (x_2 - 4)^4$

Optimization methods in R

#Workaround: optim does not return iterations

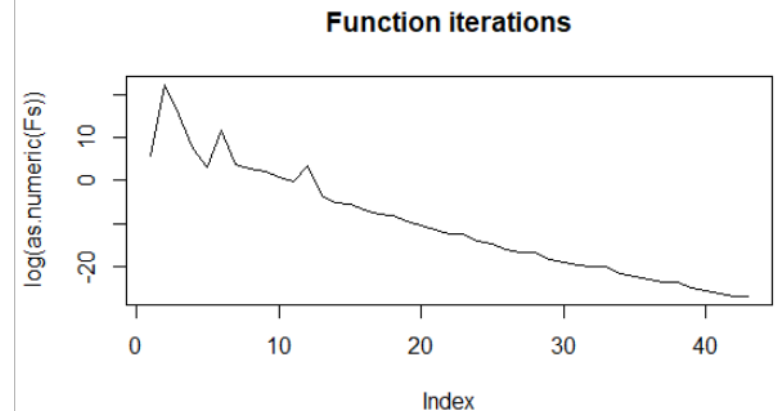
```
Fs=list()
Params=list()
k=0

myf<- function(x){
  f=(x[1]-2)^4+(x[2]-4)^4
  .GlobalEnv$k= .GlobalEnv$k+1
  .GlobalEnv$Fs[[k]]=f
  .GlobalEnv$Params[[k]]=x
  return(f)
}

myGrad <-function(x) c(4*(x[1]-2)^3, 4*(x[2]-4)^3)

res<-optim(c(0,0), fn=myf, gr=myGrad, method="BFGS")

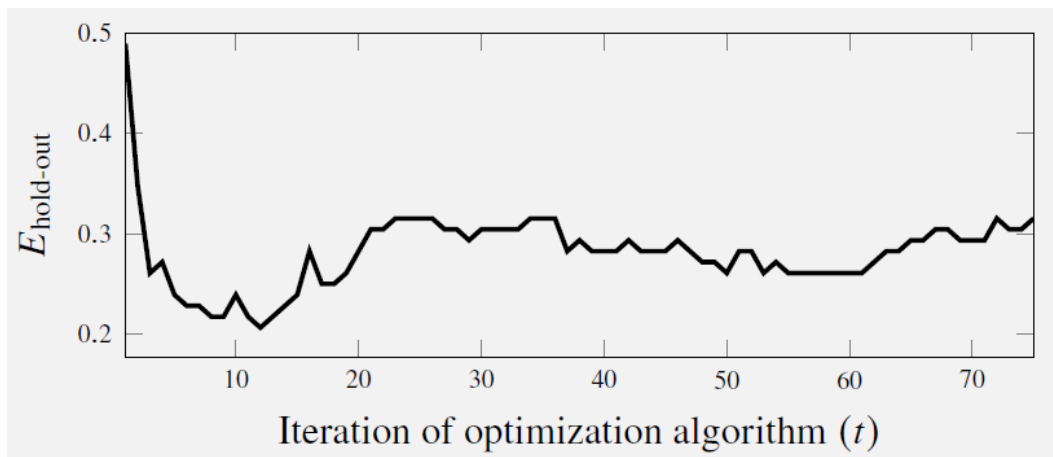
plot(log(as.numeric(Fs)), type="l", main="Function
iterations")
```



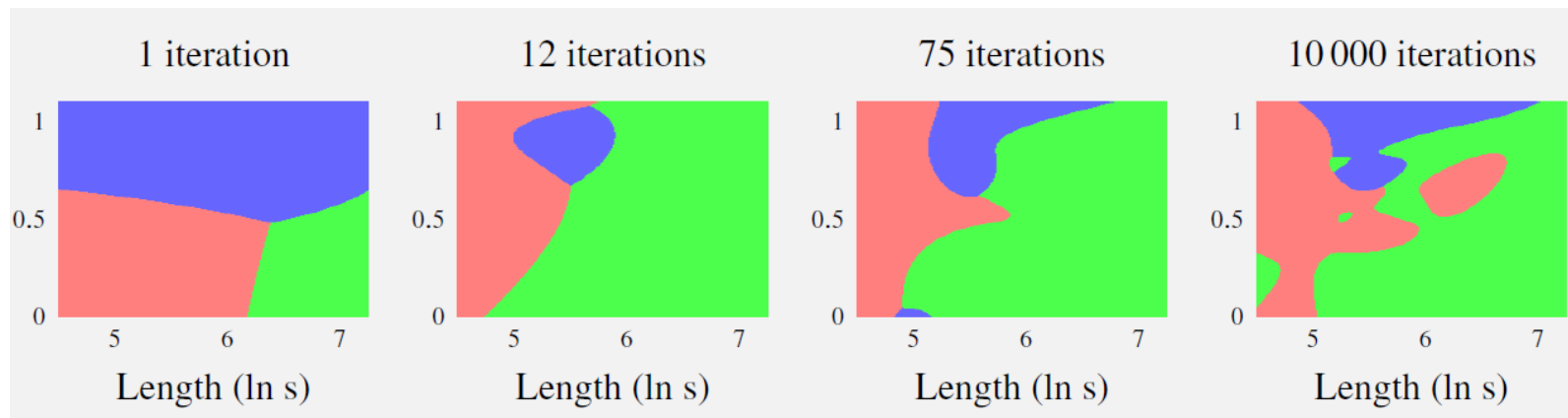
Implicit regularization

- **Early stopping**

- For complex models, accurate model optimization may lead to overfitting
- Start from some parameter set (probably not optimal, large E_{train} and E_{new})
- Trace the validation error (and training error?) for each t
- Choose model with the smallest validation error



Implicit regularization



Optimization for large data

Stochastic gradient descent

Idea: use gradient descent + approximation to expected value

- For **random** sample of size n_b from sample of size n

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \approx \frac{1}{n_b} \sum_{i=1}^{n_b} x_i$$
$$\nabla_{\theta} J(\theta) \approx \frac{1}{n_b} \sum_{(x_i, y_i) \in \text{sample}} \nabla_{\theta} L(x_i, y_i, \theta)$$

1. One **epoch**:
 1. Permute data and divide into batches of size n_b
 2. In each optimization iteration, use one batch
2. Repeat step 1

Stochastic gradient descent

Algorithm 5.3: Stochastic gradient descent

Input: Objective function $J(\theta) = \frac{1}{n} \sum_{i=1}^n L(\mathbf{x}_i, y_i, \theta)$, initial $\theta^{(0)}$, learning rate $\gamma^{(t)}$

Result: $\hat{\theta}$

```
1 Set  $t \leftarrow 0$ 
2 while Convergence criteria not met do
3   for  $i = 1, 2, \dots, E$  do
4     Randomly shuffle the training data  $\{\mathbf{x}_i, y_i\}_{i=1}^n$ 
5     for  $j = 1, 2, \dots, \frac{n}{n_b}$  do
6       Approximate the gradient using the mini-batch  $\{(\mathbf{x}_i, y_i)\}_{i=(j-1)n_b+1}^{jn_b}$ ,
7        $\hat{\mathbf{d}}^{(t)} = \frac{1}{n_b} \sum_{i=(j-1)n_b+1}^{jn_b} \nabla_{\theta} L(\mathbf{x}_i, y_i, \theta^{(t)})$ .
8       Update  $\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma^{(t)} \hat{\mathbf{d}}^{(t)}$ 
9       Update  $t \leftarrow t + 1$ 
10    end
11  end
12 return  $\hat{\theta} \leftarrow \theta^{(t-1)}$ 
```

- Different choices for γ_t , for ex $\gamma^{(t)} = \frac{1}{t^\alpha}, \alpha \in (0.5, 1]$

Hyperparameter optimization

- $E_{hold-out}$ costly to compute \rightarrow usual optimization very hard
 - Note: for each λ first we need to optimize θ ...+ gradients of $E_{hold-out}$
- Grid search (can also be costly)
 - Alternative: Bayesian optimization

