CS5014 Machine Learning

Lecture 9 Fixed basis expansion models

Lei Fang

School of Computer Science, University of St Andrews

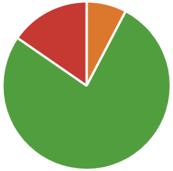
Spring 2021

University of St Andrews

Poll result last time

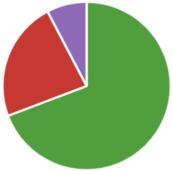
1. How do you think about the level of difficulty of maths ?

●	too easy	0
●	easy to follow	1
●	challenging but manageable	10
●	too hard	2



2. Speed of delivery?

●	Too slow	0
●	can be faster	0
●	good	9
●	can be slower	3
●	Too fast	1



Response

Speed: interrupt me during the lecture if you think I am too fast

- very hard for me to gauge the speed
- w/o visual clue, I do not know where to spend more/less time

Difficulty: for those who think this course is too easy (there were at least 2 said so in the first poll)

- minority but still not ideal to you
- I will add extra exercises and one relevant ML research paper at the end of each lecture
- you can also talk to us for further reading/exercises
- how much you get out from the course is up to you

Response to comments

Comment 1: `"little bit on the Hessian matrix"`

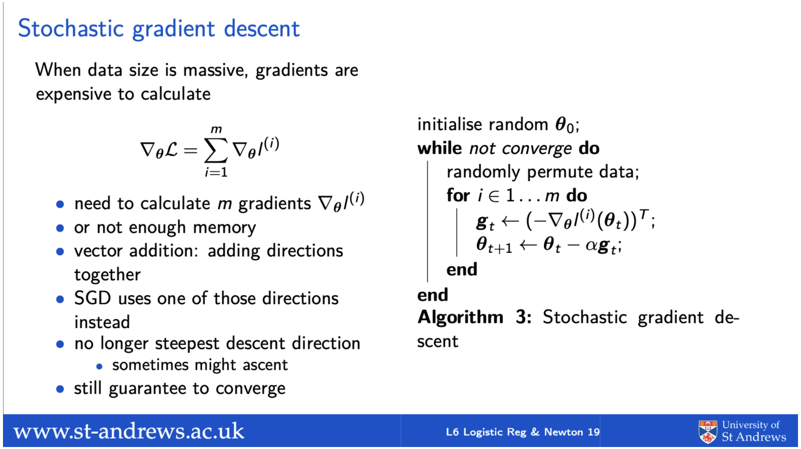
- **Response**: we will do it today

Comment 2: `"need help with all the maths"`

- **Response**: Kasim and I are running extra lab sessions in even weeks; bring your questions. Can also schedule meetings with me.

Response to comments

Commen 3: "SGD on slide 19 is confusing ... is it the gradient for an individual feature or data point"



Stochastic gradient descent

When data size is massive, gradients are expensive to calculate

$$\nabla_{\theta}\mathcal{L} = \sum_{i=1}^{m} \nabla_{\theta} l^{(i)}$$

- need to calculate m gradients $\nabla_{\theta} l^{(i)}$
- or not enough memory
- vector addition: adding directions together
- SGD uses one of those directions instead
- no longer steepest descent direction
  - sometimes might ascent
- still guarantee to converge

```
initialise random $\boldsymbol{\theta}_0$;
while not converge do
    randomly permute data;
    for $i \in 1 \ldots m$ do
        $\boldsymbol{g}_t \leftarrow (-\nabla_{\theta} l^{(i)}(\boldsymbol{\theta}_t))^T$;
        $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$;
    end
end
```

Algorithm 3: Stochastic gradient descent

Response: Good confusion/question! It worths some further discussions. Remember the total likelihood is

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{m} \underbrace{y^{(i)} \log \sigma^{(i)} + (1 - y^{(i)}) \log(1 - \sigma^{(i)})}_{l^{(i)}(\boldsymbol{\theta})} = \sum_{i=1}^{m} l^{(i)}(\boldsymbol{\theta}), \quad \text{therefore}$$

$$\nabla_{\boldsymbol{\theta}} \mathcal{L} = \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} l^{(i)} \quad \text{due to linearity of derivative: } \nabla \sum f_i(x) = \sum \nabla f_i(x)$$

- total gradient (LHS) is a sum of invidual gradients (of each data point (i))
- SGD uses $\nabla_{\boldsymbol{\theta}} l^{(i)}$ instead of $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ at each iteration
- also note the notation for future reference
  - **superscript** (i) (with "**()**") index data points, $y^{(i)}, \mathbf{x}^{(i)}, l^{(i)}$ *etc.*
  - **subscript** j index features (w/o brackets) $\boldsymbol{\theta}^T \mathbf{x}^{(i)} = \sum_{j=1}^{n} x_j^{(i)} \theta_j$

On the other hand, optimise w.r.t features (i.e. descent each θ_j in turn) is an algorithm called **coordinate descent**

- not particularly useful for linear regression or logistic regression
  - slower descent than gradient descent, why ?
  - in fact descent by axis aligned directional derivatives (with the relevant partial derivatives entries)
- useful when θ_i, θ_j are coupled though
  - descent is easier for θ_2 if conditional on θ_1
  - vice versa
  - numerically stabler than direct optimisation
- EM (Expectation Maximization) is an example
  - mixture models or Hidden Markov Models (HMMs)
  - we will study it later

Plan for today

Learning algorithm:

- demystify Hessian
- cases for and against Newton's method
- practical issues of gradients

Model:

- probabilistic view unifies classification and regression
- fixed basis models: extends linear models to nonlinear modelss
    - but still linear models: only needs to transform your data
    - functional space view

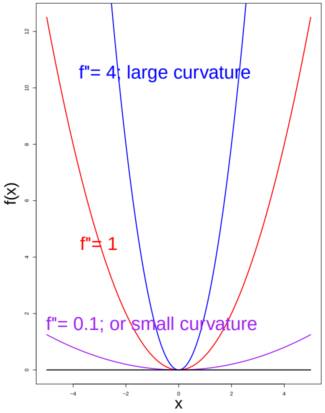
Vignette: second derivative or curvature

First, recap on univariate quadratics: $f(x) = ax^2$

- multivariate version: $f(\boldsymbol{x}) = \boldsymbol{x}^T \boldsymbol{A} \boldsymbol{x}$
- gradient: $f' = 2ax$ ($\nabla_{\boldsymbol{x}} f = 2\boldsymbol{x}^T \boldsymbol{A}$ for multivariate $\boldsymbol{x}$)
- second derivative (Hessian): $f'' = 2a$ ($\nabla^2_{\boldsymbol{x}} f = \boldsymbol{H}(f) = 2\boldsymbol{A}$)
- btw: effects of adding bx and c terms to f? (shifting f to another location; f'' doesn't change)

So what f'' really is ?

- **curvature**: curviness a curve deviates from straight line
  - large curvature → bendy curve
  - small curvature → flatter curve
- **Hessian**? the multivariant version of curvature



Vignette: directional curvature and Hessian

Hessian: multivariant curvature. how ?

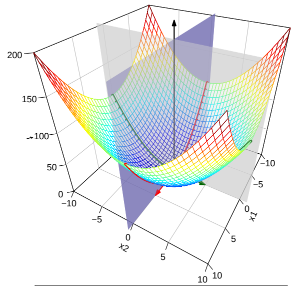
$$\boldsymbol{u}^T \boldsymbol{H}\boldsymbol{u} \text{ is directional curvature}$$

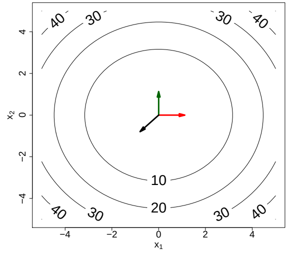
if $\boldsymbol{u} \in R^2$ is a unit vector $||\boldsymbol{u}||_2^2 = \boldsymbol{u}^T\boldsymbol{u} = 1$ (represents a direction in the input space)

Example: $f(\boldsymbol{x}) = x_1^2 + x_2^2$ or $\boldsymbol{A} = \boldsymbol{I}$; Hessian:

$$\boldsymbol{H} = 2\boldsymbol{I} = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

- if $\boldsymbol{u}_1 = [1, 0]^T$: $\boldsymbol{u}_1^T \boldsymbol{H}\boldsymbol{u}_1 = 2$ horizontal curvature
- if $\boldsymbol{u}_2 = [0, 1]^T$: $\boldsymbol{u}_2^T \boldsymbol{H}\boldsymbol{u}_2 = 2$ vertical curvature
- actually, curvature is 2 for all directions! (expected as it is a circle) $\boldsymbol{u}^T\boldsymbol{H}\boldsymbol{u} = 2\boldsymbol{u}^T\boldsymbol{u} = 2$





Vignette: directional curvature and Hessian

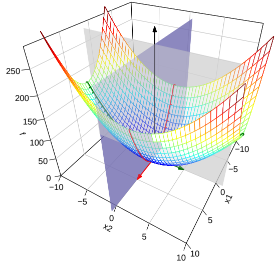
Example: $f(\boldsymbol{x}) = 4x_1^2 + x_2^2$ or $\boldsymbol{A} = \begin{bmatrix} 4, 0 \\ 0, 1 \end{bmatrix}$. Hessian:

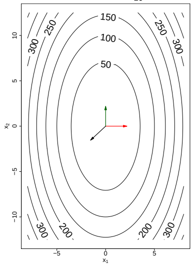
$$\boldsymbol{H} = 2\begin{bmatrix} 4, 0 \\ 0, 1 \end{bmatrix} = \begin{bmatrix} 8 & 0 \\ 0 & 2 \end{bmatrix}$$

- if $\boldsymbol{u}_1 = [1, 0]^T$: $\boldsymbol{u}_1^T \boldsymbol{H} \boldsymbol{u}_1 = 8$ horizontal curvature (curvy)
- if $\boldsymbol{u}_2 = [0, 1]^T$: $\boldsymbol{u}_2^T \boldsymbol{H} \boldsymbol{u}_2 = 2$ vertical curvature (less curvy)
- the curvature changes with directions now

$$\boldsymbol{u}^T \boldsymbol{H} \boldsymbol{u}$$

is not a constant





Newton's direction and Hessian

Newton's method step:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \underbrace{\boldsymbol{H}_t^{-1}\boldsymbol{g}_t}_{\boldsymbol{d}_t}$$

- Newton's direction: $\boldsymbol{d}_t = \boldsymbol{H}_t^{-1}\boldsymbol{g}_t$

Gradient descent step:

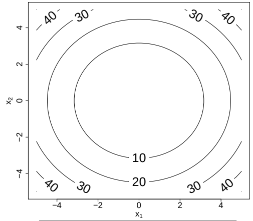
$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \alpha \boldsymbol{g}_t$$

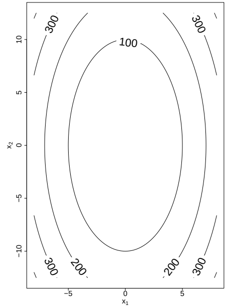
Can you explain now

- when $\boldsymbol{H}_t = \boldsymbol{I}$, $\boldsymbol{d}_t = \boldsymbol{g}_t$? what does it imply?
- the second case: $\boldsymbol{H}_t = \text{diag}(\{8, 2\})$?

Moral of the story?

- Shortcuts may not be ideal in long run. (very insightful !)





Newton's method for linear regression

Linear regression

$$\nabla_{\boldsymbol{\theta}} L = -2(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta})^T \boldsymbol{X};\ \text{and the Hessian is: } \nabla^2_{\boldsymbol{\theta}} L = 2\boldsymbol{X}^T\boldsymbol{X}$$

- constant Hessian matrix: the quadratic approximation is exact
- Newton step: assume starting from any $\boldsymbol{\theta}_0 \in R^n$

$$\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_0 - \underbrace{(2\boldsymbol{X}^T\boldsymbol{X})^{-1}}_{\boldsymbol{H}^{-1}} \underbrace{(-2\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\theta}_0))}_{\boldsymbol{g}_0}$$

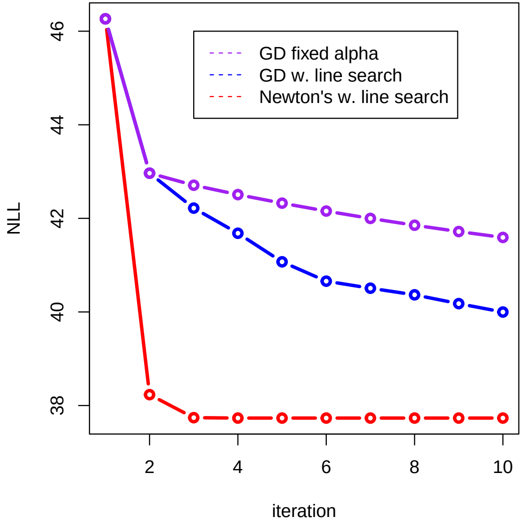
$$\boldsymbol{\theta}_1 \leftarrow \boldsymbol{\theta}_0 + (\boldsymbol{X}^T\boldsymbol{X})^{-1}(\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{\theta}_0) = \underbrace{(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}}_{\boldsymbol{\theta}_{ls}\text{: one step optimised}}$$

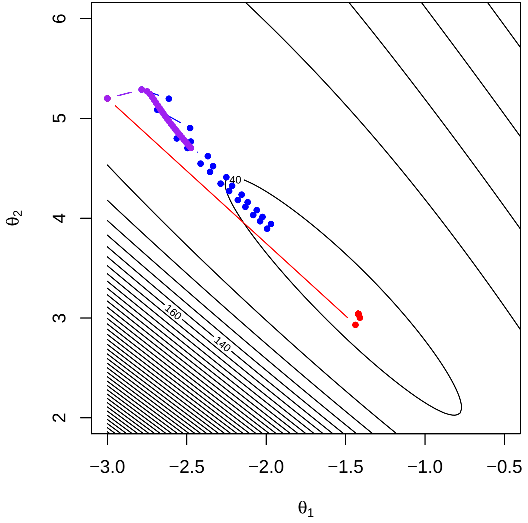
- why $\boldsymbol{\theta}_{ls}$ is a minimum ? $\boldsymbol{H} = 2\boldsymbol{X}^T\boldsymbol{X}$ is positive definite (similar to univariate function).

for all $\boldsymbol{u} \neq \boldsymbol{0}$, $\boldsymbol{u}^T\boldsymbol{H}\boldsymbol{u} = 2\underbrace{\boldsymbol{u}^T\boldsymbol{X}^T}_{\boldsymbol{v}^T}\underbrace{\boldsymbol{X}\boldsymbol{u}}_{\boldsymbol{v}} = 2\boldsymbol{v}^T\boldsymbol{v} > 0$ (vector's norm is always positive).

useful identities to find the Hessian: $\frac{\partial \boldsymbol{x}^T\boldsymbol{A}}{\partial \boldsymbol{x}} = \boldsymbol{A}^T$, $\frac{\partial \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{A}$

Newton's method for logistic regression (code on studres)





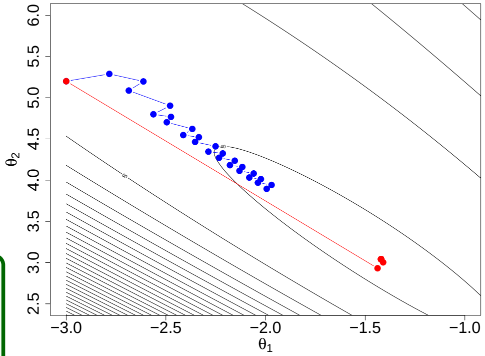
A closer look: Newton's method for logistic regression

Gradient descent: zig-zagging

- consecutive gradients are perpendicular
  - doesn't look so because of plotting ratio
- zig-zagging in a narrow valley

Newton's method

- converge faster
- does starting point matter?
- what if we start from top right corner ?
  - can you envision the cost function from the contour ?



All code to generate the figs/algorithms on studres

- in R at the moment;
- will translate to Python

Why not everyone is Newton in ML then?

Newton's method can diverge

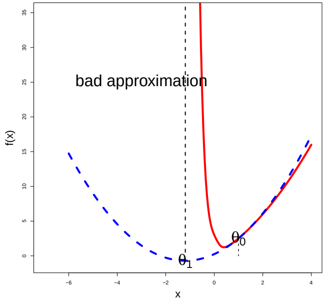
- bad quadratic approximation (top fig)
- might approximate a downward facing bowl (btm fig)
  - finding maximum instead !
  - Newton's direction can either descent or ascent
    - depends on the local approximation
    - need to check the Hessian for the lost function

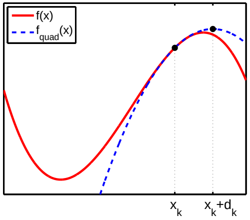
Remedy

- apply some learning rate $\alpha' < 1$ rather than exact Newton step: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}_t - \alpha' \boldsymbol{H}_t^{-1} \boldsymbol{g}_t$
  - line search to find α': even simply grid search in $(0, 1]$:

$$\alpha' \leftarrow \underset{\alpha}{\text{argmin}}\, L(\boldsymbol{\theta}_t - \alpha \boldsymbol{d}_t)$$

- or use gradient descent first; then Newton's method to speed up at the end





Why not everyone is Newton in ML then?

Newton's method is expensive

- Hessian: $n \times n$ derivatives to compute
- matrix is very expensive to invert in general

Hessian can be impossible to invert

- singular matrix ($\boldsymbol{H}^{-1}$ does not exist), similar to divide a number by 0
- *regularisation* is useful (adding small constants to diagonal entries)
    - I have used 0.01 in my Newton's algorithm above
- stochastic Newton's method ?
    - inverting a rank one matrix ! ($\boldsymbol{x}^{(i)}(\boldsymbol{x}^{(i)})^T$ is rank 1, so singular)
    - in human language: cannot estimate the curvature realiably with only one data

Quasi-Newton (Newton-like methods)

Quasi-Newton: approximate $\boldsymbol{H}_t$ (or even $\boldsymbol{H}_t^{-1}$ directly) instead

- some crude Hessian approximations work
  - gradient descent is one! $\boldsymbol{H}_t = ?$
  - some people use diagonal approximation matrix for $\boldsymbol{H}_t$
    - might perform well actually, what does it imply though ?
- Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) is more advanced
  - approximate Hessian by previous gradients
  - very widely used optimisation routine (Python, R, Matlab, Julia all have its implementation)
  - L-BFGS is limited memory version (do not use all the gradient history)

Gradients in real life

A Burning Question

how to find out the gradient and Hessian for a new loss $L(\boldsymbol{\theta})$?

Option 1: manual derivation: a lot of ML researchers do it this way

- method 1: use vector differentiation identities if you can (more efficient for implementation: vectorisation)

$$\frac{1}{2m}L(\boldsymbol{\theta}) = \frac{1}{2m}(\boldsymbol{y} - \boldsymbol{X\theta})^T(\boldsymbol{y} - \boldsymbol{X\theta}) \Rightarrow \frac{1}{2m}\nabla_{\boldsymbol{\theta}} L = \frac{1}{2m}\underbrace{\nabla_{\boldsymbol{y}-\boldsymbol{X\theta}}(\boldsymbol{y} - \boldsymbol{X\theta})^T(\boldsymbol{y} - \boldsymbol{X\theta})}_{2(\boldsymbol{y}-\boldsymbol{X\theta})^T} \cdot \underbrace{\nabla_{\boldsymbol{\theta}}(\boldsymbol{y} - \boldsymbol{X\theta})}_{-\boldsymbol{X}}$$

- method 2: or break down the total loss as a sum of individual losses; then sum up

$$\frac{1}{2m}L(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\underbrace{(y^{(i)} - \boldsymbol{\theta}^T\boldsymbol{x}^{(i)})^2}_{l^{(i)}(\boldsymbol{\theta})} \Rightarrow \nabla_{\boldsymbol{\theta}} l^{(i)} = 2(y^{(i)} - \boldsymbol{\theta}^T\boldsymbol{x}^{(i)})(-\boldsymbol{x}^{(i)})^T$$

Vector calculus shapes

		Scalar	Vector
		x	$\boldsymbol{x}$
Scalar	y	$\frac{\partial y}{\partial x}$ (scalar)	$\frac{\partial y}{\partial \boldsymbol{x}}$ (row vector)
Vector	$\boldsymbol{y}$	$\frac{\partial \boldsymbol{y}}{\partial x}$ (column vector)	$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ (matrix)

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}$$

$$\frac{\partial \boldsymbol{y}}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} \\ \frac{\partial y_2}{\partial x} \\ \vdots \\ \frac{\partial y_m}{\partial x} \end{bmatrix}$$

- notation: vectors are bold font letters, e.g. $\boldsymbol{x}$
- we use numerator notation here (easier to apply chain rule)
  - scalar y's gradient is a row vector, i.e. $\frac{\partial y}{\partial \boldsymbol{x}}$
  - vector $\boldsymbol{y}$'s derivative is a column vector, i.e. $\frac{\partial \boldsymbol{y}}{\partial x}$

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial \boldsymbol{x}} \\ \frac{\partial y_2}{\partial \boldsymbol{x}} \\ \vdots \\ \frac{\partial y_m}{\partial \boldsymbol{x}} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Check your gradients before use

Good coding practice apply in ML as well: testing testing testing

$$\boldsymbol{u}^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \approx \frac{1}{2\epsilon}(L(\boldsymbol{\theta} + \epsilon \boldsymbol{u}) - L(\boldsymbol{\theta} - \epsilon \boldsymbol{u}))$$

- check your derivation against the above rule
- for some small $\epsilon = 10^{-5}$ etc.
- $\boldsymbol{u}$ can be standard basis directions i.e. $[1, 0, 0, \ldots]^T, [0, 1, 0, \ldots]^T$

Gradients in real life

$$\frac{\partial L(\theta_j)}{\partial \theta_j} \approx \frac{1}{2\epsilon}(L(\theta_j + \epsilon) - L(\theta_j - \epsilon))$$

Option 2: finite difference approximation

- use approximated finite difference instead
- to evaluate gradient at $\boldsymbol{\theta}_t$,
  - choose small $\epsilon = 10^{-5}$
  - perturb each dimension in turn to find partial derivatives
- not practical for large model (esp. evaluation of L is expensive)
- subject to truncation error
- default choice for `scipy.optimize` method: approximation done for you

Gradients in real life

Option 3: auto differentiation

- an active (and old) research area in computing and ML
  - nobody enjoy doing derivatives by hand ...
- input: a function; output: gradient function
- use computation graph (like Neural Nets) to speed up/keep track the calculation
- backpropagation is one example
- a lot of packages out there: autograd in Python, PyTorch etc.

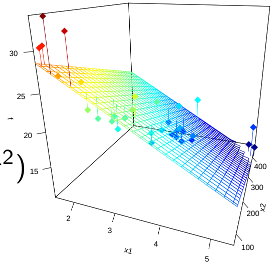
Example of autograd for logistic regression

```python
import autograd.numpy as np
from autograd import grad

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = np.log(preds) * targets + np.log((1 - preds)) * (1 - targets)
    return -np.sum(label_probabilities)

# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print("Initial loss:", training_loss(weights))
for i in range(100):
    weights -= training_gradient_fun(weights) * 0.01

print("Trained loss:", training_loss(weights))
```

Towards nonlinear models: regression

For linear regression:

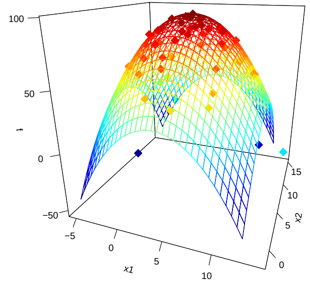
$$P(y|\boldsymbol{x},\boldsymbol{\theta}) = N(\underbrace{f(\boldsymbol{x};\boldsymbol{\theta})}_{\boldsymbol{\theta}^T\boldsymbol{x}\text{: linear}},\sigma^2) \text{ or } y = \underbrace{f(\boldsymbol{x};\boldsymbol{\theta})}_{\boldsymbol{\theta}^T\boldsymbol{x}} + \boldsymbol{\epsilon},\quad \epsilon^{(i)} \sim N(0,\sigma^2)$$



The regression function f is assumed linear

$$f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^T\boldsymbol{x}$$

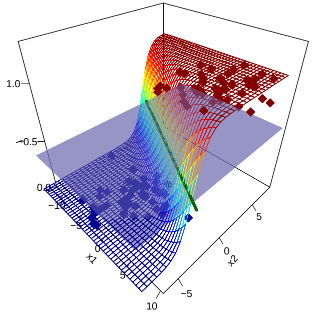
- *i.e.* fitting lines/hyperplanes
- in real life, a lot of relationships are not linear
- and we do not know what $f(\boldsymbol{x})$ should look like !



Towards nonlinear models: classification

For logistic regression:

$$P(y|\boldsymbol{x},\boldsymbol{\theta}) = \text{Ber}(\sigma(\underbrace{f(\boldsymbol{x};\boldsymbol{\theta})}_{\boldsymbol{\theta}^T\boldsymbol{x}\text{: linear}})) = \sigma^y(1-\sigma)^{(1-y)}$$

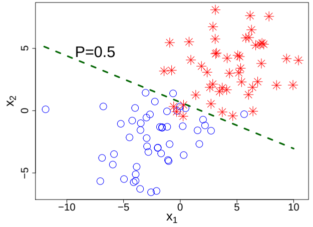


To predict the label y for any input $\boldsymbol{x}$:

$$y = 1 \text{ if } P(y=1|\boldsymbol{x},\boldsymbol{\theta}) > 0.5 \quad y = 0 \text{ if otherwise}$$

Note that the **decision boundary** is linear (hyperplane or line)

$$P(y=1|\boldsymbol{x},\boldsymbol{\theta}) = 0.5 \Rightarrow \boldsymbol{\theta}^T\boldsymbol{x} = 0$$

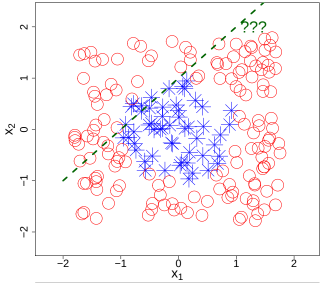


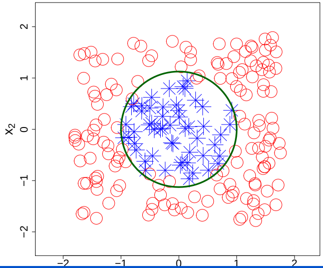
- *i.e.* separating data by lines/hyperplanes
- in reality, we do know what f should be; plane or a more general surface

Nonlinear classification data

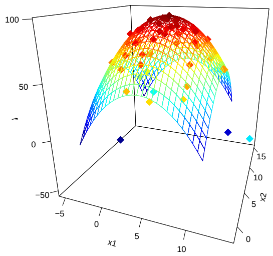
What if your data looks like this ?

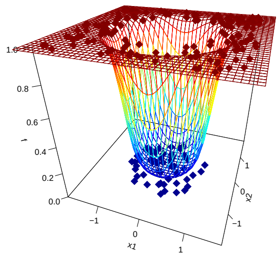
- no linear descision boudary or $f(\boldsymbol{x};\boldsymbol{\theta}) = \boldsymbol{\theta}^T\boldsymbol{x}$ seems making much sense
- but a non-linear boundary makes more sense
  - the classification rule is actually $||\boldsymbol{x}||_2^2 = x_1^2 + x_2^2 \leq 1$
  - distance to $\mathbf{0}$ is less than 1
  - the boundary is a circle
  - I know it because I generated the data





Nonlinear model: polynomial model





Both models are actually 2nd order polynomial:

$$f(\boldsymbol{x};\boldsymbol{\beta}) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + \beta_4 x_1^2 + \beta_5 x_2^2 = \sum_{k=0}^{5} \beta_j \phi_j(\boldsymbol{x}) = \boldsymbol{\beta}^T \boldsymbol{\phi}(\boldsymbol{x})$$

- where

$$\boldsymbol{\phi}(\boldsymbol{x}) = [\underbrace{1}_{\phi_0(\boldsymbol{x})}, \underbrace{x_1}_{\phi_1(\boldsymbol{x})}, \underbrace{x_2}_{\phi_2(\boldsymbol{x})}, \underbrace{x_1 x_2}_{\phi_3(\boldsymbol{x})}, \underbrace{x_1^2}_{\phi_4(\boldsymbol{x})}, \underbrace{x_2^2}_{\phi_5(\boldsymbol{x})}]^T$$

- it expands $\boldsymbol{x} = [1, x_1, x_2]^T$ to a larger vector

Nonlinear response from linear model

Note that you get a free nonlinear model by transforming the input $\boldsymbol{X}$

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} \end{bmatrix} \Rightarrow \boldsymbol{\Phi} = \begin{bmatrix} \phi(\boldsymbol{x}^{(1)}) \\ \phi(\boldsymbol{x}^{(2)}) \\ \vdots \\ \phi(\boldsymbol{x}^{(m)}) \end{bmatrix} = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & x_1^{(1)}x_2^{(1)} & (x_1^{(1)})^2 & (x_2^{(1)})^2 \\ 1 & x_1^{(2)} & x_2^{(2)} & x_1^{(2)}x_2^{(2)} & (x_1^{(2)})^2 & (x_2^{(2)})^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & x_1^{(m)}x_2^{(m)} & (x_1^{(m)})^2 & (x_2^{(m)})^2 \end{bmatrix}$$

- remember superscript (i) index data samples; and subscript index features
- $\boldsymbol{\Phi}$ is a $m \times 6$ matrix
- for higher order polynormial, the new design matrix just gets wider

The expanded model for regression is

$$\boldsymbol{y} = \boldsymbol{\Phi}\beta + \epsilon$$

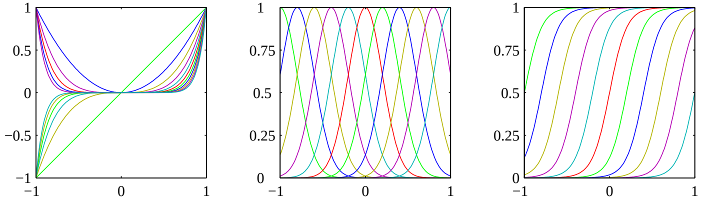
- still a linear model w.r.t ϕ, the expanded new features
- all existing results apply: gradient descent, normal equation (replace $\boldsymbol{X}$ with $\boldsymbol{\Phi}$)

$$\beta_{ls} = (\boldsymbol{\Phi}^T\boldsymbol{\Phi})^{-1}\boldsymbol{\Phi}^T\boldsymbol{y}$$

General basis function expansion

It turns out $\phi(\boldsymbol{x})$ can take a wide range of forms

- each $\phi_k(\boldsymbol{x})$ is a $R^n \rightarrow R$ transformation; so ϕ is a $R^n \rightarrow R^p$ transformation
    - previous example: $n = 3$, $p = 6$
- obviously, if $\phi(\boldsymbol{x}) = \boldsymbol{I}\boldsymbol{x}$, we recover ordinary linear regression
- previous case: $\phi_k(\boldsymbol{x}) = x_j^2,\ x_j x_{j'},\ x_j$, or 1, for 2-nd order polynormial
- 3-rd order polynomial will add $\phi_k(\boldsymbol{x}) = x_j^3,\ x_j x_{j'} x_{j''}$ on top
- $\phi_k(\boldsymbol{x}) = \log(x_j), \sqrt{x_j}, \sigma(x_j + \mu)$ (sigmoid function), ...
- radial basis function: $\phi_k(\boldsymbol{x}) = \exp\left(-\frac{||\boldsymbol{x}-\boldsymbol{\mu}_k||_2^2}{2s^2}\right)$



Fixed basis models

Regression function is assumed as

$$f(\boldsymbol{x}; \beta) = \sum_{k=0}^{p-1} \beta_k \phi_k(\boldsymbol{x})$$

- this is called fixed basis model in ML
  - it is called fixed basis because the basis have to be manually chosen for training
  - we will see adaptive basis model later (i.e. neural networks)
- we will assume linear models are just fixed basis models from now on

* A bit on abstract vector space: why called basis

In abstract vector space, a function can be a vector as well, say $\phi_k(\boldsymbol{x})$

$$f(\boldsymbol{x};\boldsymbol{\beta}) = \sum_{k=0}^{p-1} \beta_k \phi_k(\boldsymbol{x})$$

- f (our unknown regression function) is represented a linear combination of some basis vectors $\phi_0, \phi_1, \ldots$
- e.g.
$$f(x) = \beta_0 + \beta_1 x$$
can be viewed as a linear combination of two basis vectors: $\phi_0(x) = 1$ and $\phi_1(x) = x$
$$f(x) = \beta_0\phi_0(x) + \beta_1\phi_1(x)$$
- so for ordinary linear regression, we are finding the a representation of f only in the subspace span$(\{\phi_0(x), \phi_1(x)\})$
- something significant is all those things we learnt in linear algebra can still be applied (mostly) e.g. projection of f to a subspace

Suggested reading

- MLAPP 7.1-7.3; 8.1-8.3
- ESL 5.1-5.3
- Pattern recognition and ML by Chris Bishop 3.1, 4.3, *5.4 (Hessian matrix)
- *relevant ML research paper: Automatic Differentiation in Machine Learning: a Survey, Baydin et al, Journal of Machine Learning Research https://arxiv.org/pdf/1502.05767.pdf

Exercises for today's lecture

You can discuss your solution with me in extra lab session or with your classmates.

1. Find the Hessian for the linear regression's loss function by applying identities
2. Consider logistic regression's loss function (or negative log likelihood). Does the loss function has a maximum or minimum?
3. Why we cannot find the minimum for logistic regression in one step like linear regression ?

4. (Adapted from a past exam of ML at Cambridge and it practices your skill of applying MLE) To perform linear regression with noisy data, you find the normal assumption that the noise is zero mean Gaussian does not work well. Your colleague at work suggests you try *Cauchy density*

$$p(x; \alpha, \beta) = \frac{1}{\beta\pi}\left(\frac{\beta^2}{(x-\alpha)^2+\beta^2}\right)$$

(having parameters α and $\beta > 0$) instead.

- denote the parameters of the regression model by $\boldsymbol{\theta}$; given a set of m samples, each consisting of a n- dimensional vector $\boldsymbol{x}$ and corresponding target y. Find an expression for the log likleihood $p(\boldsymbol{y}|\boldsymbol{\theta})$ where $\boldsymbol{y}^T = (y_1, \ldots, y_m)$. State any assumption you make.
- derive a learning algorithm for the model; (try implementing it and see whether it works)
  - if you do not like doing derivatives manually, you should use autograd package instead (you at least need to know one way or the other!)
- is your colleague right ? why Cauchy makes sense when data is noisy ? (you may need to plot the density of a Cauchy and see its difference to Gaussian.)