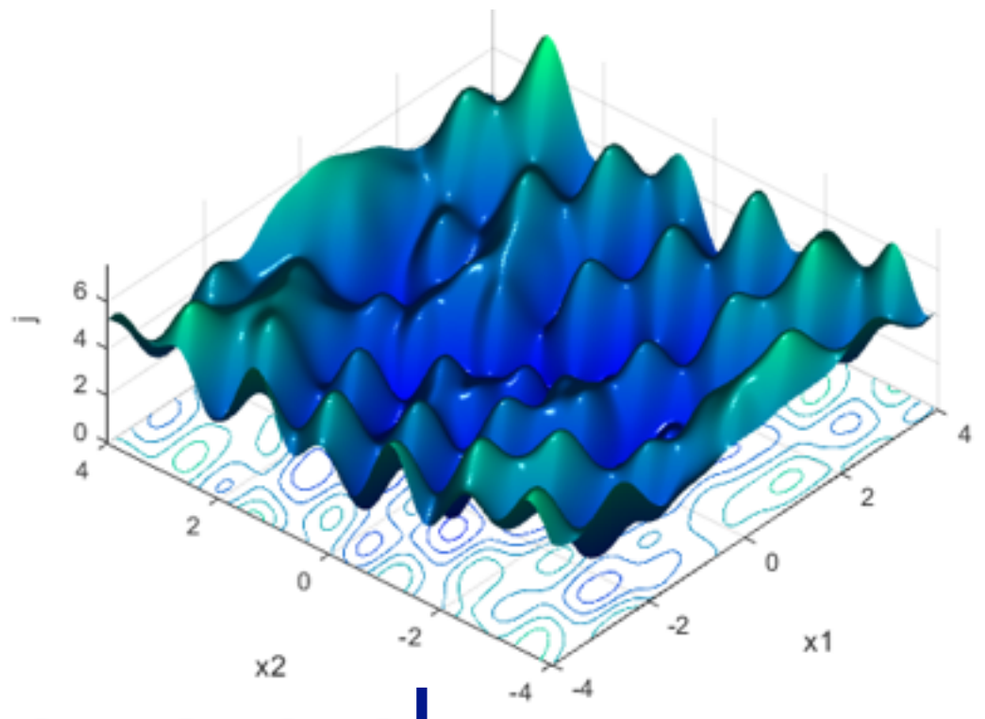
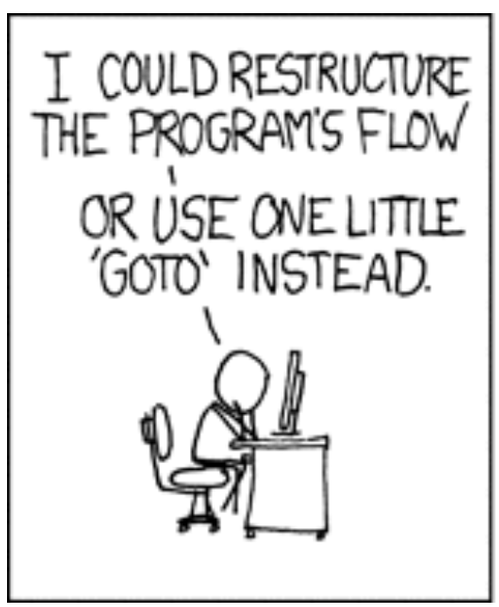
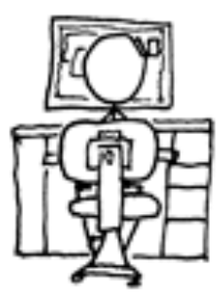
COMPUTER SCIENCE
INDIANA UNIVERSITY
School of Informatics and Computing
Bloomington




# Regularization and Optimization







# Reminders/Comments

- Thought question due today
- Assignment 1 marks should be done today
  - Write your name on the assignment to make it easier for TAs
- Updated notes for l1-regularization, to clarify the algorithm
  - You will implement this for Assignment 2
- Appendix in notes has a discussion on second-order optimization

# Bias-variance summary

$$\begin{aligned}
\mathrm{MSE}(\mathbf{w}, \boldsymbol{\omega}) &= \mathbb{E}[||\mathbf{w} - \boldsymbol{\omega}||_2^2] \\
&= \mathbb{E}[\sum_{j=1}^{d} (\mathbf{w}_j - \boldsymbol{\omega}_j)^2] \\
&= \sum_{j=1}^{d} \mathbb{E}[(\mathbf{w}_j - \boldsymbol{\omega}_j)^2] \\
&= \sum_{j=1}^{d} \mathrm{Bias}(\mathbf{w}_j, \boldsymbol{\omega}_j)^2 + \mathrm{Var}(\mathbf{w}_j)
\end{aligned}$$

$$\mathrm{Bias}(\mathbf{w}_j, \boldsymbol{\omega}_j) = \mathbb{E}[\mathbf{w}_j] - \boldsymbol{\omega}_j$$

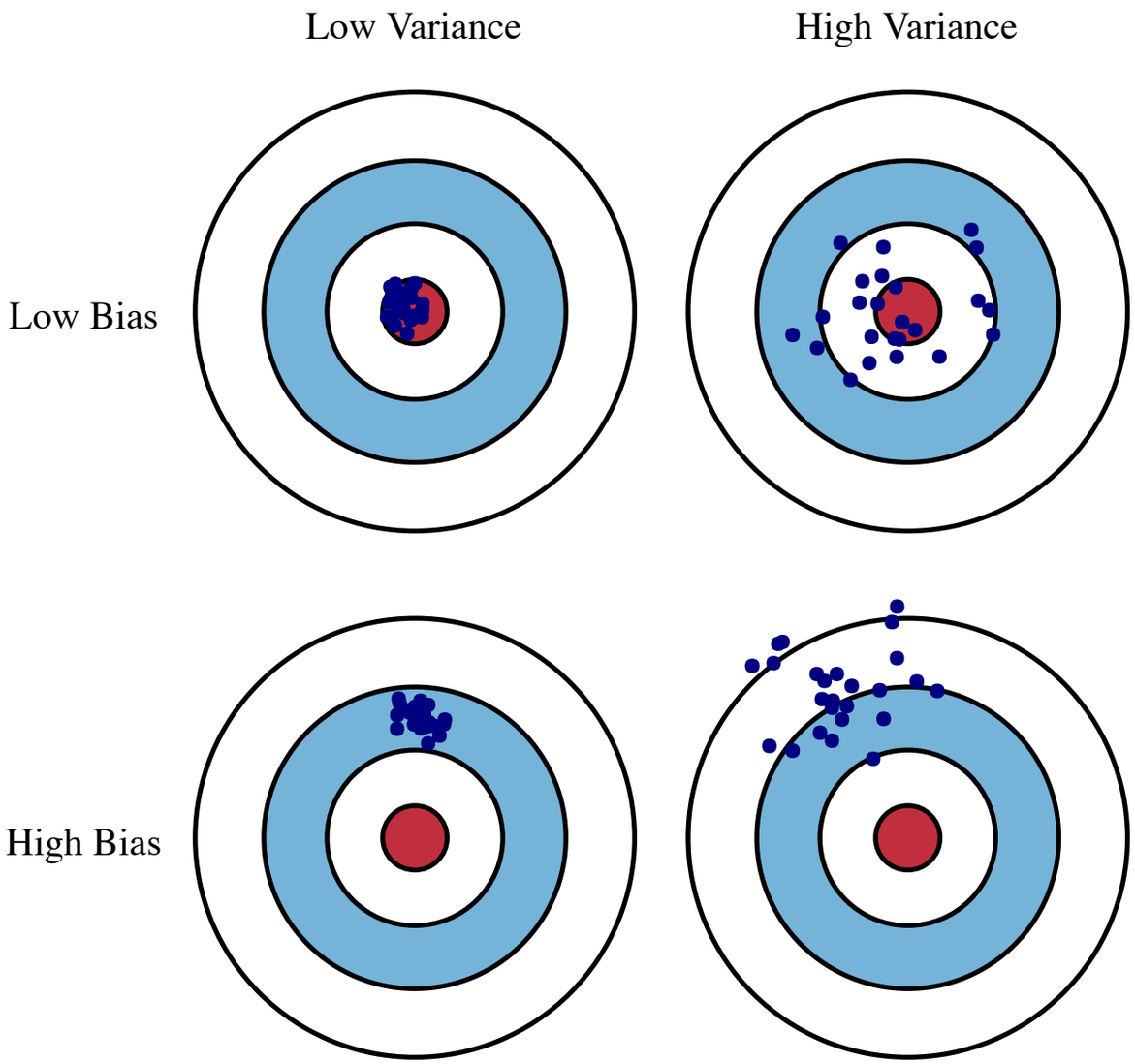
l2 regularization trades off bias and variance

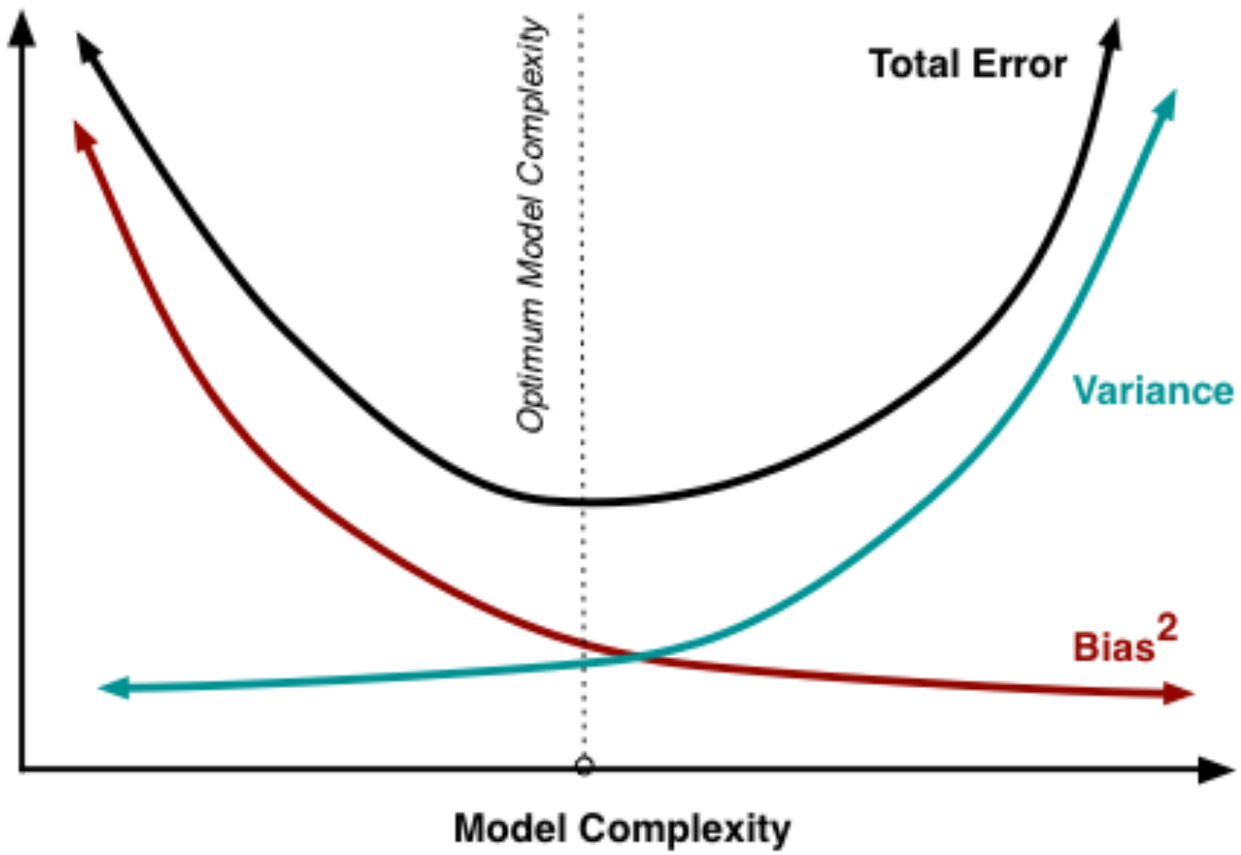
# Regularization and bias

- We picked a Gaussian prior and obtained l2 regularization
- We discussed the bias of this regularization
  - no regularization was unbiased E[w ] = true w
  - with regularization meant E[w ] was not equal to the true w

$$\mathbb{E}[(w \ - w_{\text{true}})^2] = \text{Bias}(w \ , w_{\text{true}})^2 + \text{Variance}(w \ )$$

# Bias-variance trade-off






*Nice images from: http://scott.fortmann-roe.com/docs/BiasVariance.html

# Thought question

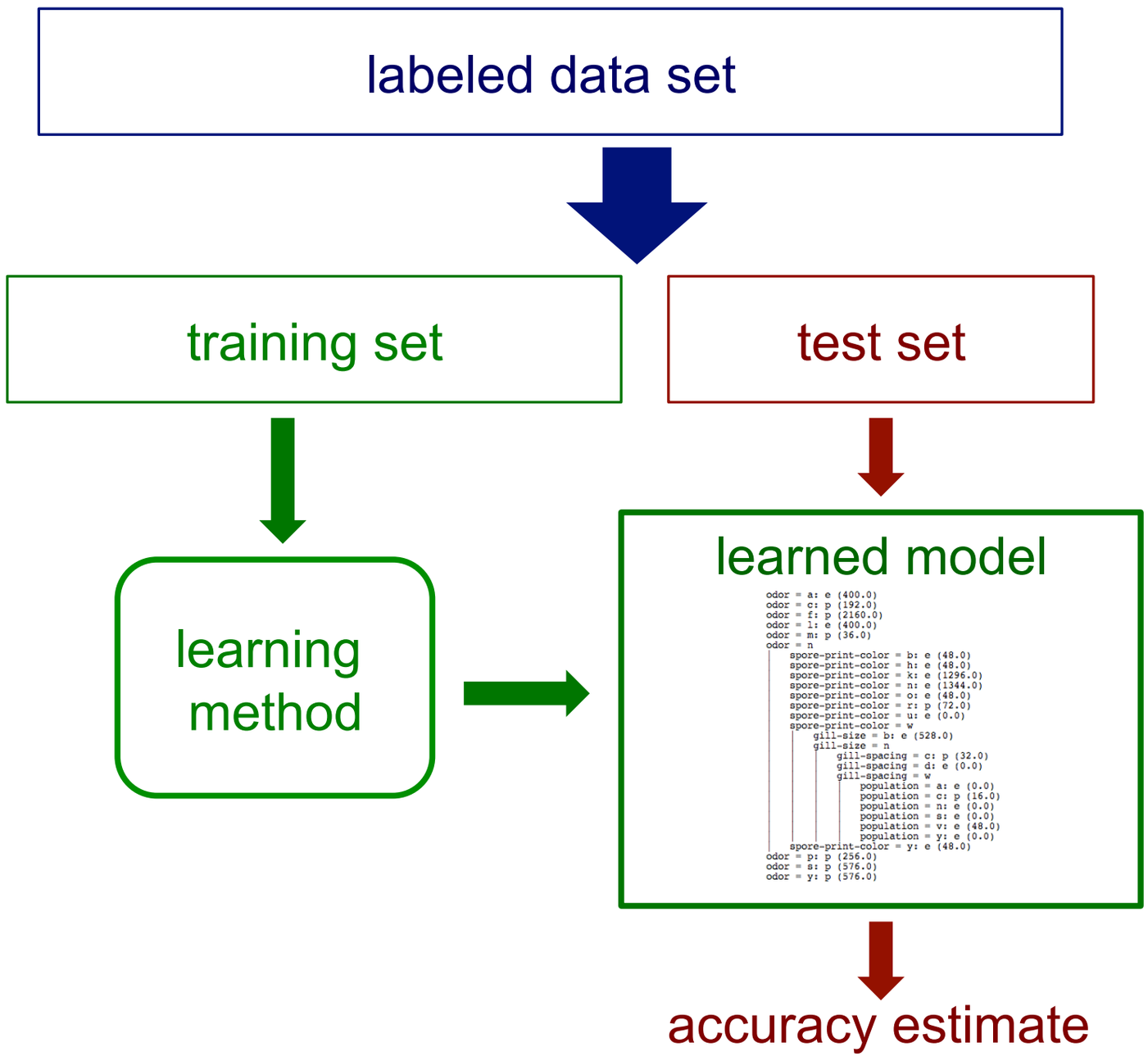
- How can we tell which model is a good model?

- We've discussed doing this empirically. But how?

# Measuring error

- What if you train many different models on a batch of data, check their accuracy on that data, and pick the best one?
  - Imagine your are predicting how much energy your appliances will use today
  - You train your models on all previous data for energy use in your home
  - How well will this perform in the real world?
- What if the models you are testing are only different in terms of the regularization parameter lambda that they use? What will you find?

# Simulating generalization error



# Simulating generalization error

- Now we have one model, trained similarly to how it will be trained, and a measure of accuracy on new data (but distributed identically to trained data)

- What if we pick the model with the best test accuracy?

# Exercise: Regularization and MAP

- Discussed that MAP and ML converge to same estimate
  - ML has lambda = 0
  - MAP has lambda > 0 indicating level of variance apriori assumed on the elements in w
- Does that happen here?

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

# Regularization intuition

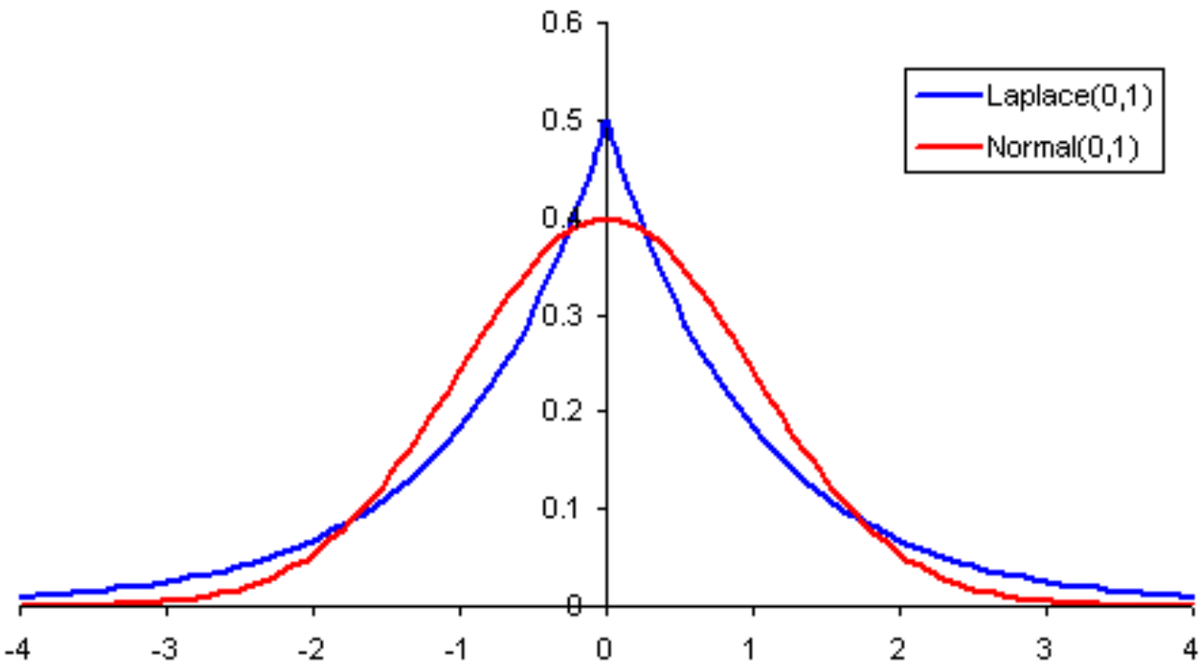
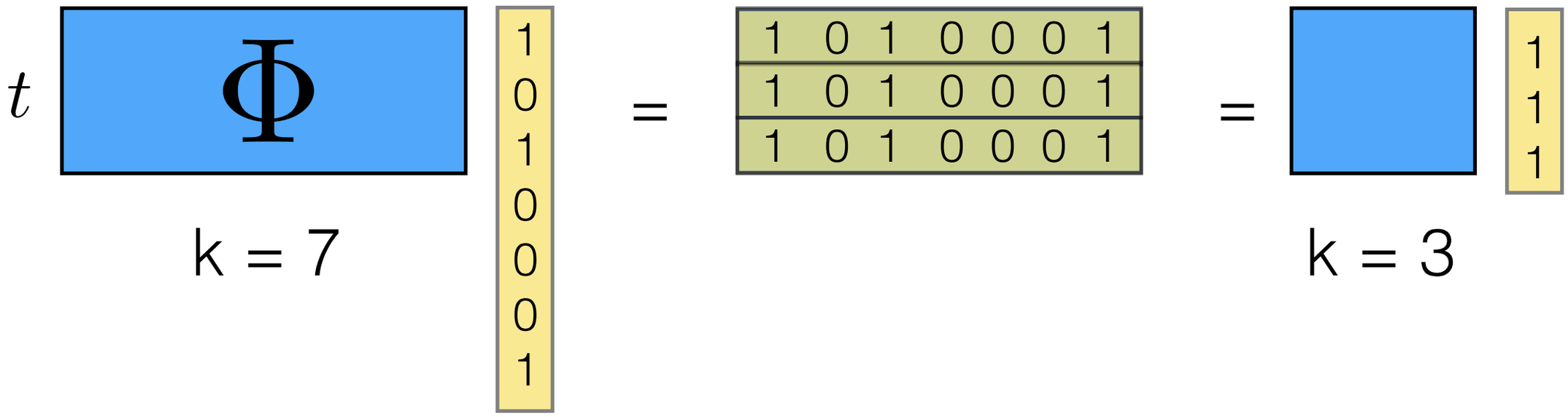



*Figure 4.5: A comparison between Gaussian and Laplace priors. The Gaussian prior prefers the values to be near zero, whereas the Laplace prior more strongly prefers the values to equal zero.*

# l1 regularization

- Feature selection, as well as preventing large weight




- How do we solve this optimization?

$$\min_{\mathbf{w} \in \mathbb{R}^d} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1$$

# Why should you care about the solution strategies?

- Understanding the optimization approaches behind the algorithms makes you more effectively choose which algorithm to run

- Understanding the optimization approaches makes you formalize your problem more effectively
  - otherwise you might formalize a very hard optimization problem; sometimes with minor modifications, can significantly simplify for the solvers, without impacting properties of solution significantly

- When you want to do something outside the given packages or solvers (which is often true)

- …also its fun!

# Where does gradient descent come from?

- Taylor series expansion with
  - First order for gradient second
  - Second order for Newton-Raphson method (also called second-order gradient descent)

# Gradient descent

**Algorithm 1:** Batch Gradient Descent(Err, $\mathbf{X}, \mathbf{y}$)

1: // A non-optimized, basic implementation of batch gradient descent
2: $\mathbf{w} \leftarrow$ random vector in $\mathbb{R}^d$
3: err $\leftarrow \infty$
4: tolerance $\leftarrow 10e^{-4}$
5: $\alpha \leftarrow 0.1$
6: **while** $|\text{Err}(\mathbf{w}) - \text{err}| > \text{tolerance}$ **do**
7: err $\leftarrow \text{Err}(\mathbf{w})$
8: // The step-size $\alpha$ should be chosen by line-search
9: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla \text{Err}(\mathbf{w}) = \mathbf{w} - \alpha \mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y})$
10: **return** $\mathbf{w}$

Recall: for error function $E(\mathbf{w})$ goal is to solve $\nabla E(\mathbf{w}) = \mathbf{0}$

# Taylor series expansion

A function $f(x)$ in the neighborhood of point $x_0$, can be approximated using the Taylor series as

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!}(x - x_0)^n,$$

where $f^{(n)}(x_0)$ is the $n$-th derivative of function $f(x)$ evaluated at point $x_0$.

e.g. $$f(x) \approx f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0).$$

# Multivariate Taylor series

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \cdot H_{f(\mathbf{x}_0)} \cdot (\mathbf{x} - \mathbf{x}_0),$$

$$\nabla f(\mathbf{x}) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_k} \right)$$

$$H_{f(\mathbf{x})} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_k} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & & \\ \vdots & & \ddots & \\ \frac{\partial^2 f}{\partial x_k \partial x_1} & & & \frac{\partial^2 f}{\partial x_k^2} \end{bmatrix}$$

# First and Second Order

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \cdot H_{f(\mathbf{x}_0)} \cdot (\mathbf{x} - \mathbf{x}_0),$$

- If we only have access to the function and the first derivate (gradient) and not the Hessian, then **first-order**
- If have Hessian, can use **second-order** techniques
  - Mostly removes the need for a step-size parameter, and/or makes the choice of this parameter much less sensitive
- For certain situations, computing the Hessian is too expensive (in space and computation) and so first-order methods are used OR quasi-second order (LBFGS)
  - e.g., huge number of features

# Second-order: Newton-Raphson for single-variate setting

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0).$$

$$f'(x) \approx f'(x_0) + (x - x_0)f''(x_0) = 0.$$

Solving this equation for $x$ gives us

$$x = x_0 - \frac{f'(x_0)}{f''(x_0)}.$$

Iterating gives us:

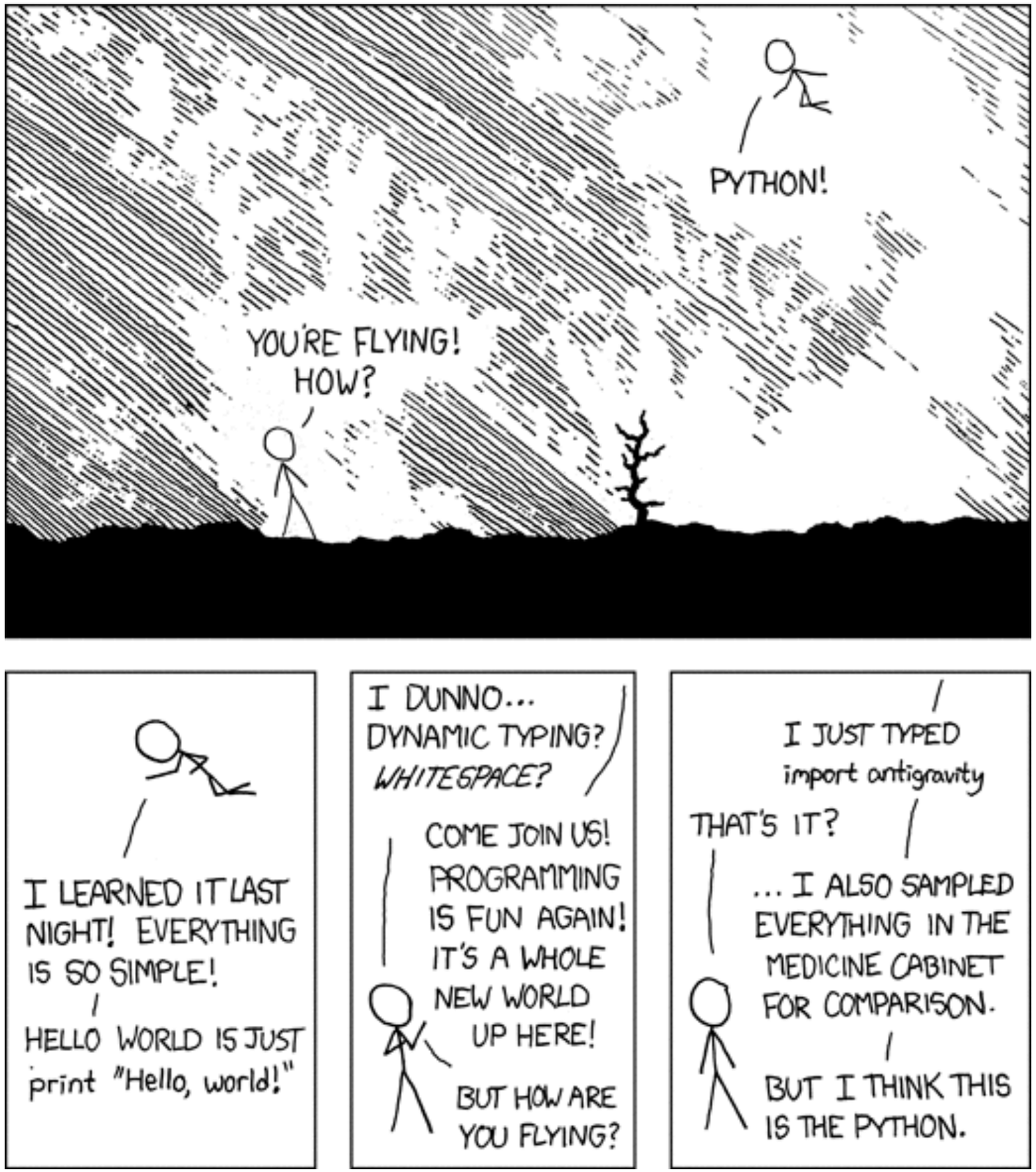
$$x^{(i+1)} = x^{(i)} - \frac{f'(x^{(i)})}{f''(x^{(i)})}.$$

# Second-order: Newton-Raphson for multi-variate setting

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T \cdot (\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} (\mathbf{x} - \mathbf{x}_0)^T \cdot H_{f(\mathbf{x}_0)} \cdot (\mathbf{x} - \mathbf{x}_0),$$

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} - \left(H_{f(\mathbf{x}^{(i)})}\right)^{-1} \cdot \nabla f(\mathbf{x}^{(i)}),$$

# Feb. 20


PYTHON!
YOU'RE FLYING!
HOW?
I LEARNED IT LAST NIGHT! EVERYTHING IS SO SIMPLE!
HELLO WORLD IS JUST print "Hello, world!"
I DUNNO... DYNAMIC TYPING? WHITESPACE?
COME JOIN US! PROGRAMMING IS FUN AGAIN! IT'S A WHOLE NEW WORLD UP HERE!
BUT HOW ARE YOU FLYING?
I JUST TYPED import antigravity
THAT'S IT?
... I ALSO SAMPLED EVERYTHING IN THE MEDICINE CABINET FOR COMPARISON.
BUT I THINK THIS IS THE PYTHON.

# Python demo

- Code uses object-oriented programming
- Main script_regression.py runs algorithms on a dataset
  - loads data
  - splits data
  - gather errors from running algorithms
- regressionalgorithms.py contains algorithm code
  - parent class Regressor
  - all regression algorithms are child classes
- utils.py contains some useful additional functions

# Thought exercise

- Why do we care about line search? What does it gain us?
- Why do we care about first-order gradient descent versus second-order gradient descent (Newton-Rhapson)?
- Additional possible questions you have
  - Why do we want to find an optimal point, w, with minimum loss(w)?
  - I don’t understand why gradient descent converges to a point with gradient = 0, or I don’t understand why we want that
  - Why do we use gradient descent? Can we search the space of values some other way (like a binary search)?
  - Does the amount of data influence whether we obtain local or global solutions?

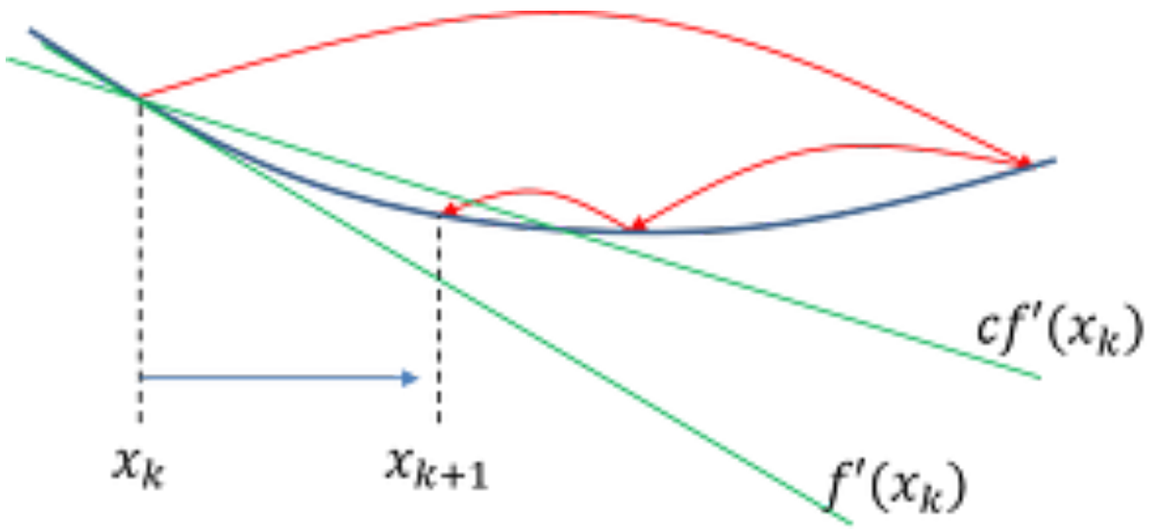
# Line search

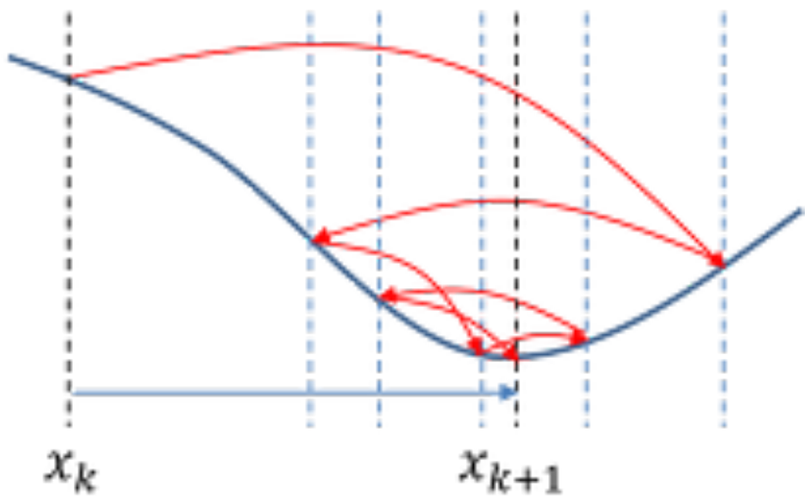
Want step-size such that

$$\alpha = \arg\min_{\alpha} E(\mathbf{w} - \alpha \nabla E(\mathbf{w}))$$

Backtracking line search:

1. Start with relatively large $\alpha$ (say $\alpha = 1$)
2. Check if $E(\mathbf{w} - \alpha \nabla E(\mathbf{w})) < E(\mathbf{w})$
3. If yes, use that $\alpha$
4. Otherwise, decrease $\alpha$ (e.g., $\alpha = \alpha/2$), and check again

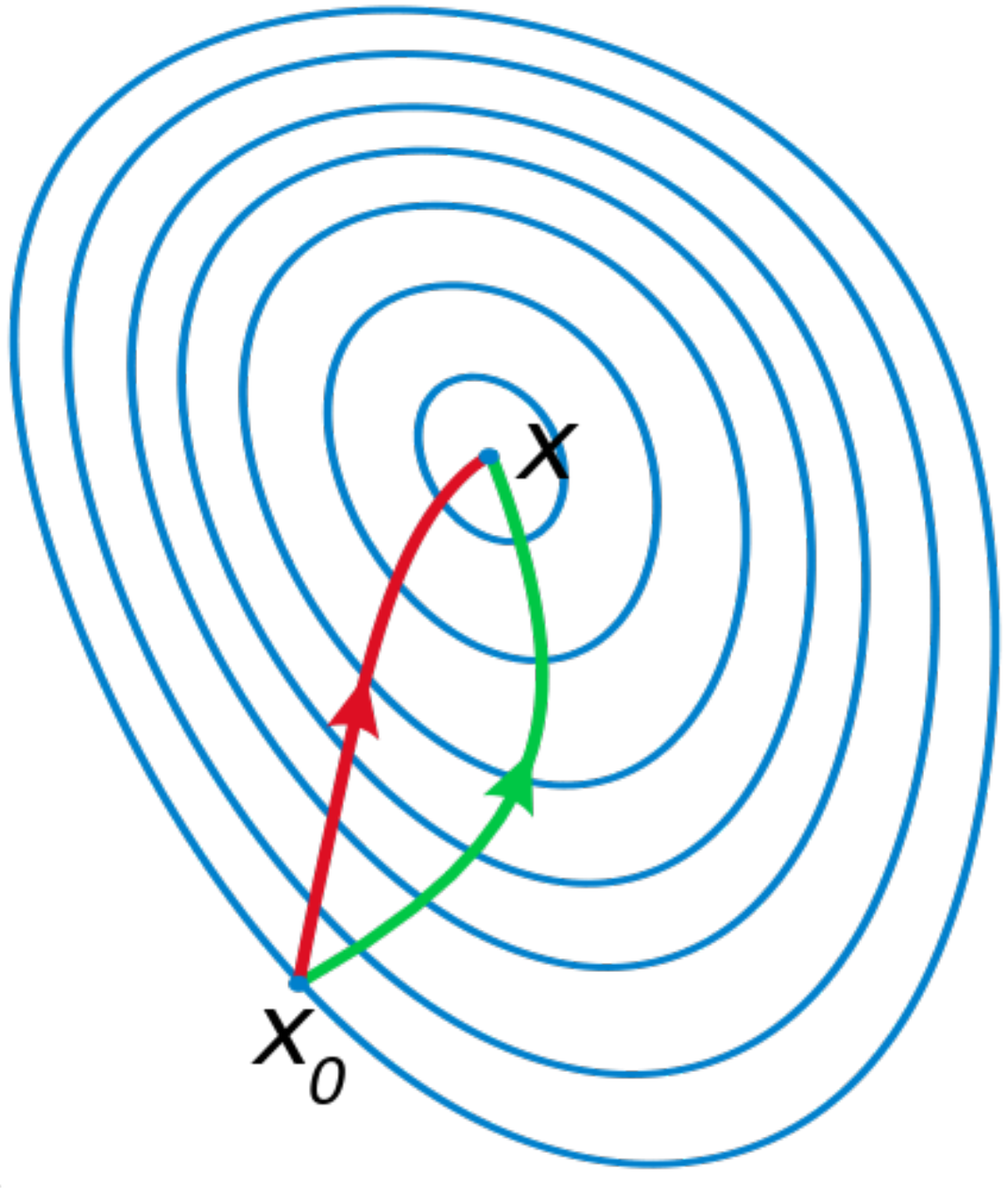


# Intuition for first and second order

- Locally approximate function at current point
- For first order, locally approximate as linear and step in the direction of the minimum of that linear function
- For second order, locally approximate as quadratic and step in the direction of the minimum of that quadratic function
  - a quadratic approximation is more accurate
- What happens if the true function is quadratic?

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} - \left(H_{f(\mathbf{x}^{(i)})}\right)^{-1} \cdot \nabla f(\mathbf{x}^{(i)}),$$




Newton in red

# Batch gradient descent

- We obtain a sample average of the true gradient using a batch of data

$$\text{For batch error:} \quad \hat{E}(\mathbf{w}) = \sum_{t=1}^{n} E_t(\mathbf{w})$$
$$\text{e.g.,} \quad E_t(\mathbf{w}) = (\mathbf{x}_t^\top \mathbf{w} - y_t)^2$$

# Unbiased gradient

$$\mathbb{E}\left[\frac{1}{n}\nabla\hat{E}(\mathbf{w})\right] = \frac{1}{n}\mathbb{E}\left[\sum_{i=1}^{n}\nabla E_i(\mathbf{w})\right]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[\nabla E_i(\mathbf{w})]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\mathbb{E}[\nabla E(\mathbf{w})]$$

$$= \frac{1}{n}\sum_{i=1}^{n}\nabla E(\mathbf{w})$$

$$= \nabla E(\mathbf{w})$$

# Stochastic gradient descent

- Stochastic gradient descent (stochastic approximation) minimizes with only one instance, which still gives an unbiased sample of the gradient

$$\mathbb{E}[\nabla E_t(\mathbf{w})] = \nabla E(\mathbf{w})$$

**Algorithm 2:** Stochastic Gradient Descent$(E, \mathbf{X}, \mathbf{y})$

1: $\mathbf{w} \leftarrow$ random vector in $\mathbb{R}^d$
2: **for** $t = 1, \ldots, n$ **do**
3: // For some settings, we need the step-size $\alpha_t$ to decrease with time
4: $\mathbf{w} \leftarrow \mathbf{w} - \alpha_t \nabla E_t(\mathbf{w}) = \mathbf{w} - \alpha_t(\mathbf{x}_t^\top \mathbf{w} - y_t)\mathbf{x}_t$
5: **end for**
6: **return** $\mathbf{w}$

# Stochastic gradient descent

- Can also approximate gradient with more than one sample (e.g., mini-batch), as long as

$$\mathbb{E}[\nabla E_t(\mathbf{w})] = \nabla E(\mathbf{w})$$

- Proof of convergence and conditions on step-size: Robbins-Monro ("A Stochastic Approximation Method", Robbins and Monro, 1951)

- A big focus in recent years in the machine learning community; many new approaches for improving convergence rate, reducing variance, etc.
  - adadelta, adagrad, adam for neural networks