Week 7, Lecture 14 - Autodiff and Clustering Redux

Aaron Meyer

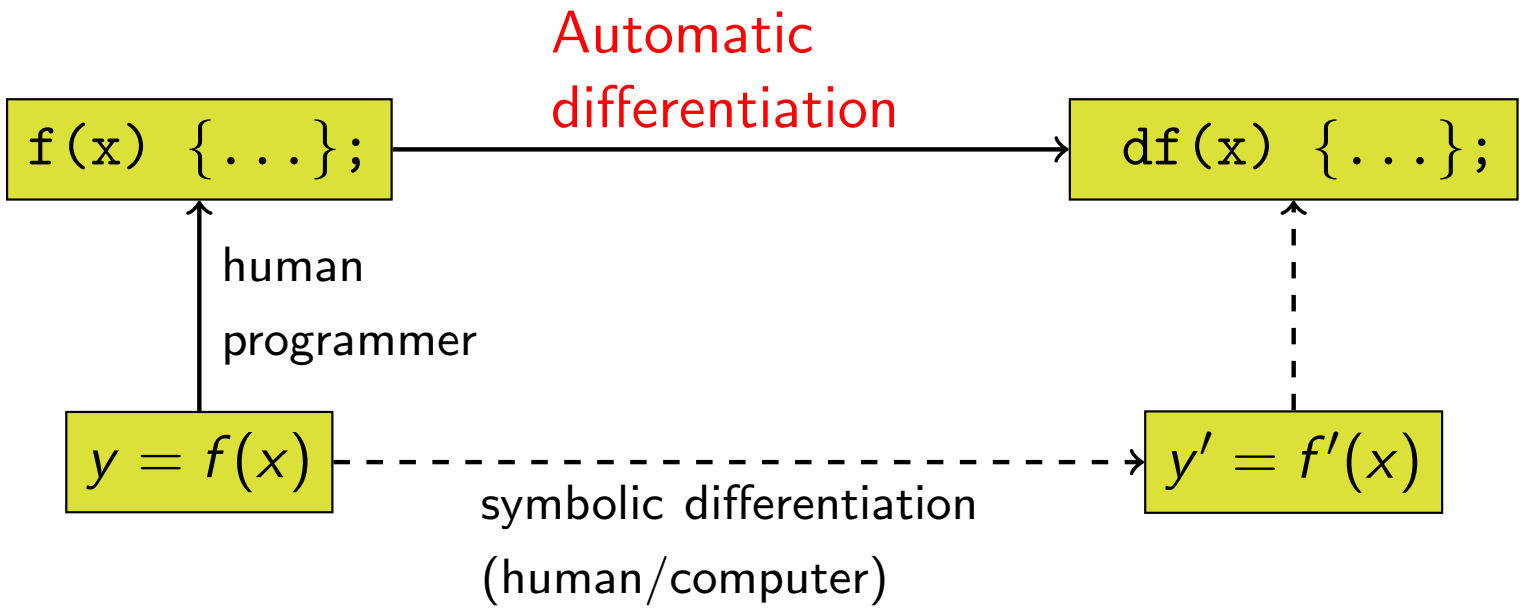
Outline

- Administrative Issues
- Autodifferentiation
- Gaussian mixtures
- Implementation

Based on slides from Håvard Berland and David Sontag.

What is automatic differentiation?

Automatic differentiation (AD) is software to transform code for one function into code for the derivative of the function.



Why automatic differentiation?

Scientific code often uses both functions *and* their derivatives:

- E.g. example Newtons method for solving (nonlinear) equations
- find x such that $f(x) = 0$

The Newton iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

But how to compute $f'(x_n)$ when we only know $f(x)$?

- Symbolic differentiation?
- Divided difference?
- Something else? **Yes!**

Divided differences

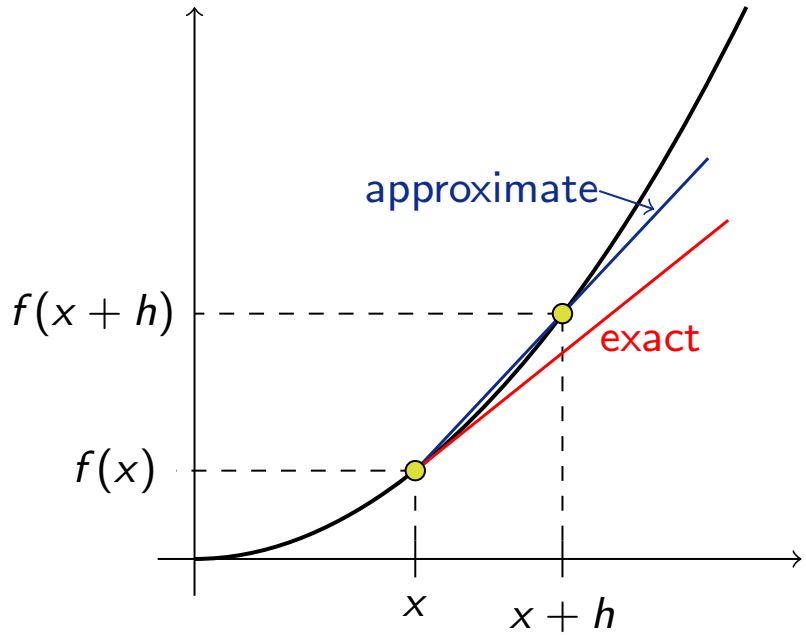
By definition, the derivative is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

so why not use

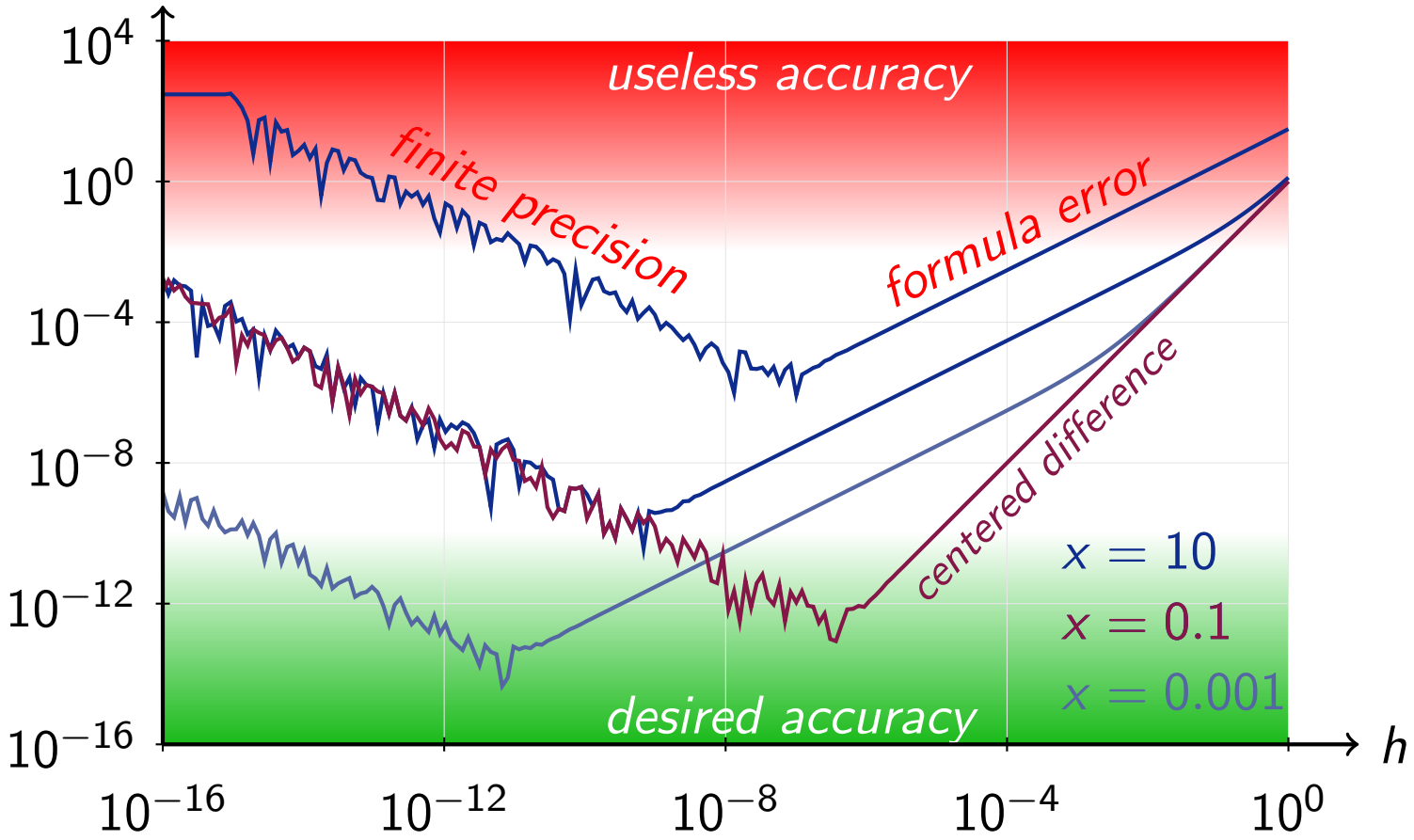
$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

for some appropriately small h?



Accuracy for divided differences on $f(x) = x^3$

$$\text{error} = \left| \frac{f(x+h) - f(x)}{h} - 3x^2 \right|$$



- Automatic differentiation will ensure desired accuracy.

Dual numbers

Extend all numbers by adding a second component,

$$x \mapsto x + \dot{x}\mathbf{d}$$

- $\mathbf{d}$ is just a symbol distinguishing the second component,
- analogous to the imaginary unit $\mathbf{i} = \sqrt{-1}$.
- But, let $\mathbf{d}^2 = 0$, as opposed to $\mathbf{i}^2 = -1$.

Arithmetic on dual numbers:

$$(x + \dot{x}\mathbf{d}) + (y + \dot{y}\mathbf{d}) = x + y + (\dot{x} + \dot{y})\mathbf{d}$$

$$\begin{aligned}(x + \dot{x}\mathbf{d}) \cdot (y + \dot{y}\mathbf{d}) &= xy + x\dot{y}\mathbf{d} + \dot{x}y\mathbf{d} + \overbrace{\dot{x}\dot{y}\mathbf{d}^2}^{=0} \\ &= xy + (x\dot{y} + \dot{x}y)\mathbf{d}\end{aligned}$$

$$-(x + \dot{x}\mathbf{d}) = -x - \dot{x}\mathbf{d}, \qquad \frac{1}{x + \dot{x}\mathbf{d}} = \frac{1}{x} - \frac{\dot{x}}{x^2}\mathbf{d} \quad (x \neq 0)$$

Polynomials over dual numbers

Let

$$P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$$

and extend x to a dual number $x + \dot{x}\mathbf{d}$.
Then,

$$\begin{aligned} P(x + \dot{x}\mathbf{d}) &= p_0 + p_1(x + \dot{x}\mathbf{d}) + \cdots + p_n(x + \dot{x}\mathbf{d})^n \\ &= p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n \\ &\quad + p_1 \dot{x}\mathbf{d} + 2p_2 x \dot{x}\mathbf{d} + \cdots + n p_n x^{n-1} \dot{x}\mathbf{d} \\ &= P(x) + P'(x)\dot{x}\mathbf{d} \end{aligned}$$

- $\dot{x}$ may be chosen arbitrarily, so choose $\dot{x} = 1$ (currently).
- *The second component is the derivative of $P(x)$ at x*

Functions over dual numbers

Similarly, one may derive

$$\begin{aligned}
\sin(x + \dot{x}\mathbf{d}) &= \sin(x) + \cos(x)\,\dot{x}\mathbf{d} \\
\cos(x + \dot{x}\mathbf{d}) &= \cos(x) - \sin(x)\,\dot{x}\mathbf{d} \\
e^{(x+\dot{x}\mathbf{d})} &= e^{x} + e^{x}\dot{x}\mathbf{d} \\
\log(x + \dot{x}\mathbf{d}) &= \log(x) + \frac{\dot{x}}{x}\mathbf{d} \quad x \neq 0 \\
\sqrt{x + \dot{x}\mathbf{d}} &= \sqrt{x} + \frac{\dot{x}}{2\sqrt{x}}\mathbf{d} \quad x \neq 0
\end{aligned}$$

Conclusion from dual numbers

- Derived from dual numbers:
  - A function applied on a dual number will return its derivative in the second/dual component.
- We can extend to functions of many variables by introducing more dual components:

$$f(x_1, x_2) = x_1 x_2 + \sin(x_1)$$

extends to:

$$f(x_1 + \dot{x}_1 \mathbf{d}_1, x_2 + \dot{x}_2 \mathbf{d}_2) =$$
$$(x_1 + \dot{x}_1 \mathbf{d}_1)(x_2 + \dot{x}_2 \mathbf{d}_2) + \sin(x_1 + \dot{x}_1 \mathbf{d}_1) =$$
$$x_1 x_2 + (x_2 + \cos(x_1))\dot{x}_1 \mathbf{d}_1 + x_1 \dot{x}_2 \mathbf{d}_2$$

where $d_i d_j = 0$.

Decomposition of functions, the chain rule

Computer code for $f(x_1, x_2) = x_1 x_2 + \sin(x_1)$ might read

Original program

$w_1 = x_1$
$w_2 = x_2$
$w_3 = w_1 w_2$
$w_4 = \sin(w_1)$
$w_5 = w_3 + w_4$

Dual program

$\dot{w}_1 = 0$
$\dot{w}_2 = 1$
$\dot{w}_3 = \dot{w}_1 w_2 + w_1 \dot{w}_2 = 0 \cdot x_2 + x_1 \cdot 1 = x_1$
$\dot{w}_4 = \cos(w_1)\dot{w}_1 = \cos(x_1) \cdot 0 = 0$
$\dot{w}_5 = \dot{w}_3 + \dot{w}_4 = x_1 + 0 = x_1$

and

$$\frac{\partial f}{\partial x_2} = x_1$$

The chain rule

$$\frac{\partial f}{\partial x_2} = \frac{\partial f}{\partial w_5}\frac{\partial w_5}{\partial w_3}\frac{\partial w_3}{\partial w_2}\frac{\partial w_2}{\partial x_2}$$

ensures that we can *propagate* the dual components throughout the computation.

Realization of automatic differentiation

Our current procedure:

1. Decompose original code into intrinsic functions
2. Differentiate the intrinsic functions, effectively symbolically
3. Multiply together according to the chain rule

How to "automatically" transform "original program" to "dual program"?

Three approaches:

- Source code transformation
- Operator overloading
- Computation graph

Source code transformation by example

function.c

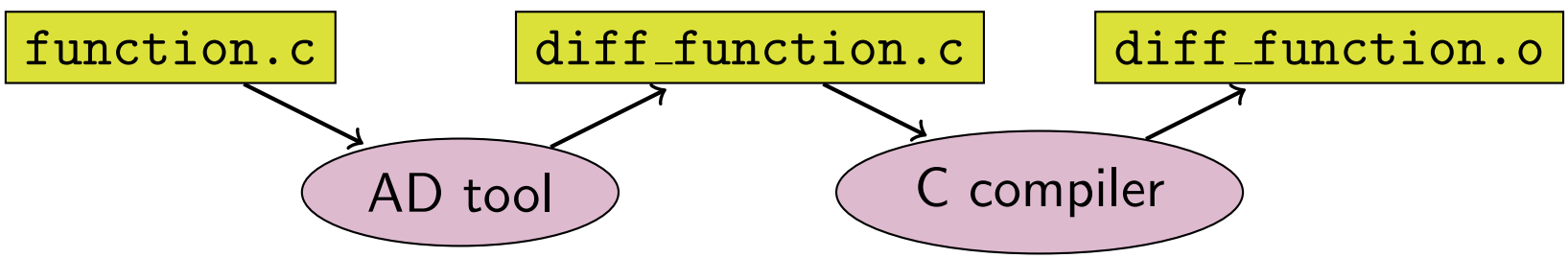
```
double  f(double x1, double x2) {
  double w3, w4, w5;
  w3 = x1 * x2;

  w4 = sin(x1);

  w5 = w3 + w4;



  return w5;
}
```

function.c

Source code transformation by example

diff_function.c

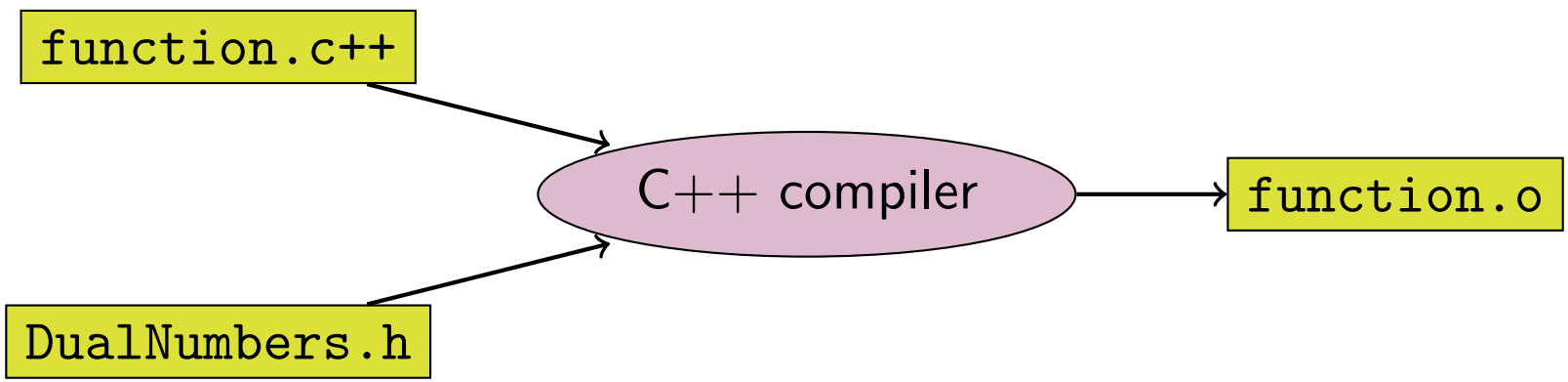
```
double* f(double x1, double x2, double dx1, double dx2) {
  double w3, w4, w5, dw3, dw4, dw5, df[2];
  w3 = x1 * x2;
  dw3 = dx1 * x2 + x1 * dx2;
  w4 = sin(x1);
  dw4 = cos(x1) * dx1;
  w5 = w3 + w4;
  dw5 = dw3 + dw4;
  df[0] = w5;
  df[1] = dw5;
  return df;
}
```



Operator overloading

function.c++

```
Number f(Number x1, Number x2) {
  w3 = x1 * x2;
  w4 = sin(x1);
  w5 = w3 + w4;
  return w5;
}
```



Source transformation vs. operator overloading

Source code transformation:

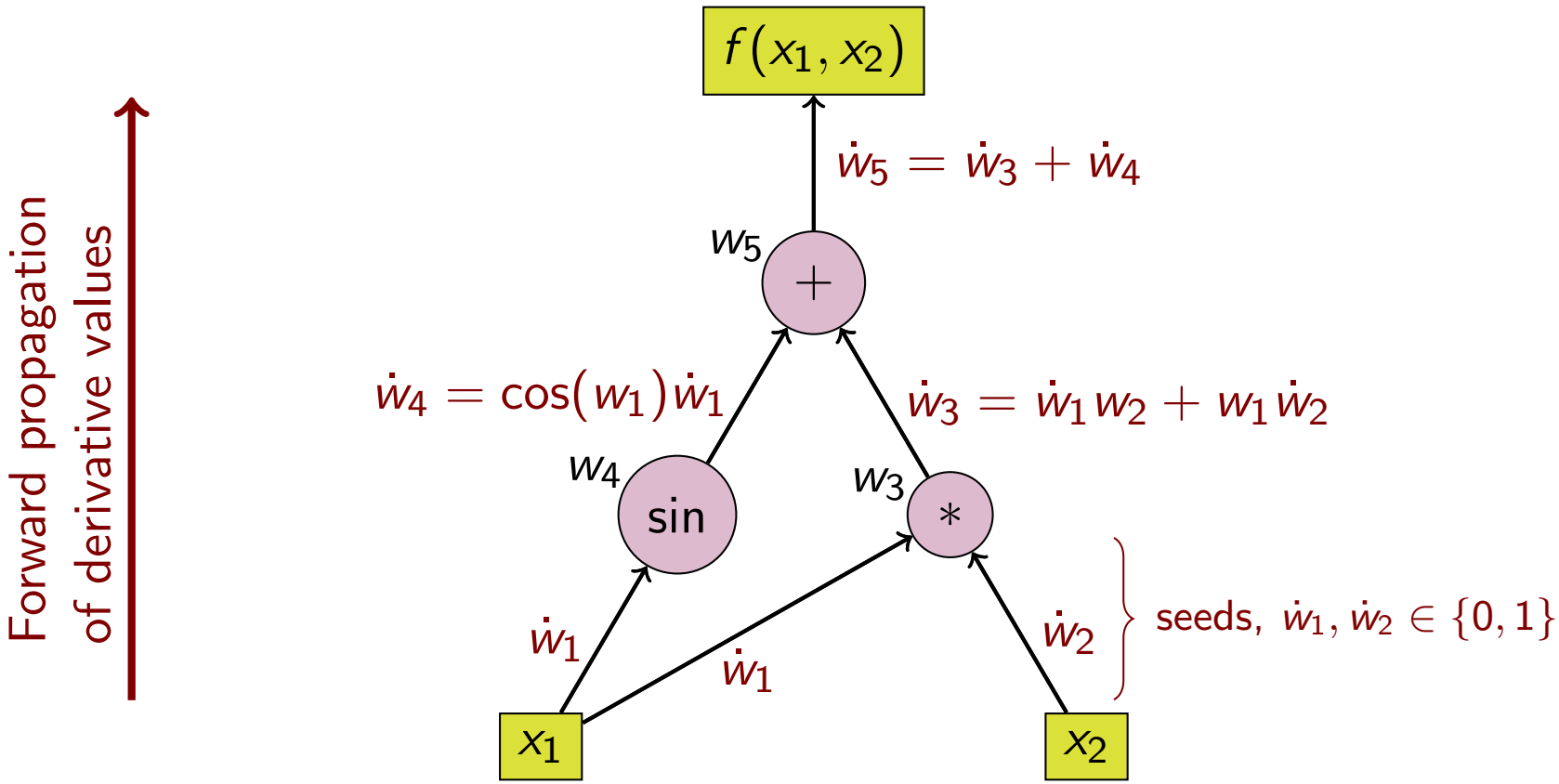
- Possible in all computer languages
- Can be applied to your old legacy Fortran/C code. Allows easier compile time optimizations.
- Source code swell
- More difficult to code the AD tool

Operator overloading:

- No changes in your original code
- Flexible when you change your code or tool Easy to code the AD tool
- Only possible in selected languages
- Current compilers lag behind, code runs slower

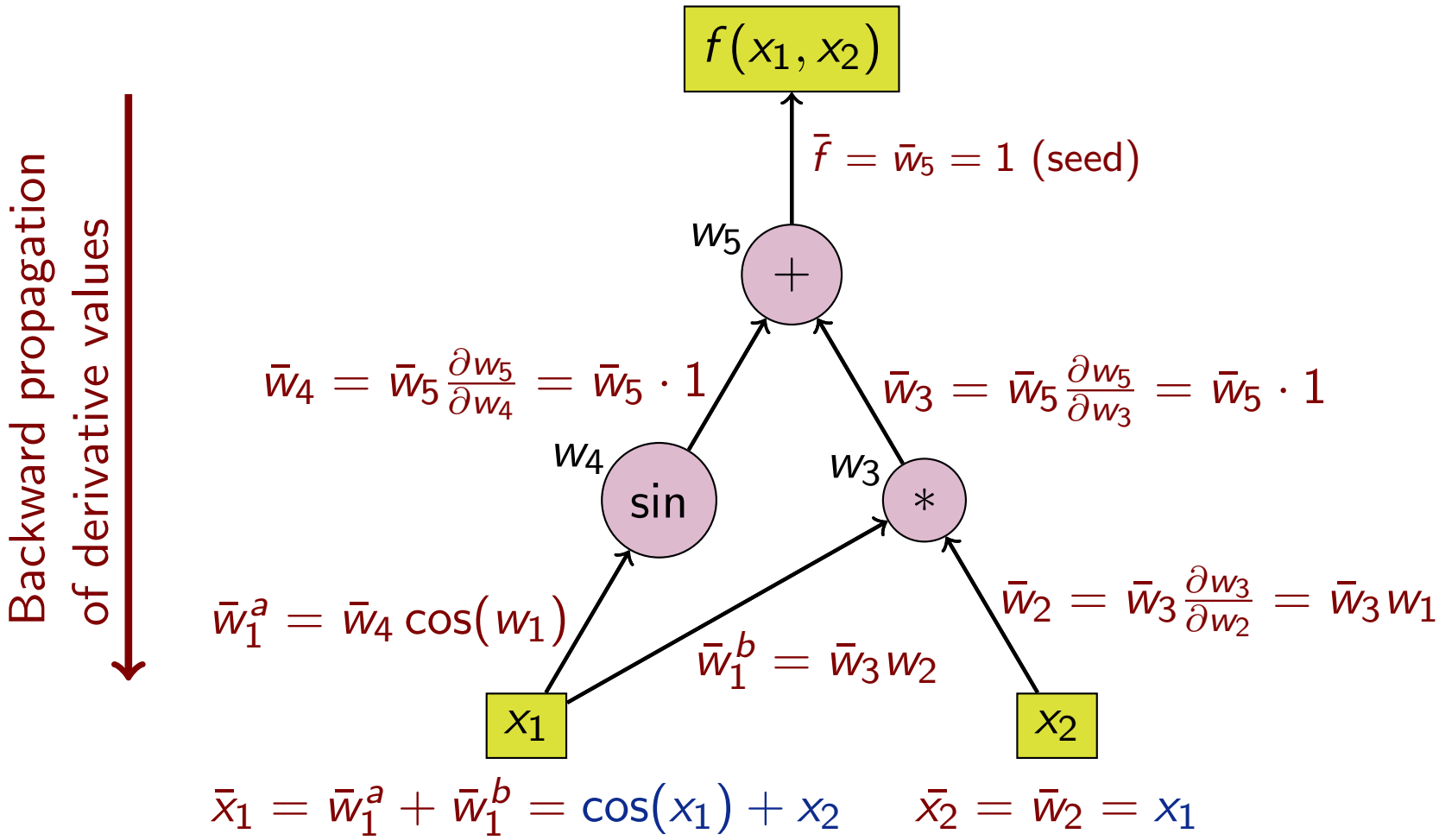
Forward mode AD

- We have until now only described forward mode AD.
- Repetition of the procedure using the computational graph:



Reverse mode AD

- The chain rule works in both directions.
- The computational graph is now traversed from the top.



Jacobian computation

Given $F : \mathbf{R}^n \mapsto \mathbf{R}^m$ and the Jacobian $J = DF(\mathbf{x}) \in \mathbf{R}^{m \times n}$.

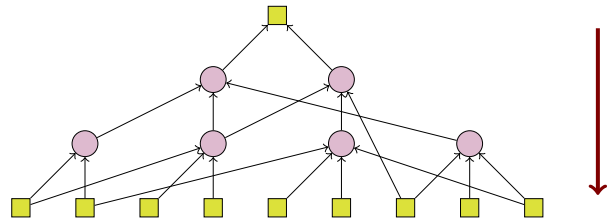
$$J = DF(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

- One sweep of *forward mode* can calculate one column vector of the Jacobian, $J\dot{\mathbf{x}}$, where $\dot{\mathbf{x}}$ is a column vector of seeds.
- One sweep of *reverse mode* can calculate one row vector of the Jacobian, $\bar{\mathbf{y}}J$, where $\bar{\mathbf{y}}$ is a row vector of seeds.
- Computational cost of one sweep forward or reverse is roughly equivalent, but reverse mode requires access to *intermediate* variables, requiring more memory.

Forward or reverse mode AD?

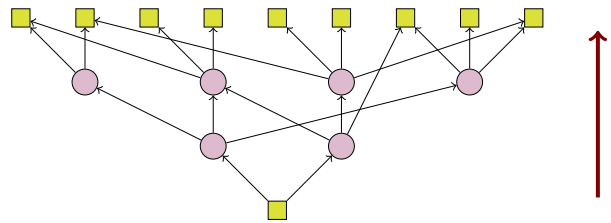
Reverse mode AD is best suited for

$$F : \mathbf{R}^n \to \mathbf{R}$$

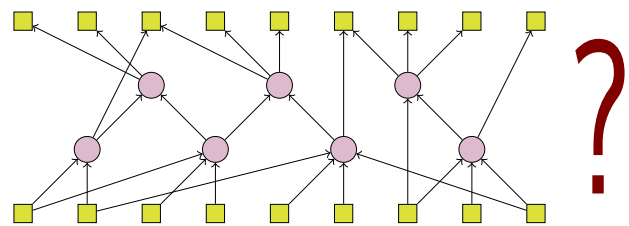


Forward mode AD is best suited for

$$G : \mathbf{R} \to \mathbf{R}^m$$



- Forward and reverse mode represents just two possible (extreme) ways of recursing through the chain rule.
- For $n > 1$ and $m > 1$ there is a golden mean, but finding the optimal way is probably an *NP*-hard problem.



Discussion

- Accuracy is guaranteed and complexity is not worse than that of the original function.
- AD works on iterative solvers, on functions consisting of thousands of lines of code.
- AD is trivially generalized to higher derivatives. Hessians are used in some optimization algorithms. Complexity is quadratic in highest derivative degree.
- The alternative to AD is usually symbolic differentiation, or rather using algorithms not relying on derivatives.
- Divided differences may be just as good as AD in cases where the underlying function is based on discrete or measured quantities, or being the result of stochastic simulations.

Implementation of AD

Implementation is quite specific to software package.

- tensorflow (python, forward/reverse mode, operator overloading)
- Theano (python, symbolic transformation, operator overloading)
- cppad (C++, forward/reverse mode, operator overloading)
- adapt (C++, forward/reverse mode, operator overloading)

Applications of AD

- Newton's method for solving nonlinear equations
- Optimization (utilizing gradients/Hessians)
- Inverse problems/data assimilation
- Neural networks
- Solving stiff ODEs

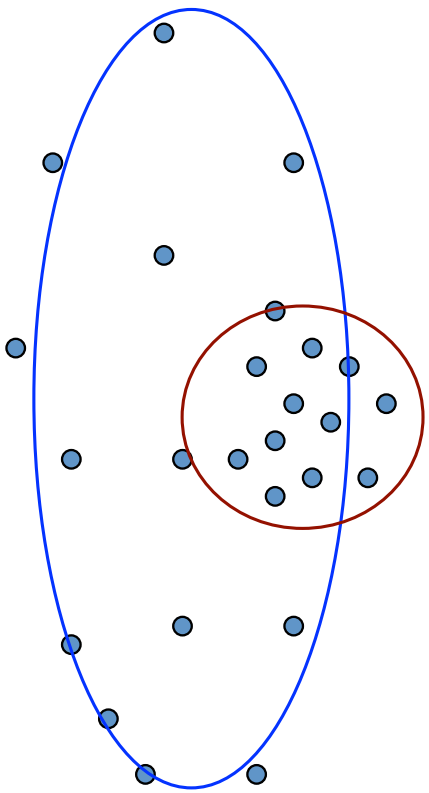
Recommended literature:

- Andreas Griewank: *Evaluating Derivatives.* SIAM 2000.

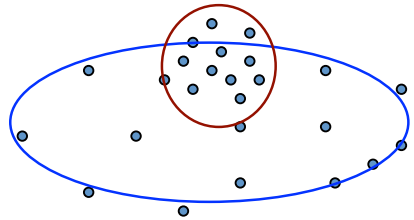
Gaussian mixtures

The Evils of "Hard Assignments"?

- Clusters may overlap
- Some clusters may be "wider" than others
- Distances can be deceiving!



Probabilistic Clustering



- Try a probabilistic model!
  - Allows overlaps, clusters of different size, etc.
- Can tell a *generative story* for data
  - $P(X \mid Y)P(Y)$
- Challenge: we need to estimate model parameters without labeled Ys

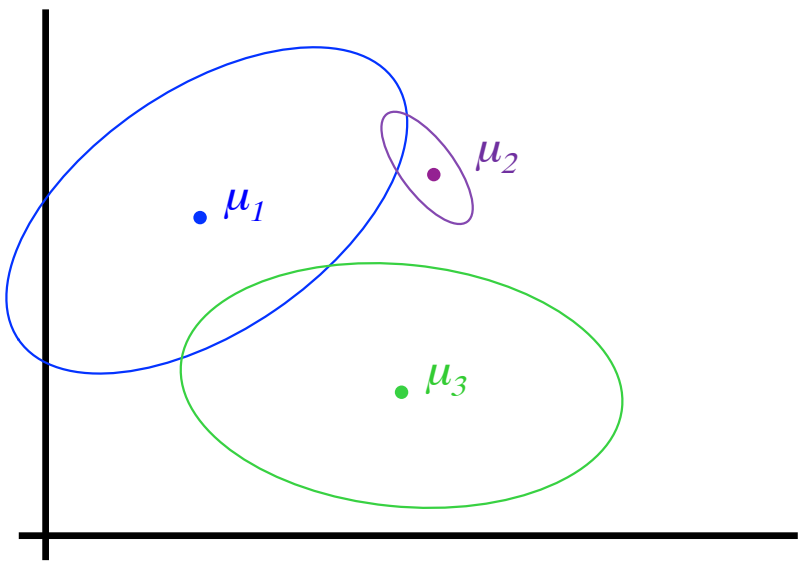
Y	X_1	X_2
??	0.1	2.1
??	0.5	-1.1
??	0.0	3.0
??	-0.1	-2.0
??	0.2	1.5
...	...	...

The General GMM assumption

- $P(Y)$: There are k components
- $P(X \mid Y)$: Each component generates data from a *multivariate Gaussian* with mean μ_i and covariance matrix Σ_i

Each data point is sampled from a **generative process**:

1. Choose component i with probability $P(y = i)$
2. Generate datapoint $N(m_i, \Sigma_i)$



What Model Should We Use?

- Depends on X.
- If we know which points are in a cluster, then we can define the best distribution for it.
  - Multinomial over clusters Y
  - (Independent) Gaussian for each X_i given Y

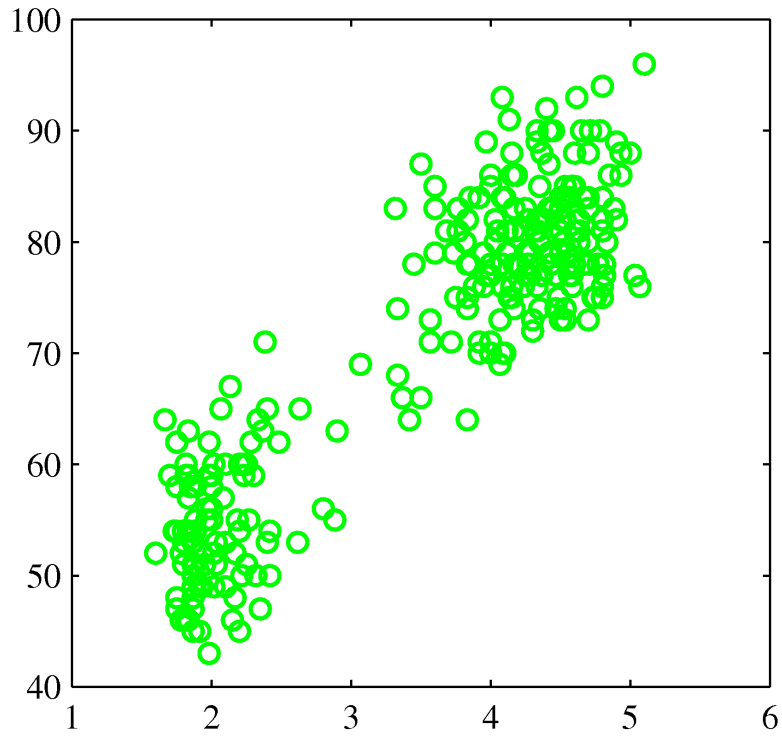
$$p(Y_i = y_k) = \theta_k$$

$$P(X_i = x \mid Y = y_k) = N(x \mid \mu_{ik}, \sigma_{ik})$$

Could we make fewer assumptions?

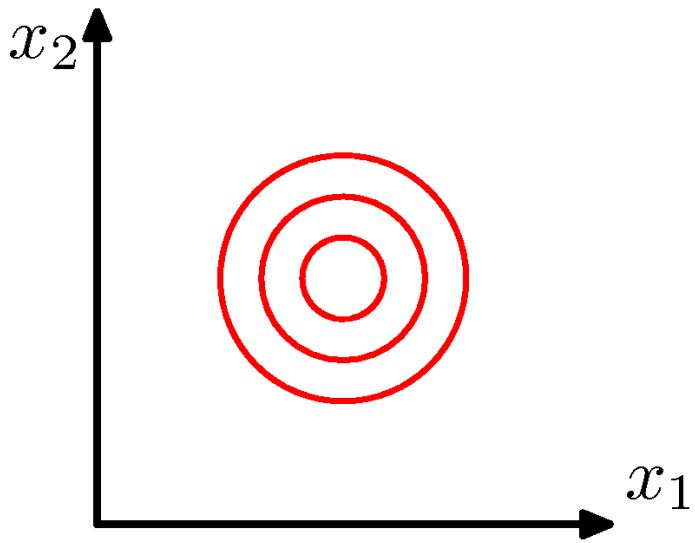
- What if the X_i co-vary?
- What if there are multiple peaks?
- **Gaussian Mixture Models!**
  - $P(Y)$ still multinormal
  - $P(\mathbf{X} \mid Y)$ is a *multivariate* Gaussian distribution:

$$P(X = x_j \mid Y = i) = N(x_j, \mu_i, \Sigma_i)$$



Multivariate Gaussians

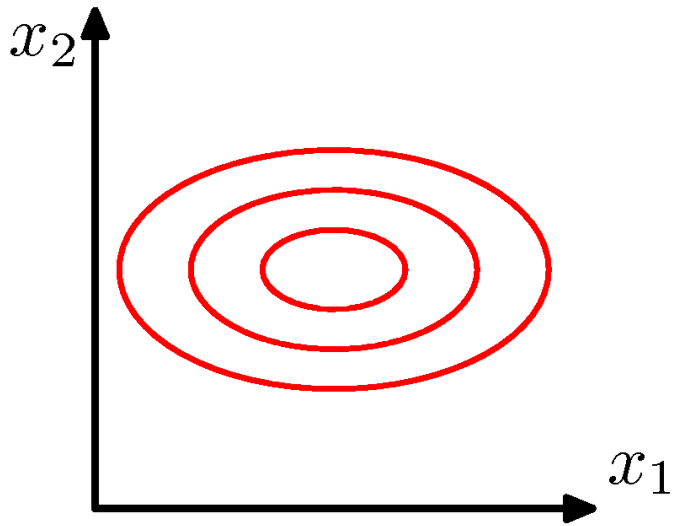
$$P(X=\mathbf{x}_j) = \frac{1}{(2\pi)^{m/2} \|\Sigma\|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_j - \mu)^T \Sigma^{-1} (\mathbf{x}_j - \mu)\right]$$



$\Sigma \propto$ identity matrix

Multivariate Gaussians

$$P(X=\mathbf{x}_j) = \frac{1}{(2\pi)^{m/2}\,\|\Sigma\|^{1/2}} \exp\left[-\frac{1}{2}\left(\mathbf{x}_j-\mu\right)^T \Sigma^{-1}\left(\mathbf{x}_j-\mu\right)\right]$$

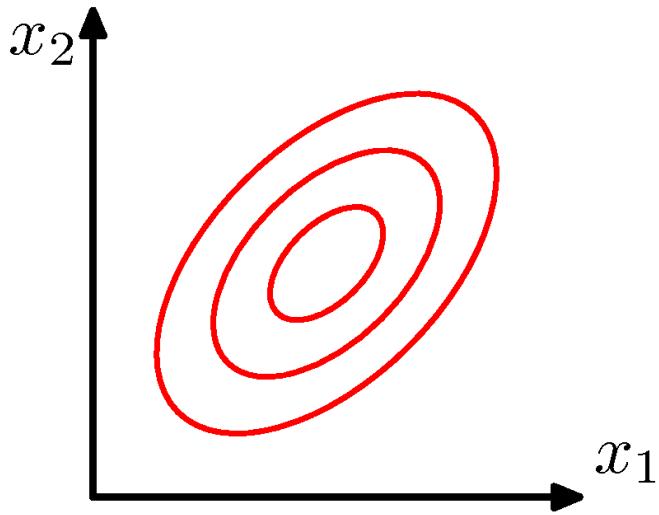


Σ = diagonal matrix

X_i are independent *ala* Gaussian NB

Multivariate Gaussians

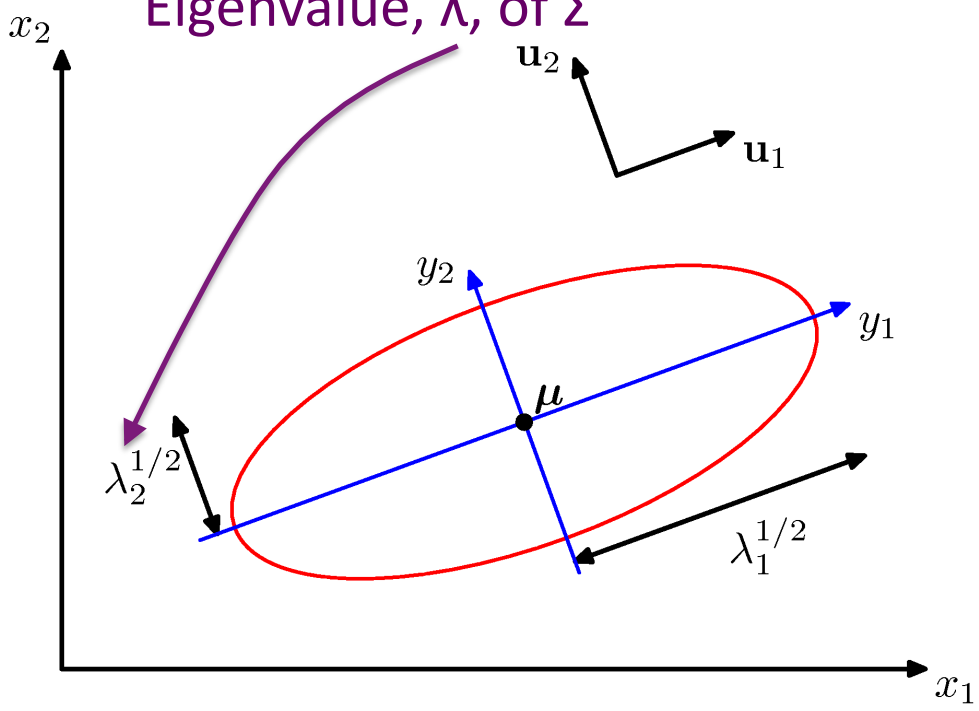
$$P(X=\mathbf{x}_j) = \frac{1}{(2\pi)^{m/2}\,\|\Sigma\|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_j-\mu)^T \Sigma^{-1}(\mathbf{x}_j-\mu)\right]$$



Σ = arbitrary (semidefinite) matrix:

- specifies rotation (change of basis)
- eigenvalues specify relative elongation

Multivariate Gaussians

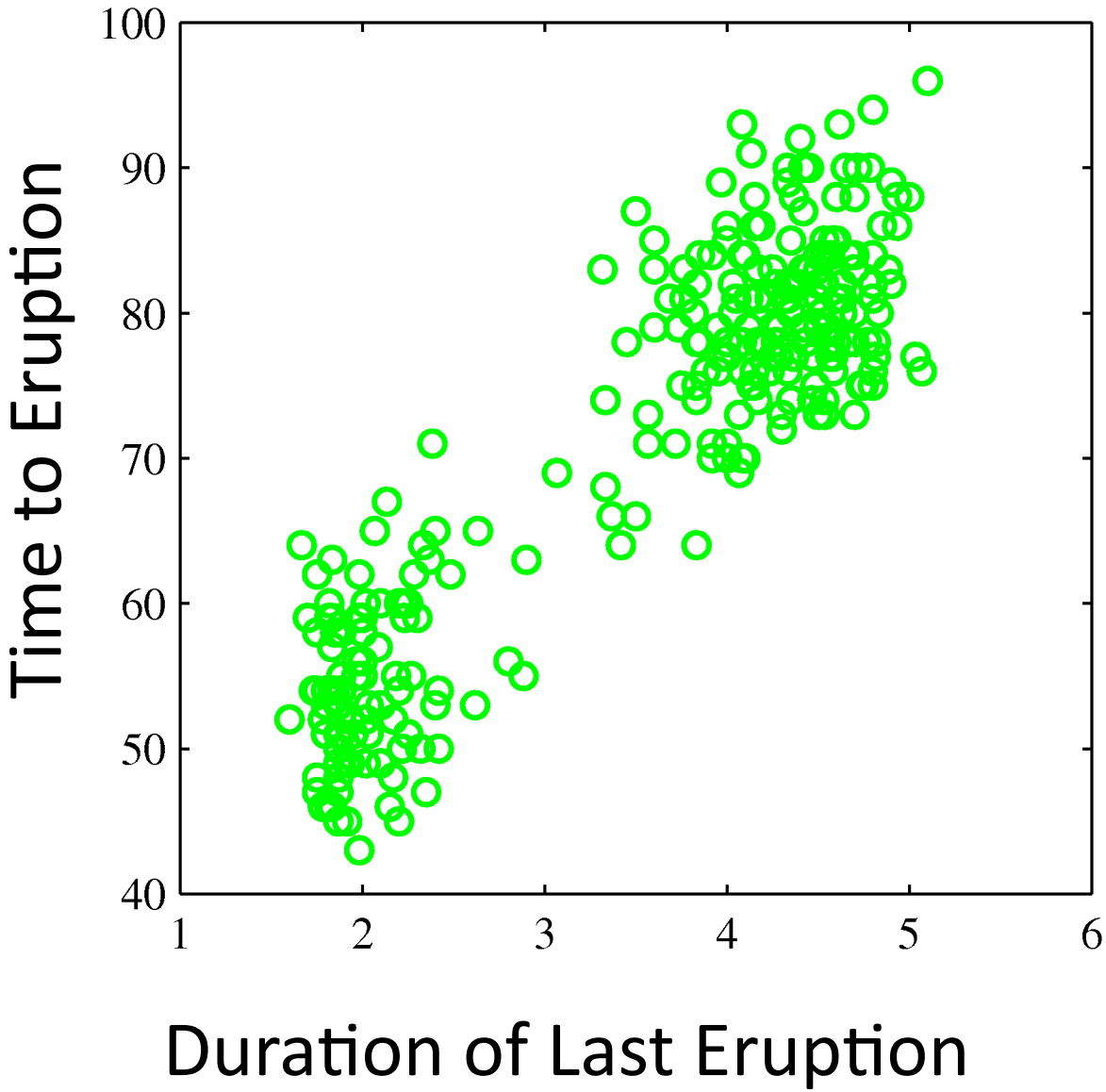


Covariance matrix, Σ, = degree to which x_i vary together

$$P(X=\boldsymbol{x}_j) = \frac{1}{(2\pi)^{m/2} \|\Sigma\|^{1/2}} \exp\left[-\frac{1}{2}\left(\mathbf{x}_j - \boldsymbol{\mu}\right)^T \Sigma^{-1} \left(\mathbf{x}_j - \boldsymbol{\mu}\right)\right]$$

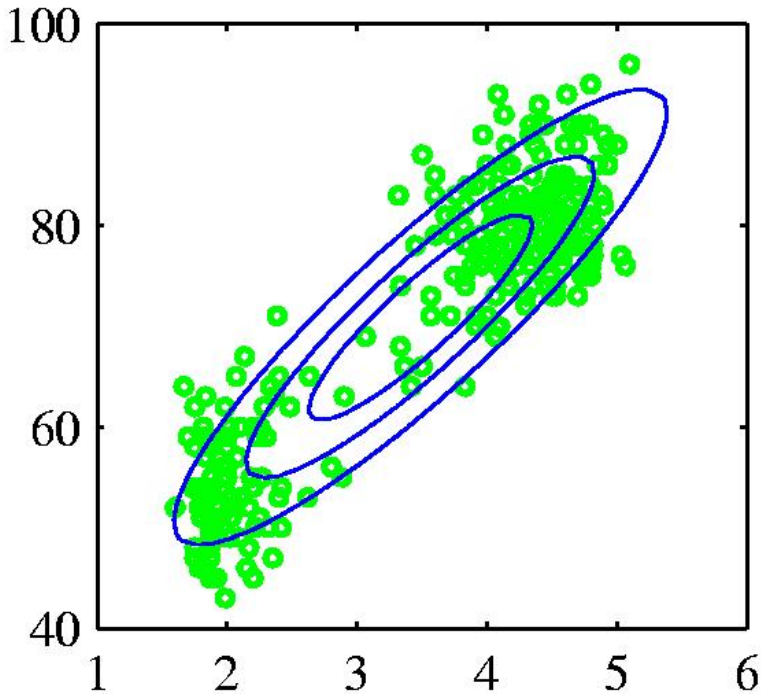
Mixtures of Gaussians (1)

Old Faithful Data Set

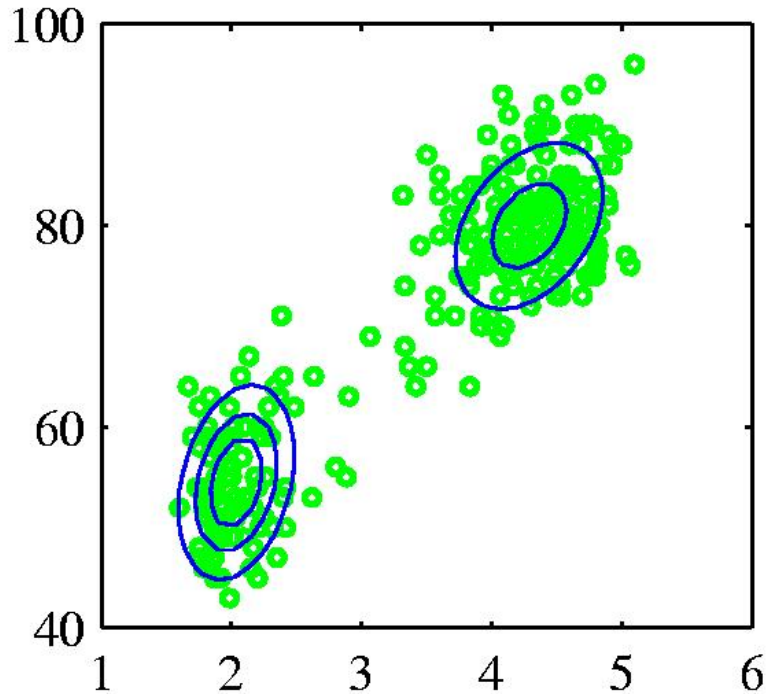


Mixtures of Gaussians (1)

Old Faithful Data Set



Single Gaussian



Mixture of two Gaussians

Mixtures of Gaussians (2)

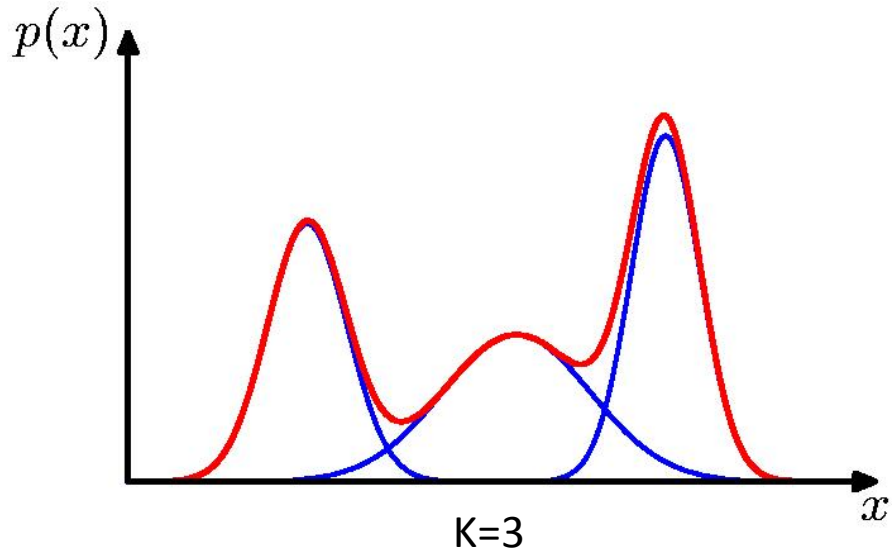
Combine simple models into a complex model:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

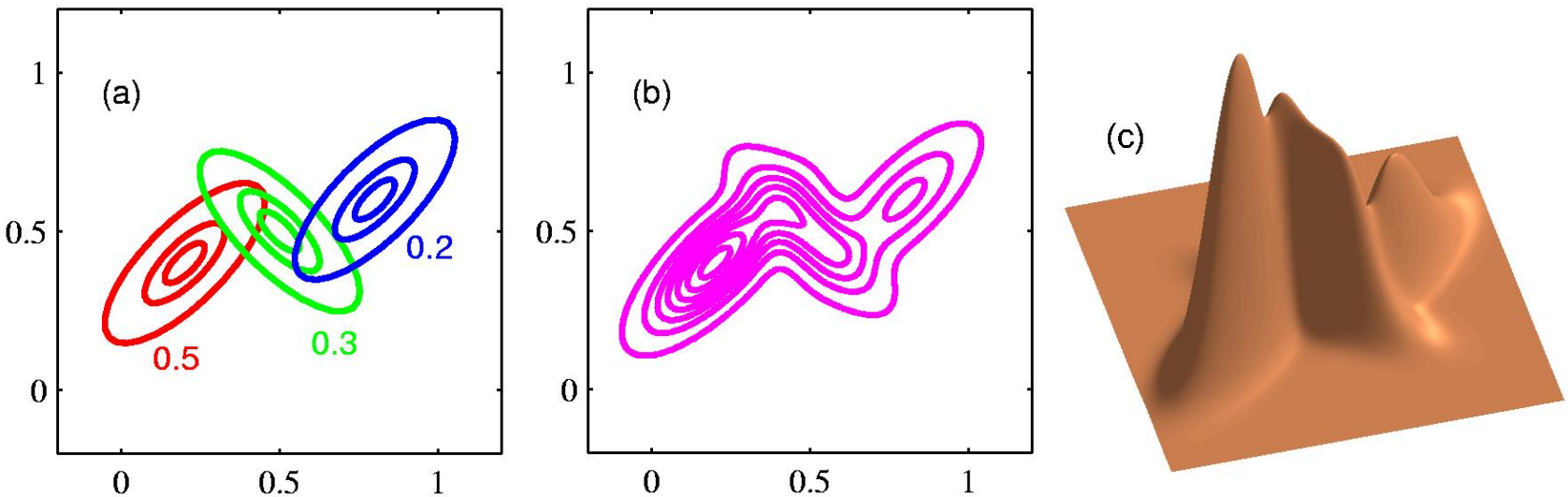
Component

Mixing coefficient

$$\forall k : \pi_k \geqslant 0 \qquad \sum_{k=1}^{K} \pi_k = 1$$

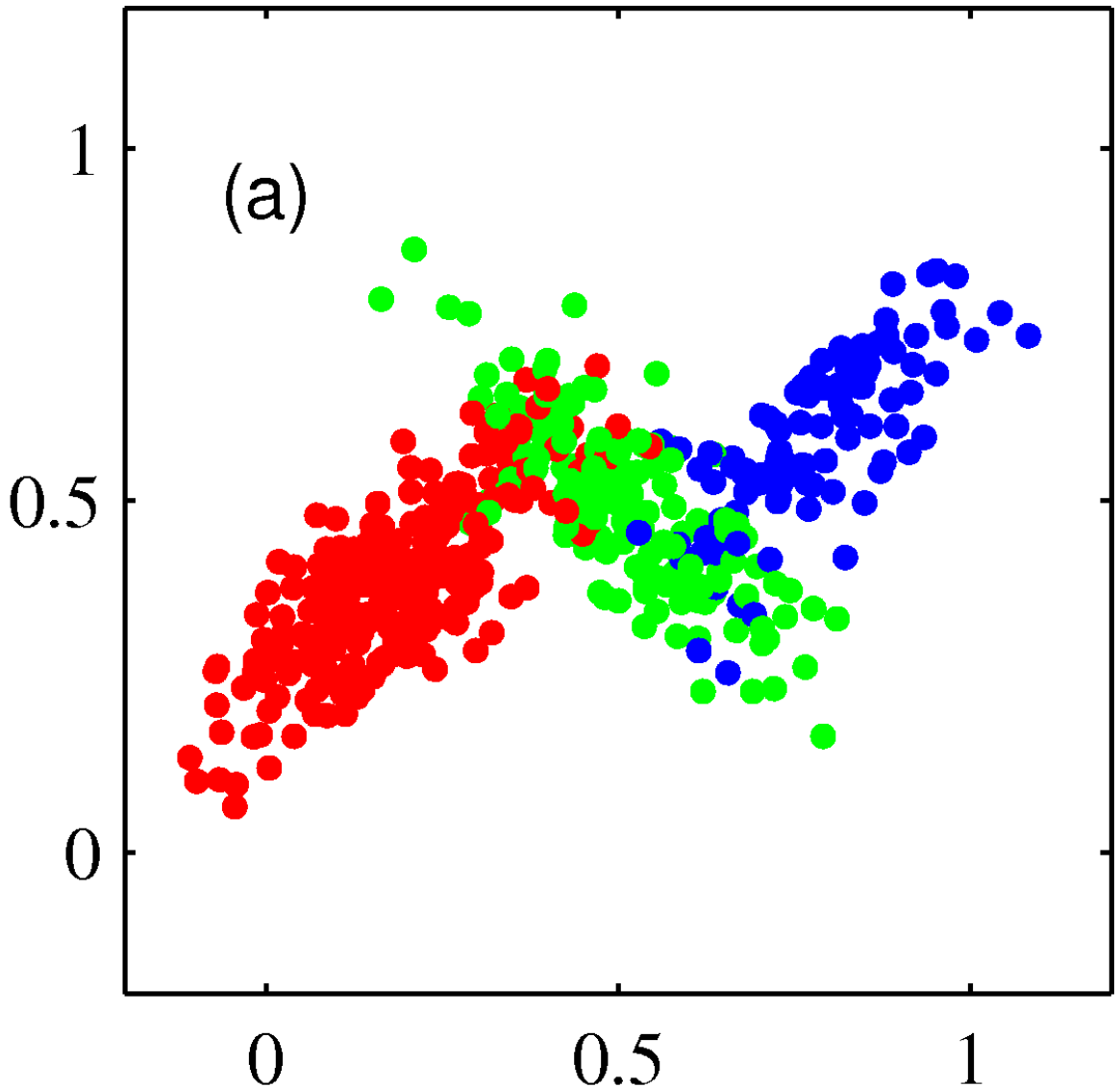


Mixtures of Gaussians (3)



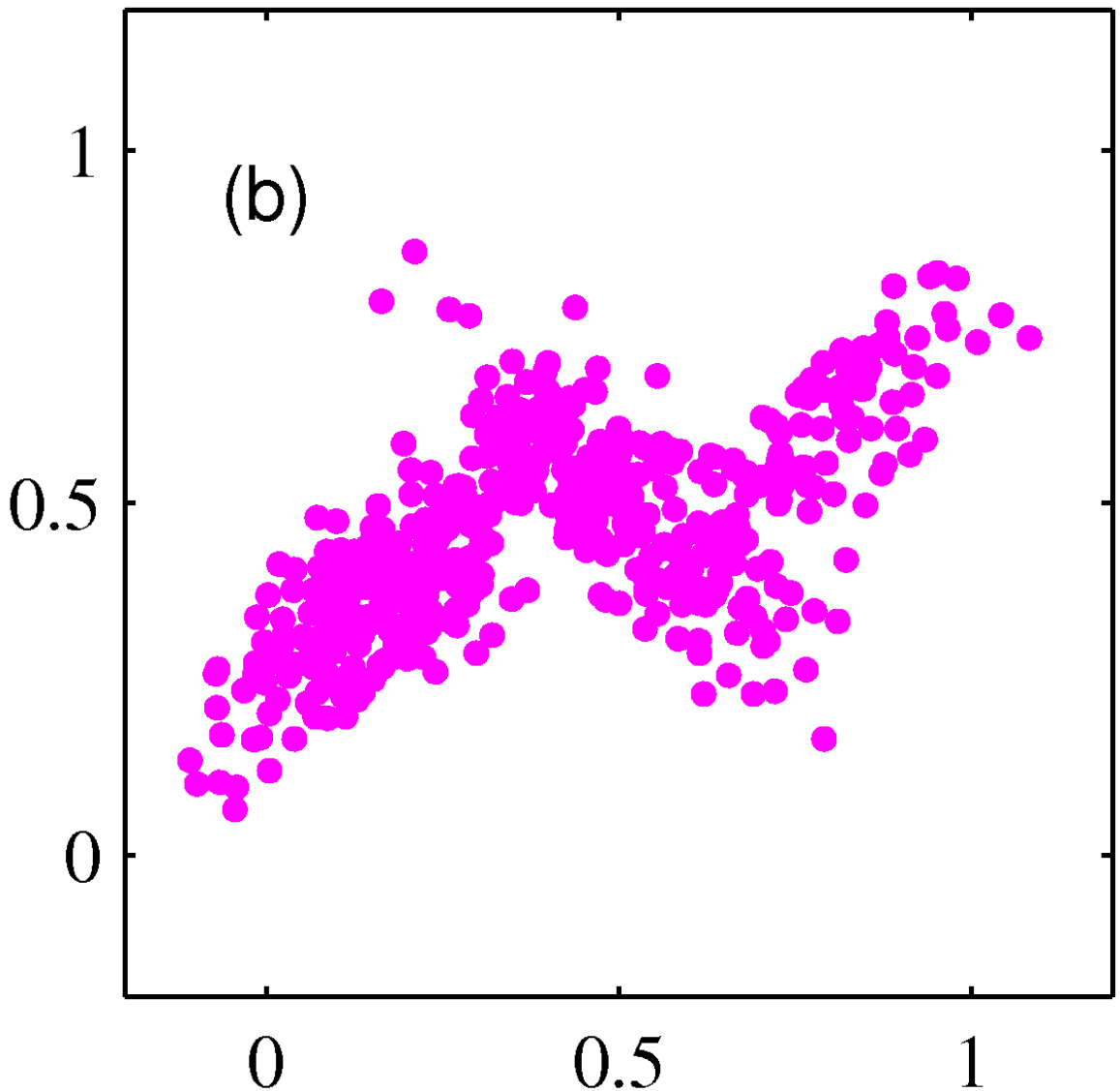
Eliminating Hard Assignments to Clusters

Model data as mixture of multivariate Gaussians



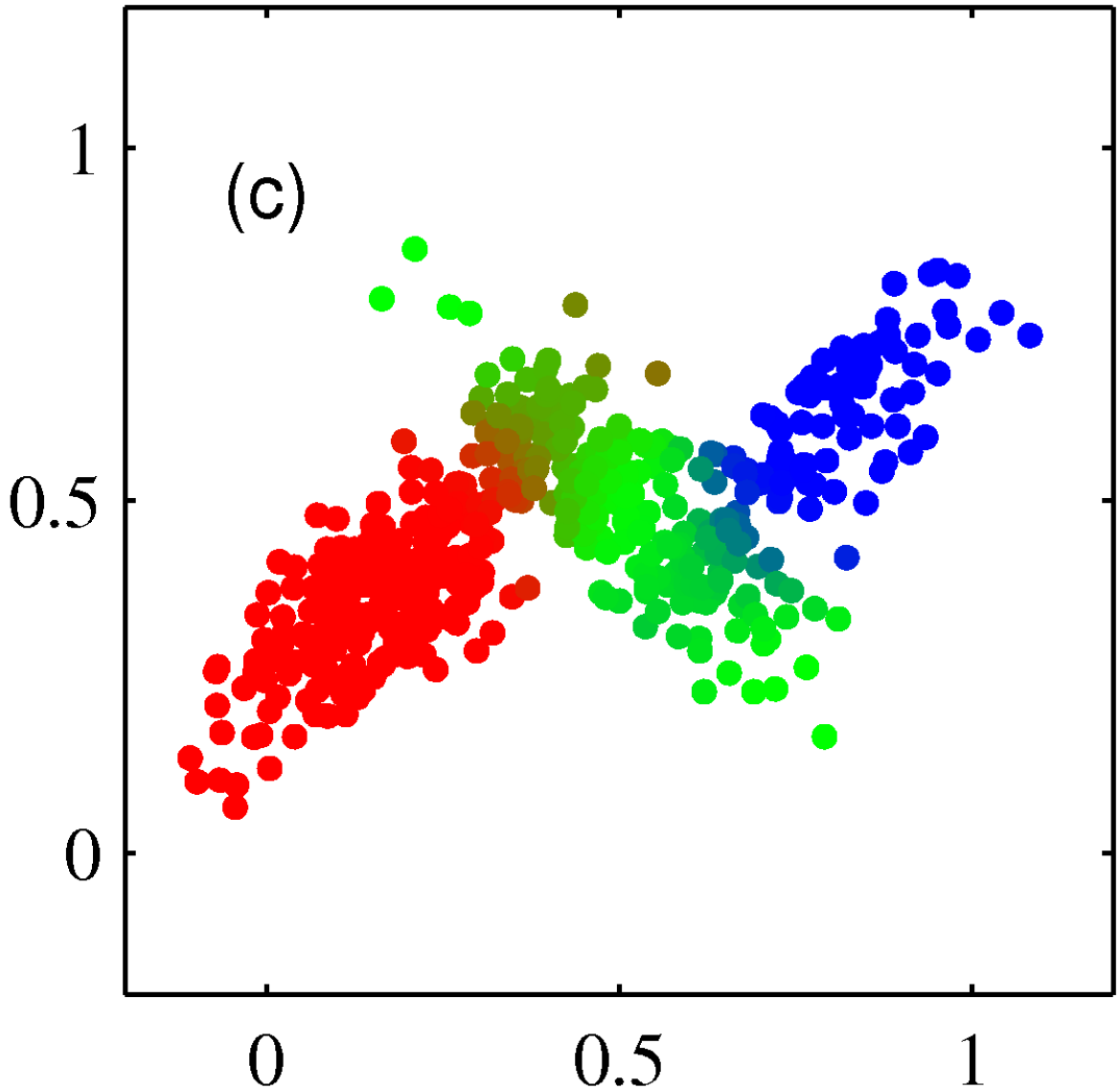
Eliminating Hard Assignments to Clusters

Model data as mixture of multivariate Gaussians



Eliminating Hard Assignments to Clusters

Model data as mixture of multivariate Gaussians



ML estimation in **supervised** setting

- Univariate Gaussian

$$\mu_{MLE} = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \sigma^2_{MLE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{\mu})^2$$

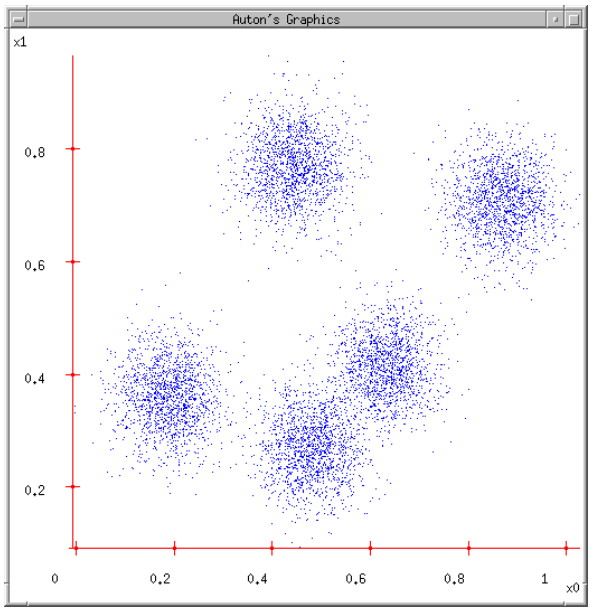
- **Mixture** of **Multi**variate Gaussians
  - ML estimate for each of the Multivariate Gaussians is given by:

$$\mu^k_{ML} = \frac{1}{n}\sum_{j=1}^{n} x_n \qquad \Sigma^k_{ML} = \frac{1}{n}\sum_{j=1}^{n}\left(\mathbf{x}_j - \mu^k_{ML}\right)\left(\mathbf{x}_j - \mu^k_{ML}\right)^T$$

Just sums over x generated from the k'th Gaussian

But what if unobserved data?

- MLE:
  - $\arg\max_\theta \prod_j P(y_i, x_j)$
  - θ: all model parameters
    - eg, class probs, means, and variances
- But we don't know y_j's!
- Maximize **marginal likelihood**:
  - $\arg\max_\theta \prod_j P(x_j) = \arg\max_\theta \prod_j \sum_{k=1}^{K} P(Y_j = k, x_j)$



How do we optimize? Closed Form?

- Maximize **marginal likelihood**:

$$\arg\max_{\theta} \prod_j P(x_j) = \arg\max \prod_j \sum_{k=1}^{K} P(Y_j = k, x_j)$$

- Almost always a hard problem!
  - Usually no closed form solution
  - Even when lgP(X,Y) is convex, lgP(X) generally isn't...
  - For all but the simplest P(X), we will have to do gradient ascent, in a big messy space with lots of local optima...

Learning general mixtures of Gaussians

$$P(y=k \mid \mathbf{x}_j) \propto \frac{1}{(2\pi)^{m/2} \|\Sigma_k\|^{1/2}} \exp\left[-\frac{1}{2}\left(\mathbf{x}_j - \mu_k\right)^T \Sigma_k^{-1}\left(\mathbf{x}_j - \mu_k\right)\right] P(y=k)$$

- Marginal likelihood:

$$\prod_{j=1}^{m} P(\mathbf{x}_j) = \prod_{j=1}^{m} \sum_{k=1}^{K} P(\mathbf{x}_j, y=k)$$

$$= \prod_{j=1}^{m} \sum_{k=1}^{K} \frac{1}{(2\pi)^{m/2} \|\Sigma_k\|^{1/2}} \exp\left[-\frac{1}{2}\left(\mathbf{x}_j - \mu_k\right)^T \Sigma_k^{-1}\left(\mathbf{x}_j - \mu_k\right)\right] P(y=k)$$

- Need to differentiate and solve for μ_k, $\sum_k$, and P(Y=k) for k=1..K
- There will be no closed form solution, gradient is complex, lots of local optimum
- **Wouldn't it be nice if there was a better way!?!**

EM

Expectation Maximization

The EM Algorithm

- A clever method for maximizing marginal likelihood:
  - $\arg\max_\theta \prod_j P(x_j) = \arg\max_\theta \prod_j \sum_{k=1}^{K} P(Y_j = k, x_j)$
  - A type of gradient ascent that can be easy to implement
    - e.g. no line search, learning rates, etc.
- Alternate between two steps:
  - Compute an expectation
  - Compute a maximization
- Not magic: **still optimizing a non-convex function with lots of local optima**
  - The computations are just easier (often, significantly so!)

EM: Two Easy Steps

Objective: $\text{argmax}_\theta \lg \prod_j \sum_{k=1}^{K} P(Y_j=k, x_j \mid \theta) = \sum_j \lg \sum_{k=1}^{K} P(Y_j=k, x_j \mid \theta)$

Data: $\{x_j \mid j=1 .. n\}$

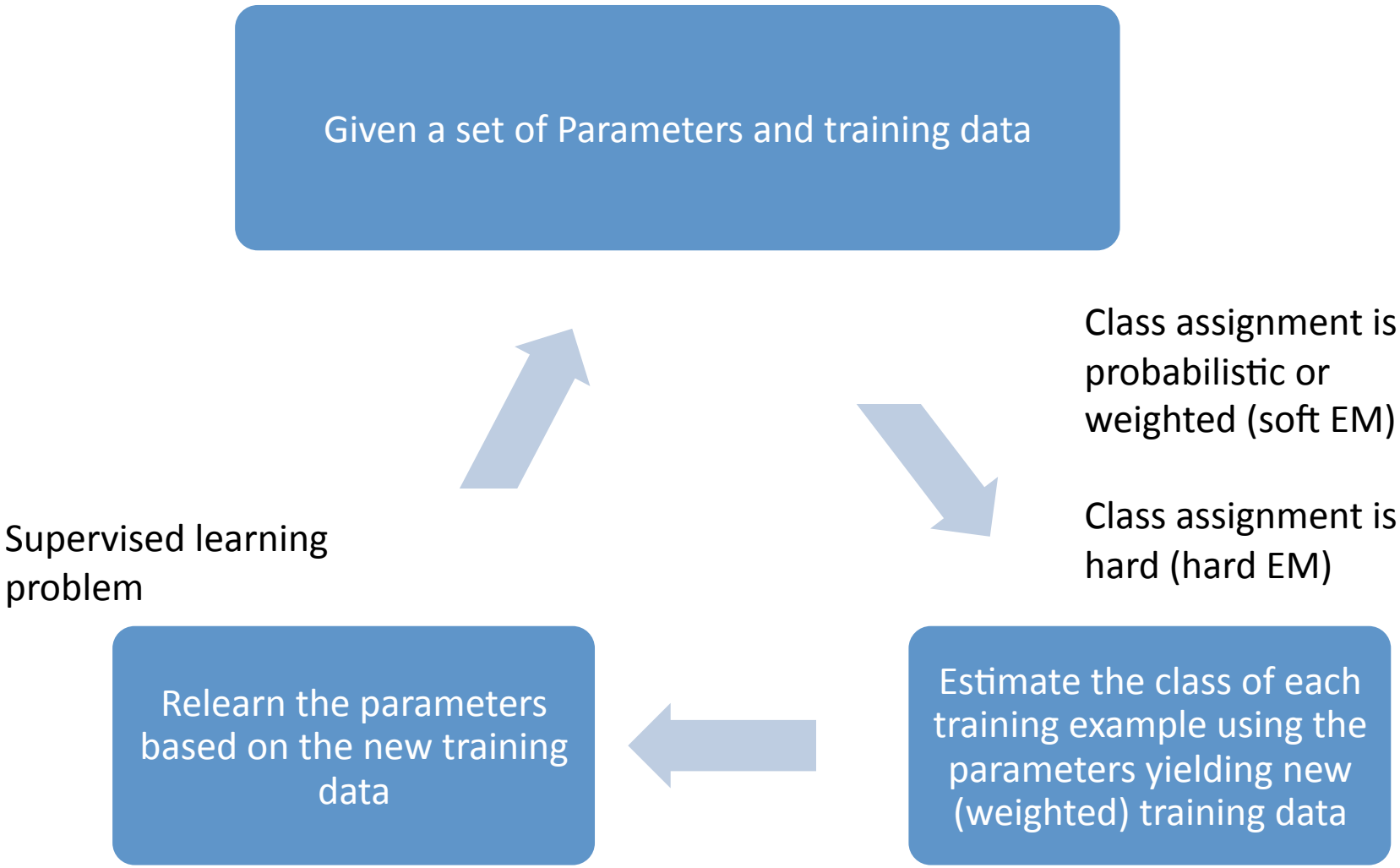
Notation a bit inconsistent
Parameters = $\theta=\lambda$

- **E-step**: Compute expectations to "fill in" missing y values according to current parameters, θ
  - For all examples j and values k for Y_j, compute: $P(Y_j=k \mid x_j, \theta)$

- **M-step**: Re-estimate the parameters with "weighted" MLE estimates
  - Set $\theta = \text{argmax}_\theta \sum_j \sum_k P(Y_j=k \mid x_j, \theta) \log P(Y_j=k, x_j \mid \theta)$

Especially useful when the E and M steps have closed form solutions!!!

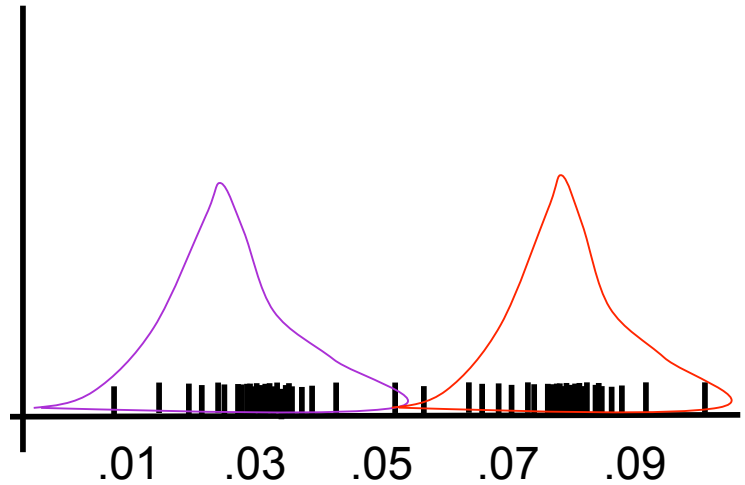
EM algorithm: Pictorial View



Simple example: learn means only!

Consider:

- 1D data
- Mixture of k=2 Gaussians
- Variances fixed to σ=1
- Distribution over classes is uniform
- Just need to estimate μ_1 and μ_2



$$\prod_{j=1}^{m}\sum_{k=1}^{K} P(x, Y_j = k) \propto \prod_{j=1}^{m}\sum_{k=1}^{K=2} \exp\left[-\frac{1}{2\sigma^2}\left\|x - \mu_k\right\|^2\right] P(Y_j = k)$$

EM for GMMs: only learning means

Iterate: On the *t*'th iteration let our estimates be

$$\lambda_t = \{ \mu_1^{(t)}, \mu_2^{(t)} \ldots \mu_K^{(t)} \}$$

E-step

Compute "expected" classes of all datapoints

$$P\left(Y_j = k \middle| x_j, \mu_1 \ldots \mu_K\right) \propto \exp\left(-\frac{1}{2\sigma^2}\left\|x_j - \mu_k\right\|^2\right) \mathrm{P}\left(Y_j = k\right)$$

M-step

Compute most likely new **μ**s given class expectations

$$\mu_k = \frac{\sum_{j=1}^{m} P\left(Y_j = k \middle| x_j\right) x_j}{\sum_{j=1}^{m} P\left(Y_j = k \middle| x_j\right)}$$

E.M. for General GMMs

$p_k^{(t)}$ is shorthand for estimate of $P(y=k)$ on t'th iteration

Iterate: On the t'th iteration let our estimates be

$$\lambda_t = \{ \mu_1^{(t)}, \mu_2^{(t)} \ldots \mu_K^{(t)}, \Sigma_1^{(t)}, \Sigma_2^{(t)} \ldots \Sigma_K^{(t)}, p_1^{(t)}, p_2^{(t)} \ldots p_K^{(t)} \}$$

E-step

Compute "expected" classes of all datapoints for each class

$$P(Y_j = k | x_j, \lambda_t) \propto p_k^{(t)} p(x_j | \mu_k^{(t)}, \Sigma_k^{(t)})$$

Just evaluate a Gaussian at x_j

M-step

Compute weighted MLE for $\boldsymbol{\mu}$ given expected classes above

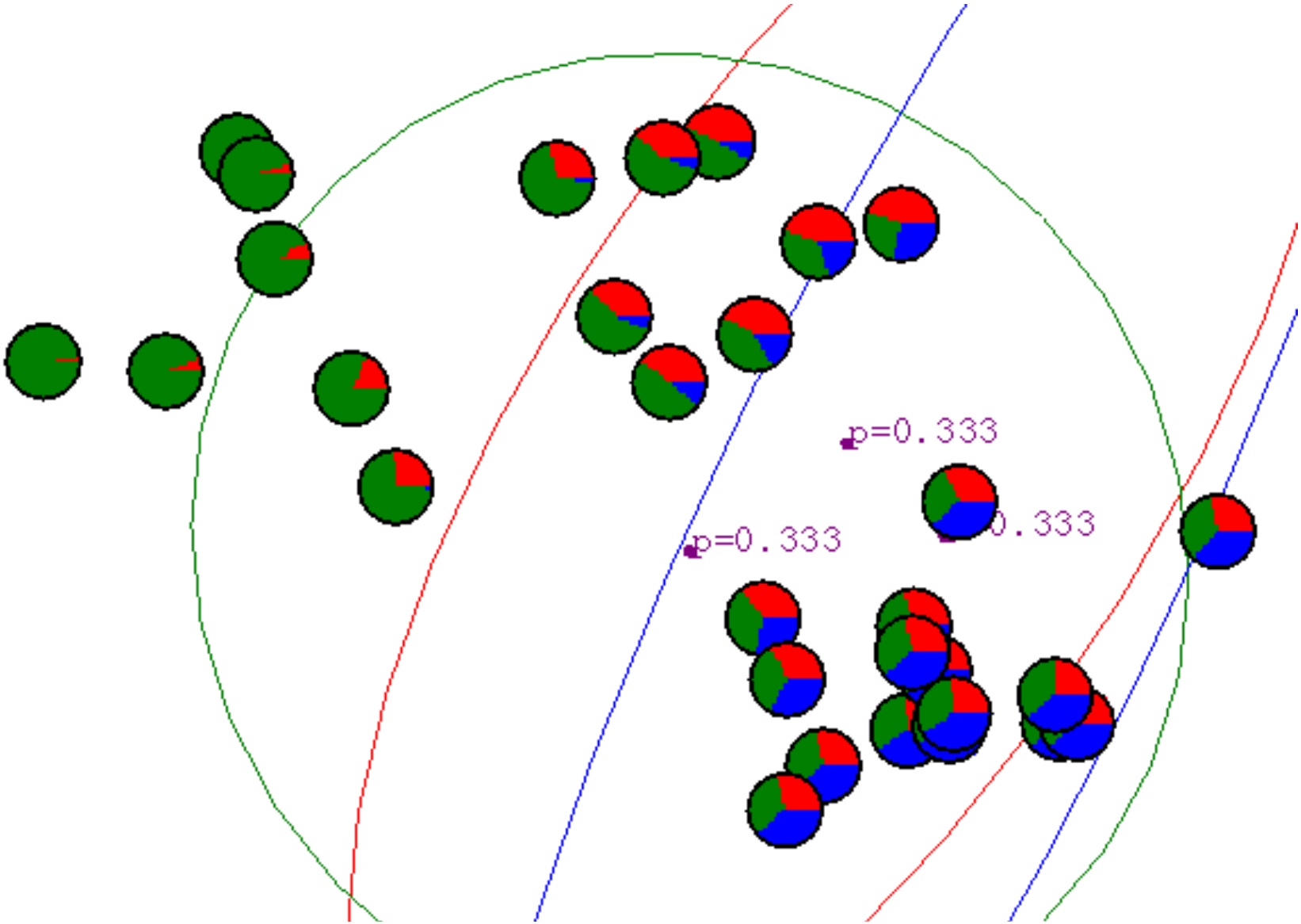
$$\mu_k^{(t+1)} = \frac{\sum_j P(Y_j = k | x_j, \lambda_t) x_j}{\sum_j P(Y_j = k | x_j, \lambda_t)}$$

$$\Sigma_k^{(t+1)} = \frac{\sum_j P(Y_j = k | x_j, \lambda_t) [x_j - \mu_k^{(t+1)}][x_j - \mu_k^{(t+1)}]^T}{\sum_j P(Y_j = k | x_j, \lambda_t)}$$

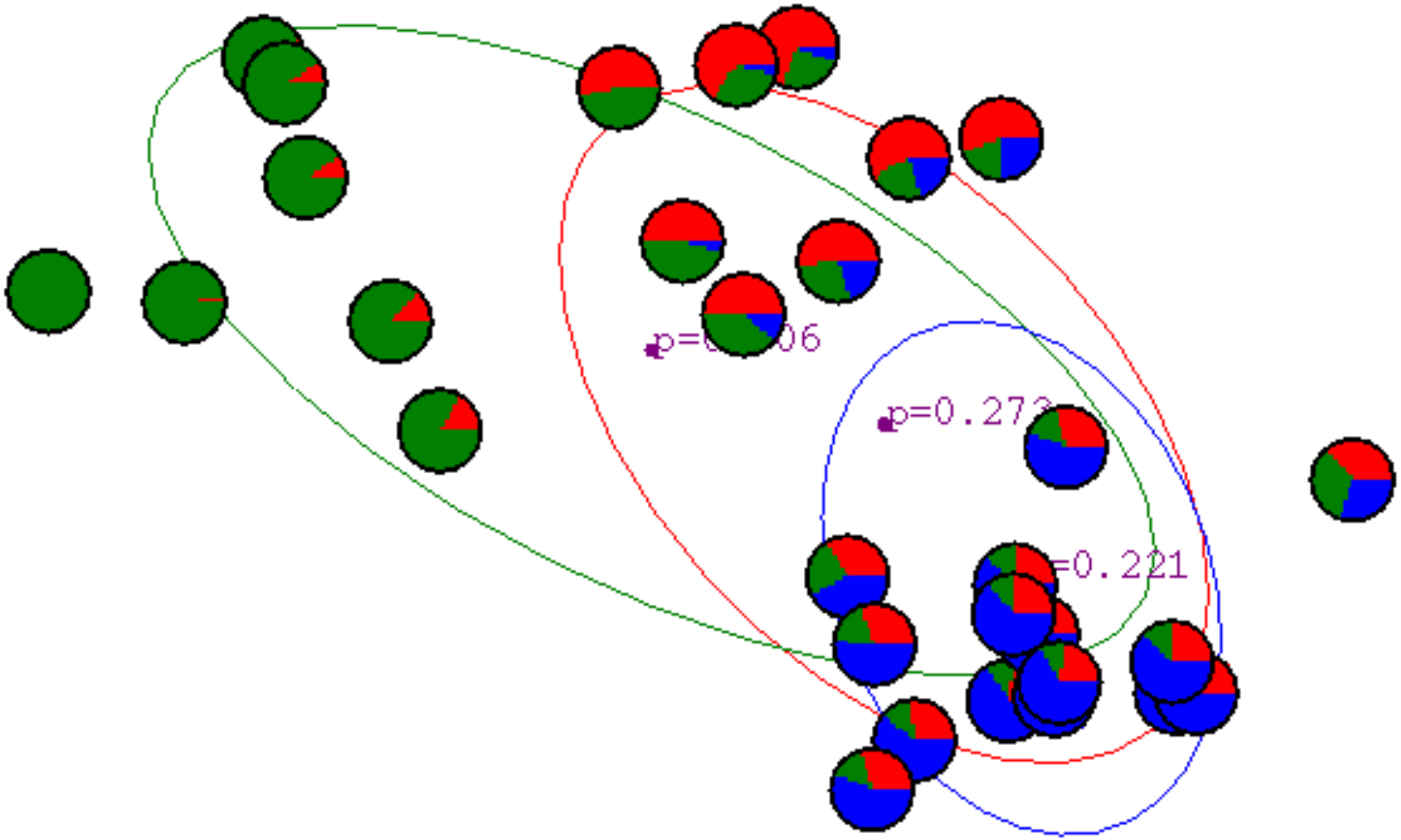
$$p_k^{(t+1)} = \frac{\sum_j P(Y_j = k | x_j, \lambda_t)}{m}$$

m = #training examples

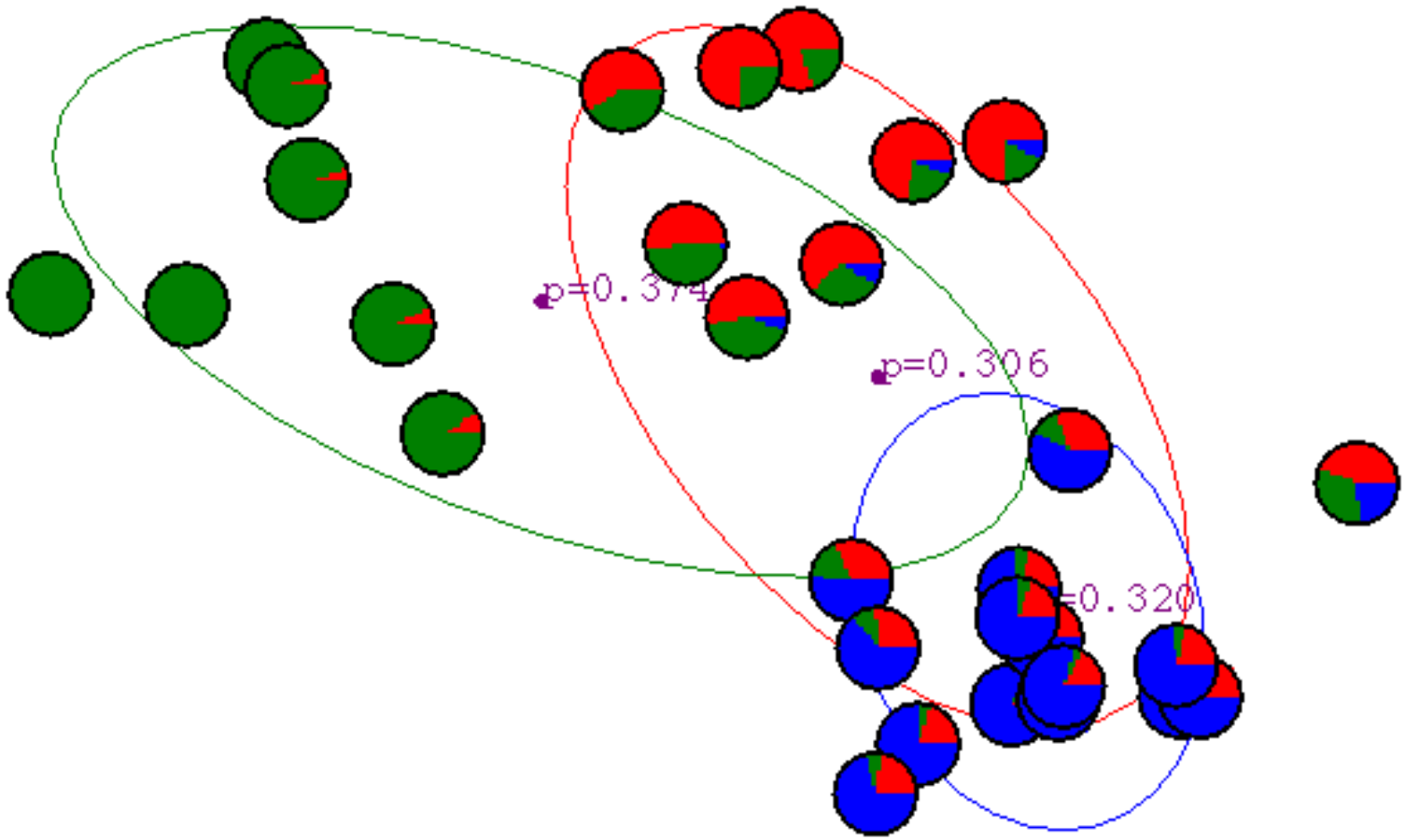
Gaussian Mixture Example: Start



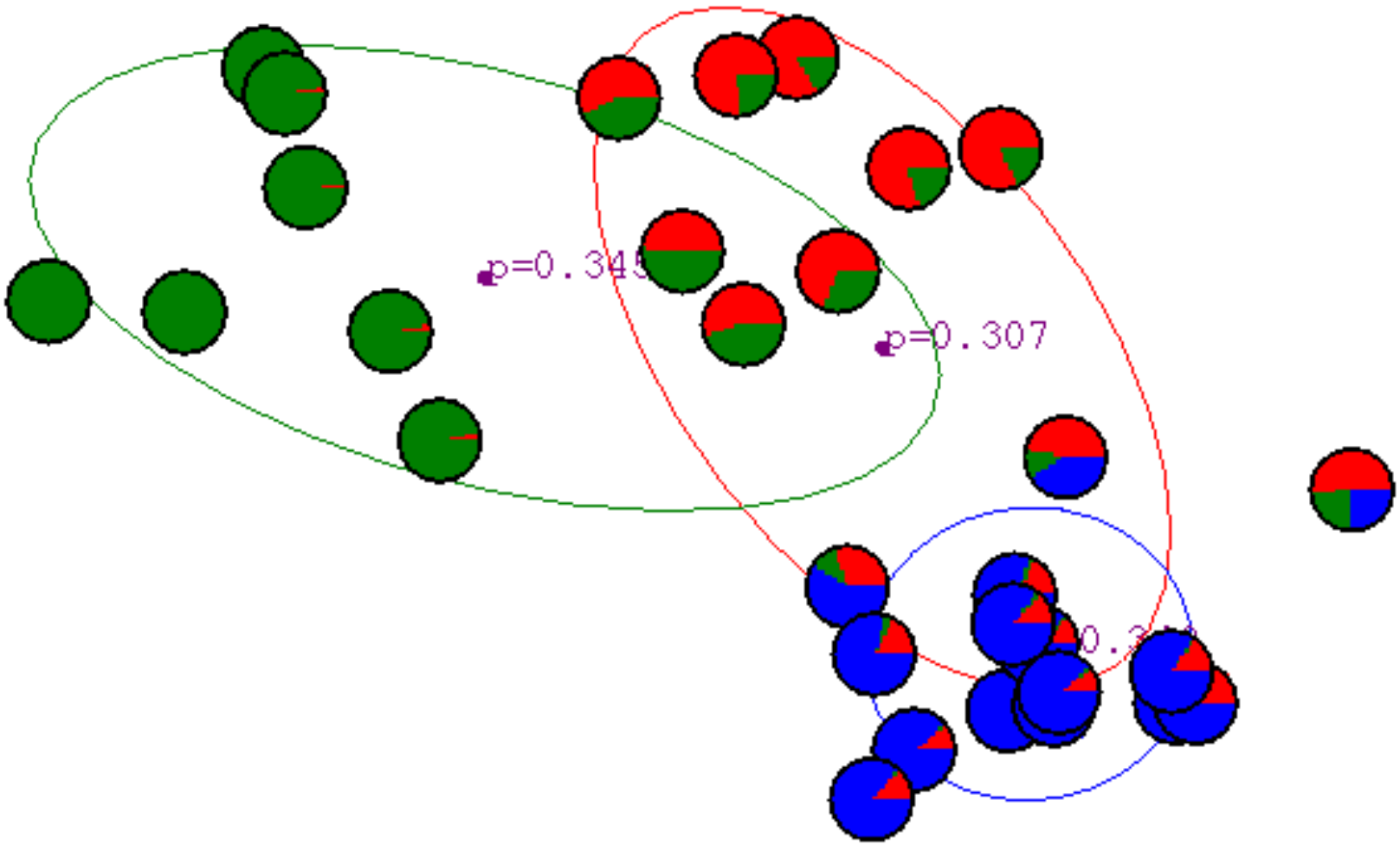
After First Iteration



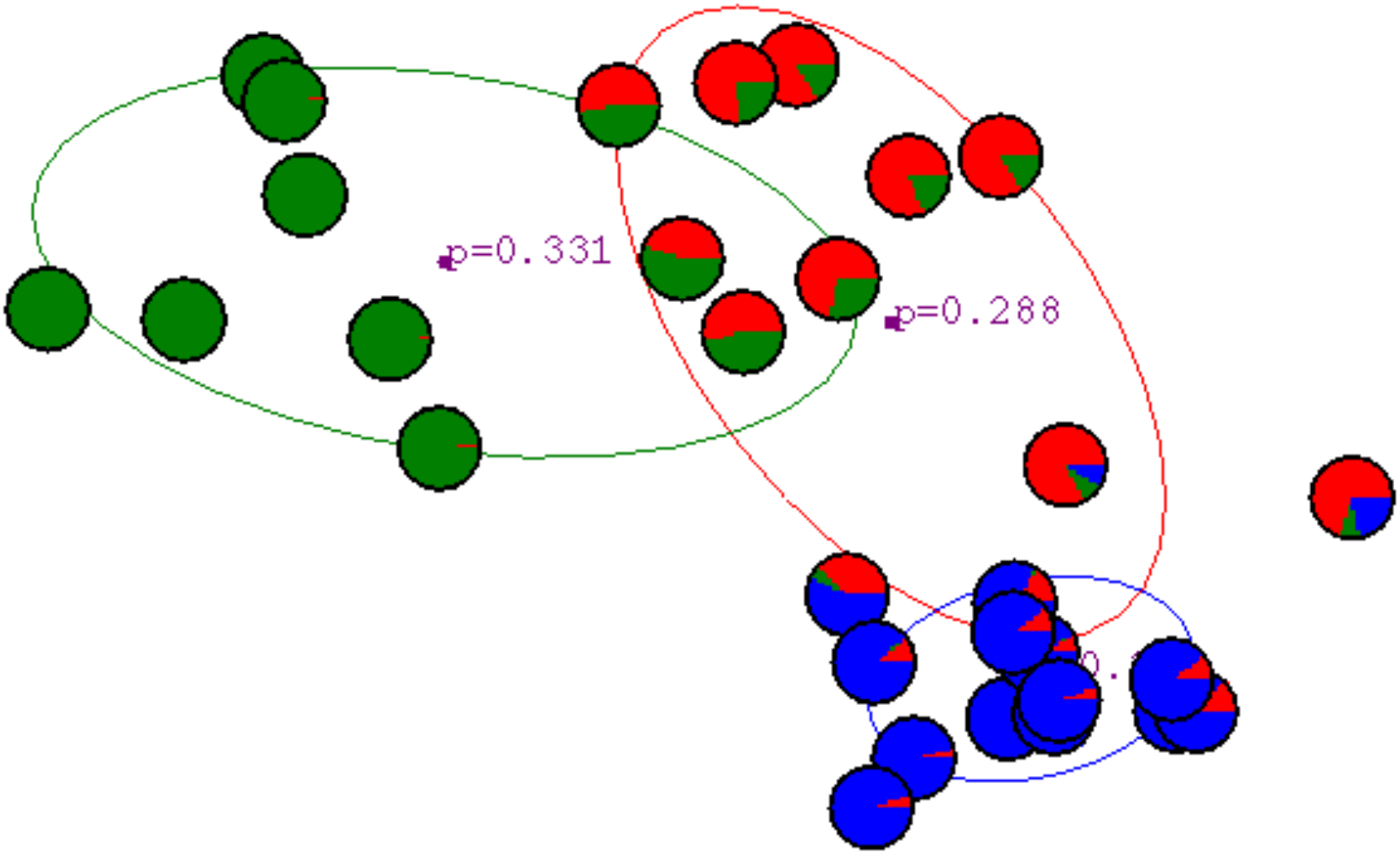
After 2nd Iteration



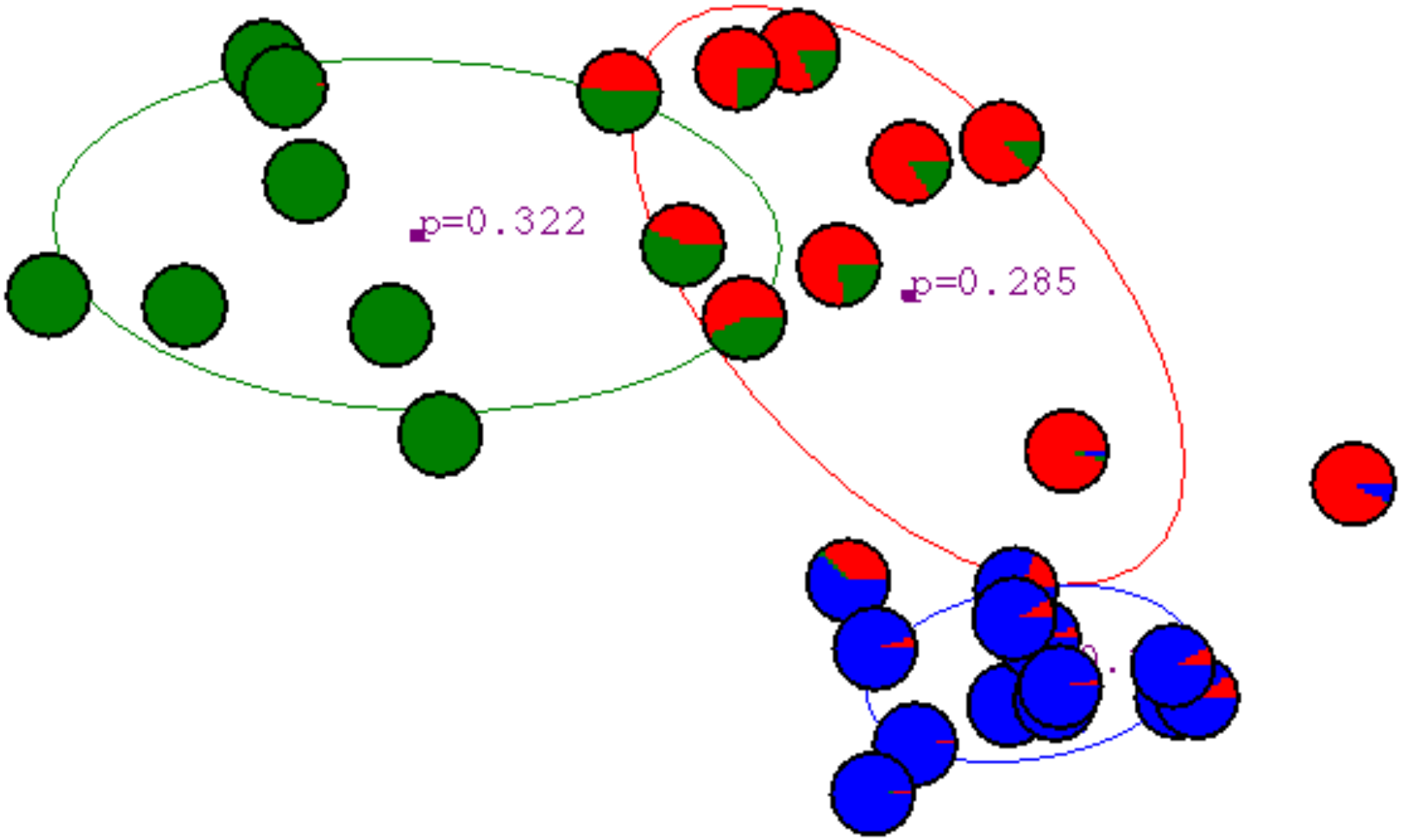
After 3rd iteration



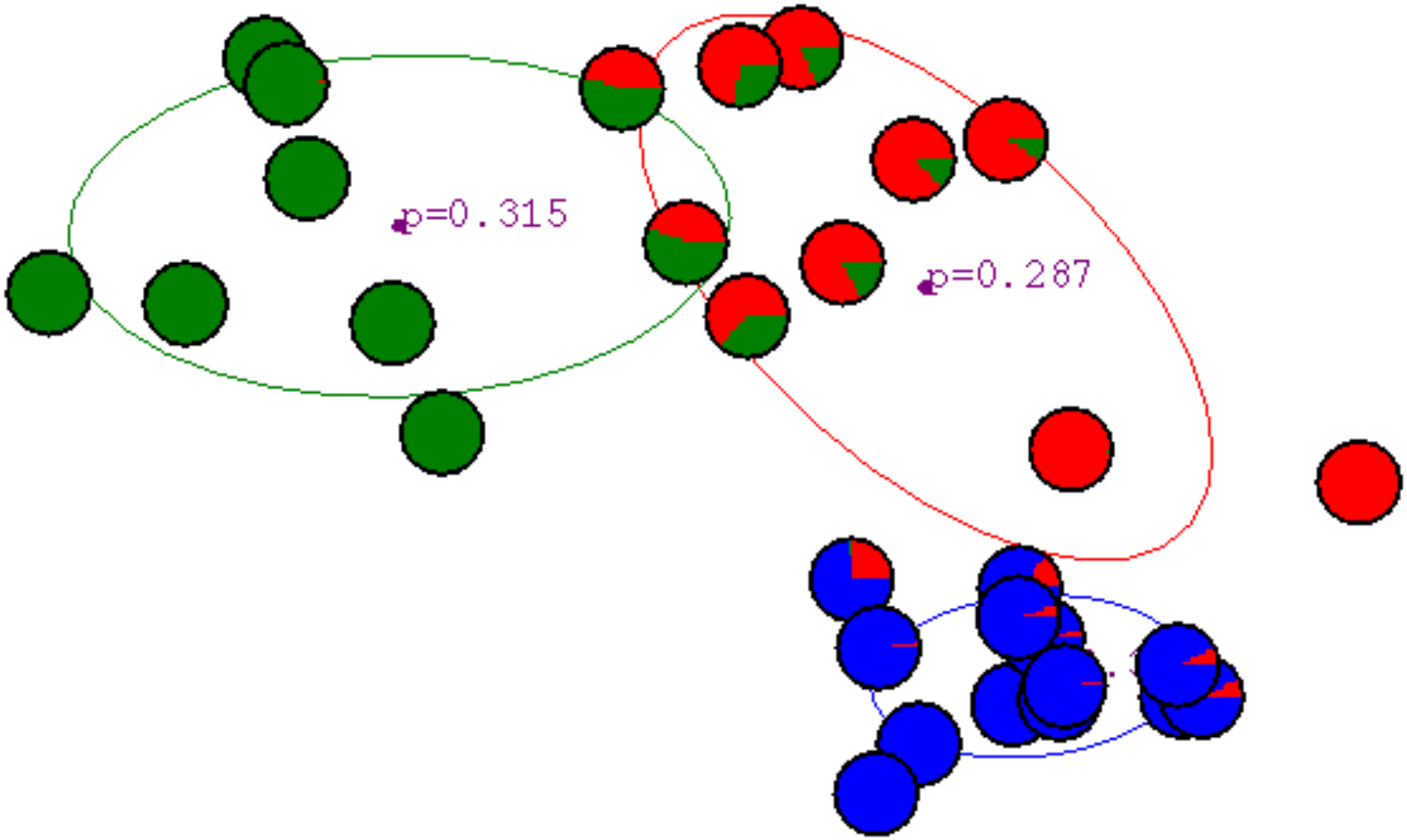
After 4th iteration



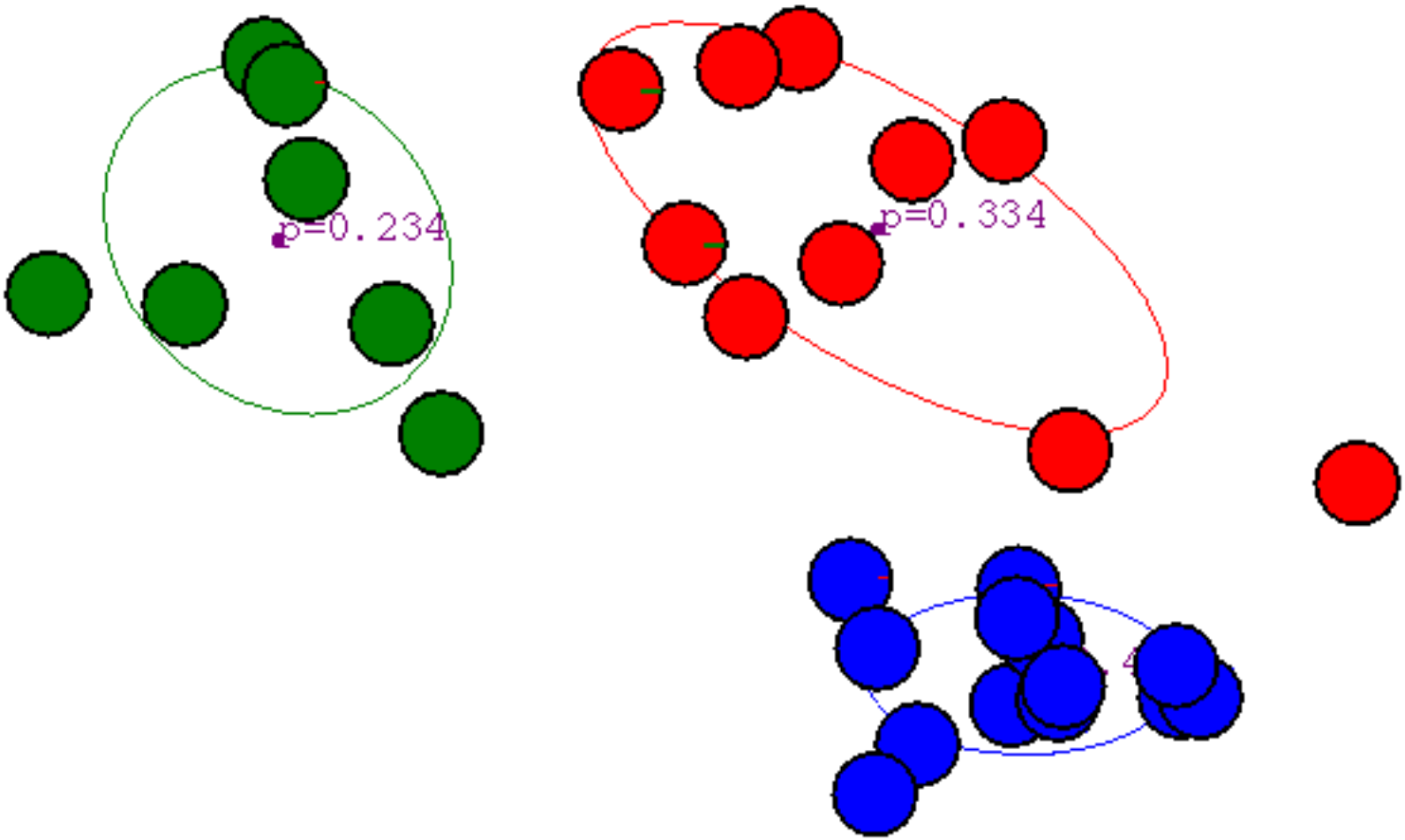
After 5th iteration



After 6th iteration



After 20th iteration



What if we do hard assignments?

Iterate: On the t'th iteration let our estimates be

$$\lambda_t = [\mu_1^{(t)}, \mu_2^{(t)}, \ldots \mu_3^{(t)}]$$

E-step

Compute "expected" classes of all datapoints

$$P\left(Y_j = k \middle| x_j, \mu_1 \ldots \mu_K\right) \propto \exp\left(-\frac{1}{2\sigma^2}\left\|x_j - \mu_k\right\|^2\right) \mathrm{P}\left(Y_j = k\right)$$

M-step

Compute most likely new **μ**s given class expectations

$$\mu_k = \frac{\sum_{j=1}^{m} P\left(Y_j = k \middle| x_j\right) x_j}{\sum_{j=1}^{m} P\left(Y_j = k \middle| x_j\right)}$$

$$\mu_k = \frac{\delta\left(Y_j = k, x_j\right) x_j}{\sum_{j=1}^{m} \delta\left(Y_j = k, x_j\right)}$$

δ represents hard assignment to "most likely" or nearest cluster

Equivalent to k-means clustering algorithm!!!

Implementation

`sklearn.mixture.GaussianMixture` implements GMMs within `sklearn`.

- `GaussianMixture` creates the class
  - `n_components` indicates the number of Gaussians to use.
  - `covariance_type` is type of covariance
    - `full`
    - `spherical`
    - `diag`
    - `tied` means all components share the same covariance matrix
  - `max_iter` is EM iterations to use
- Functions
  - `M.fit(X)` fits using the EM algorithm
  - `M.predict_proba(X)` is the posterior probability of each component given the data
  - `M.predict(X)` predict the class labels of each data point

Further Reading

- `sklearn.mixture.GaussianMixture`
- Python Data Science Handbook: GMMs