Intro to Machine Learning for Data Science

charles nainan

generative vs. discriminative

- Generative – model joint density $p_{XY}(x,y)$
  - Ex: Naive Bayes, Bayesian Nets, Hidden Markov Models(HMM)
  - Allows us to sample data from joint: very powerful!
  - Allows for inference on parameters, predictions,etc.

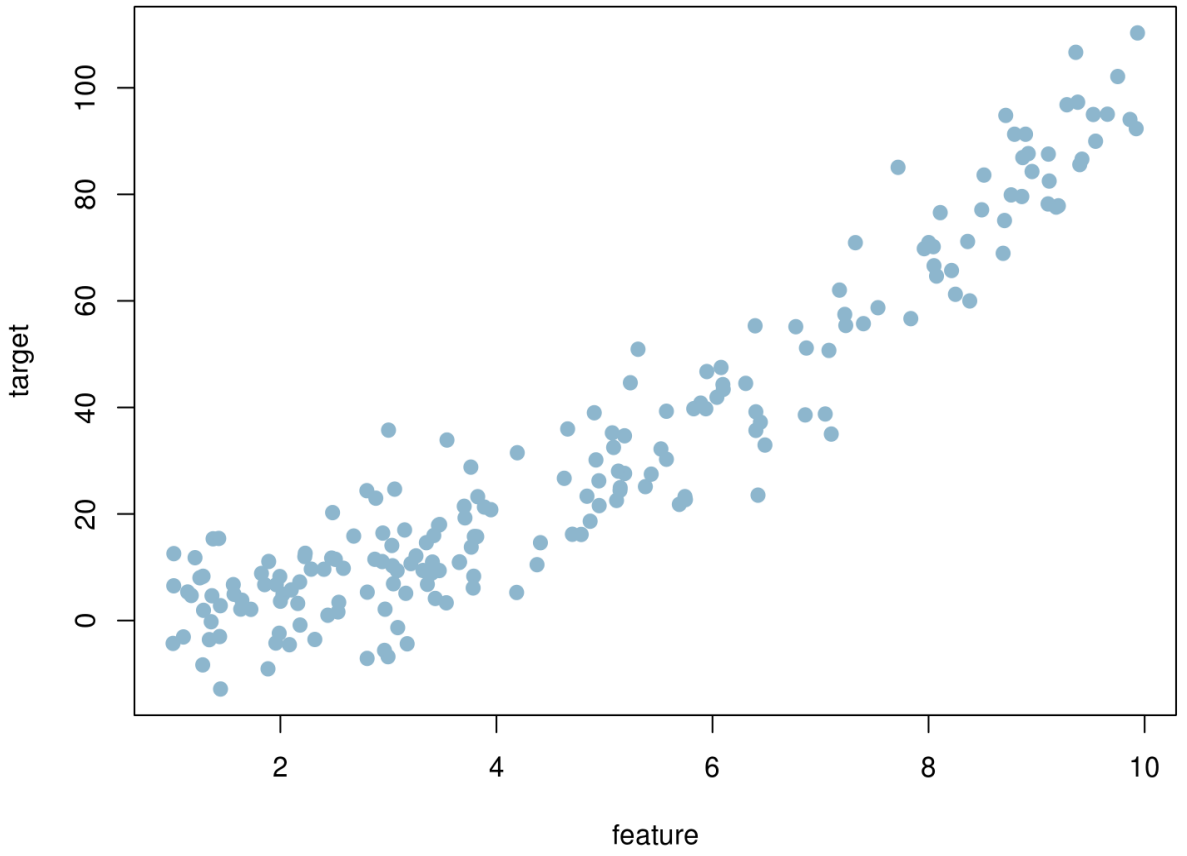
- Discriminative – models decision boundary or y = f(x) directly
  - Ex: Logistic Regression,CRF, SVM, Decision Trees
  - Conditional – models p(y|x)
    - Y can still be sampled form p(y|x)
    - Allows partial inference
  - Purely discriminative – models y as f(x) , probabilistic inference is difficult

regression

data

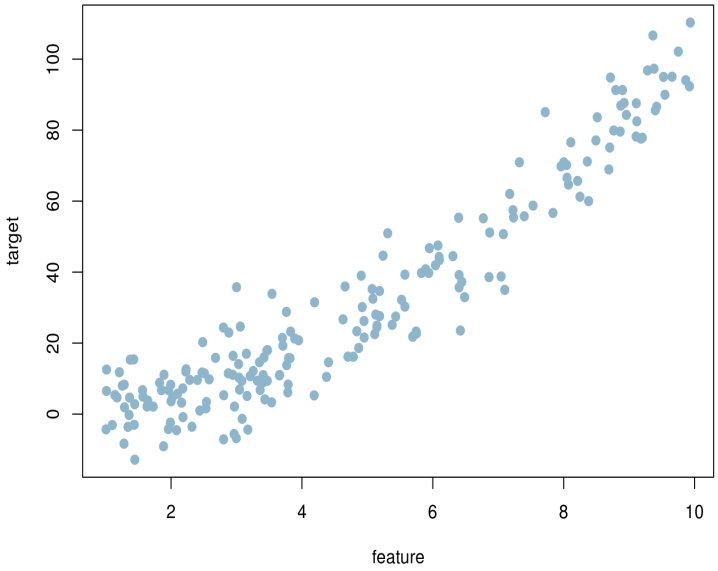
feature	label
3.44	16.79
1.47	-4.07
1.38	7.96
1.09	-1.17
1.28	-1.66
2.43	10.52
3.16	14.3
.	.
.	.
.	.
2.11	12.08

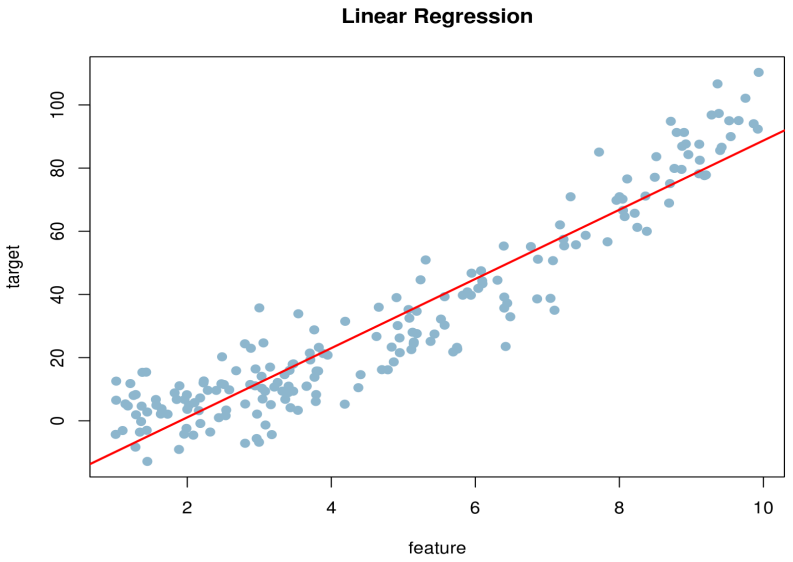
plot

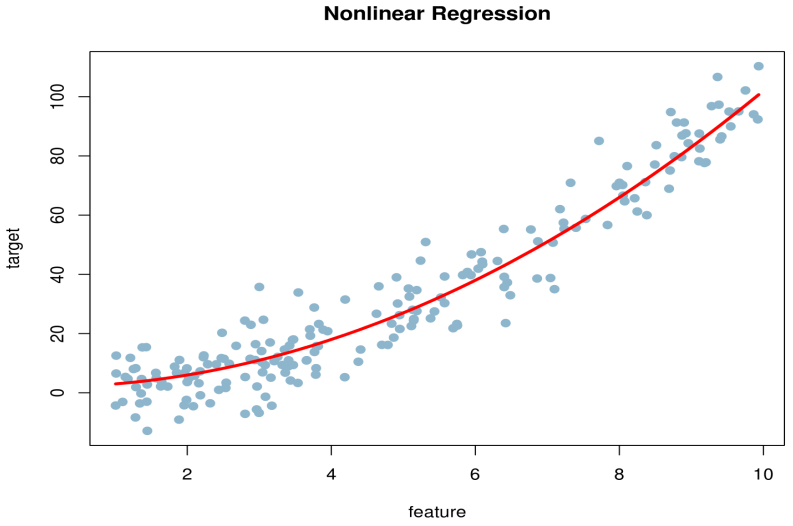


regression

models





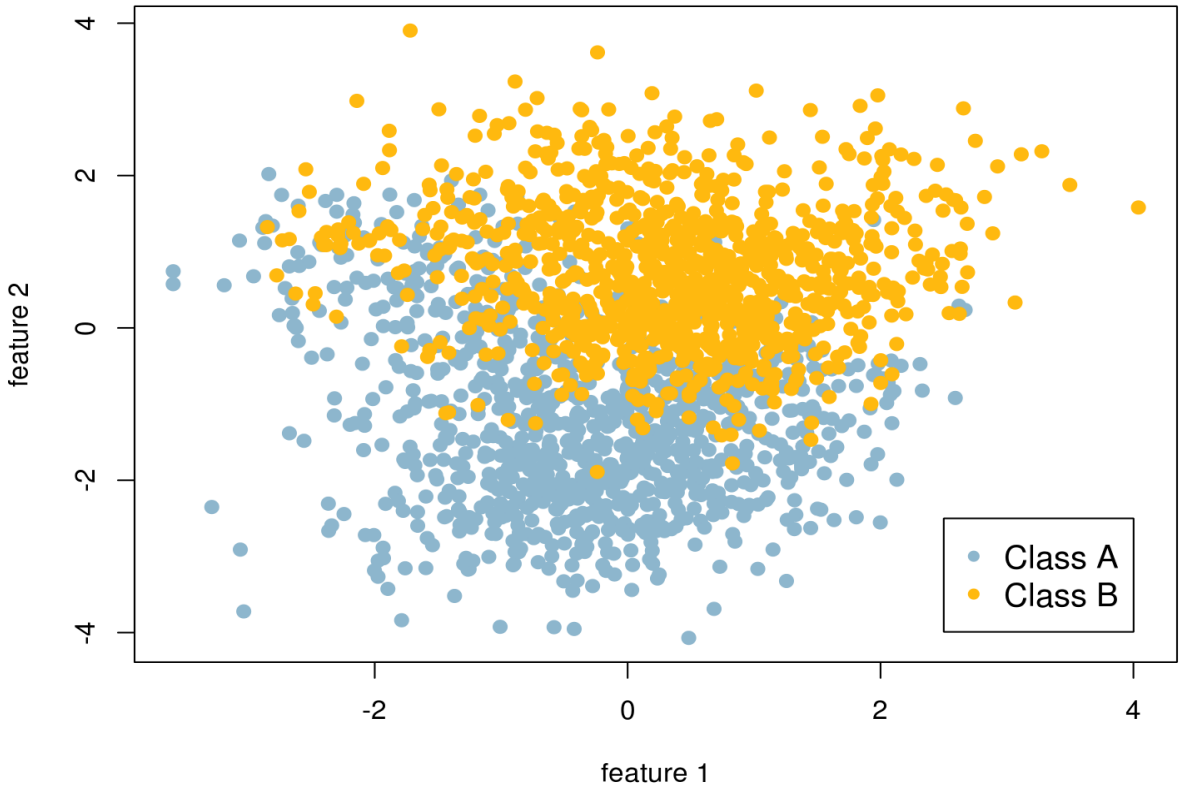


classification

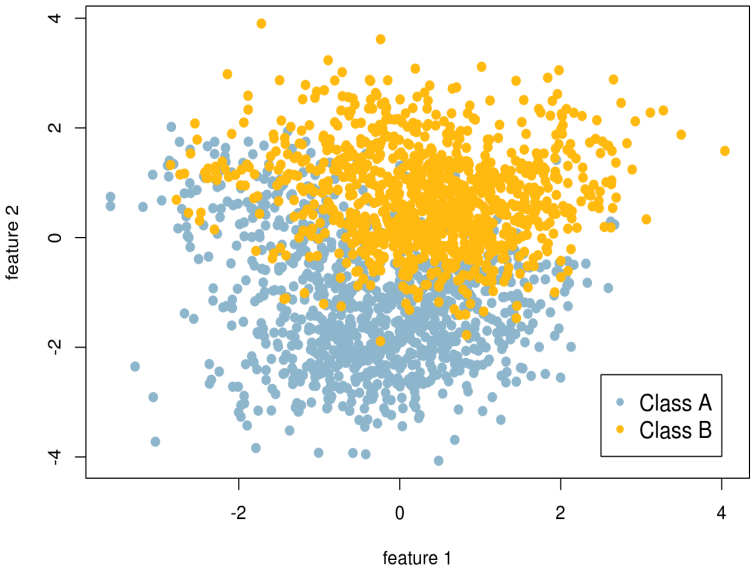
data

feature1	feature2	label
3.44	2.66	A
1.47	-1.26	A
1.38	2.24	B
1.09	5.8	B
1.28	-3.22	B
2.43	6.72	A
3.16	0.04	A
.	.	.
.	.	.
.	.	.
2.13	-0.22	B

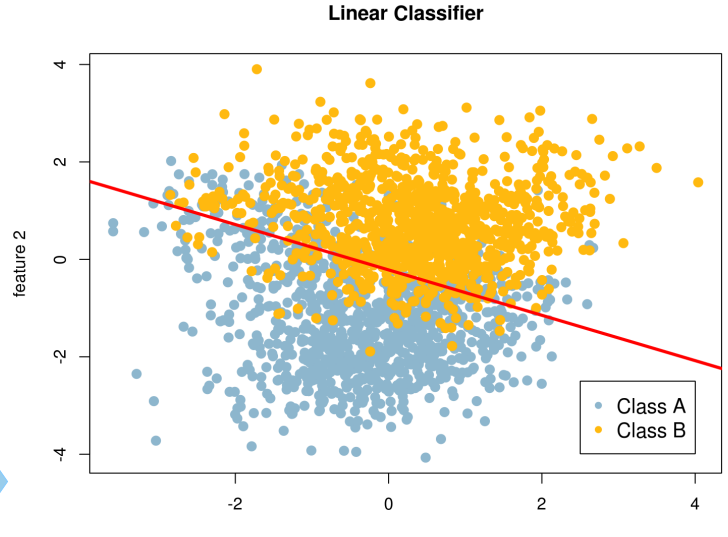
plot

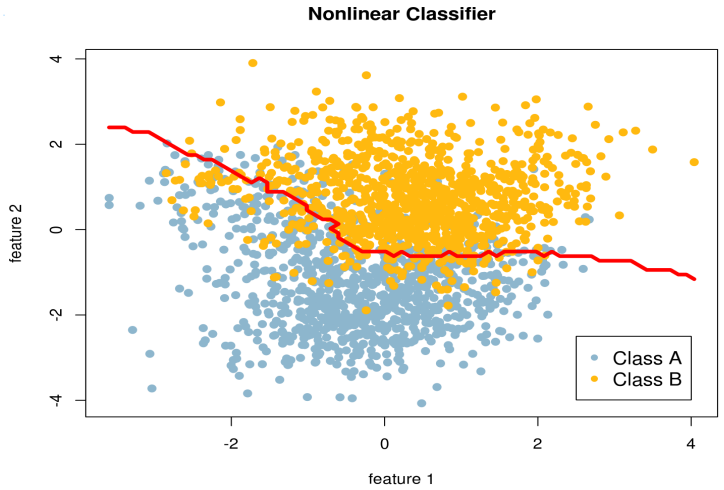


classification



model



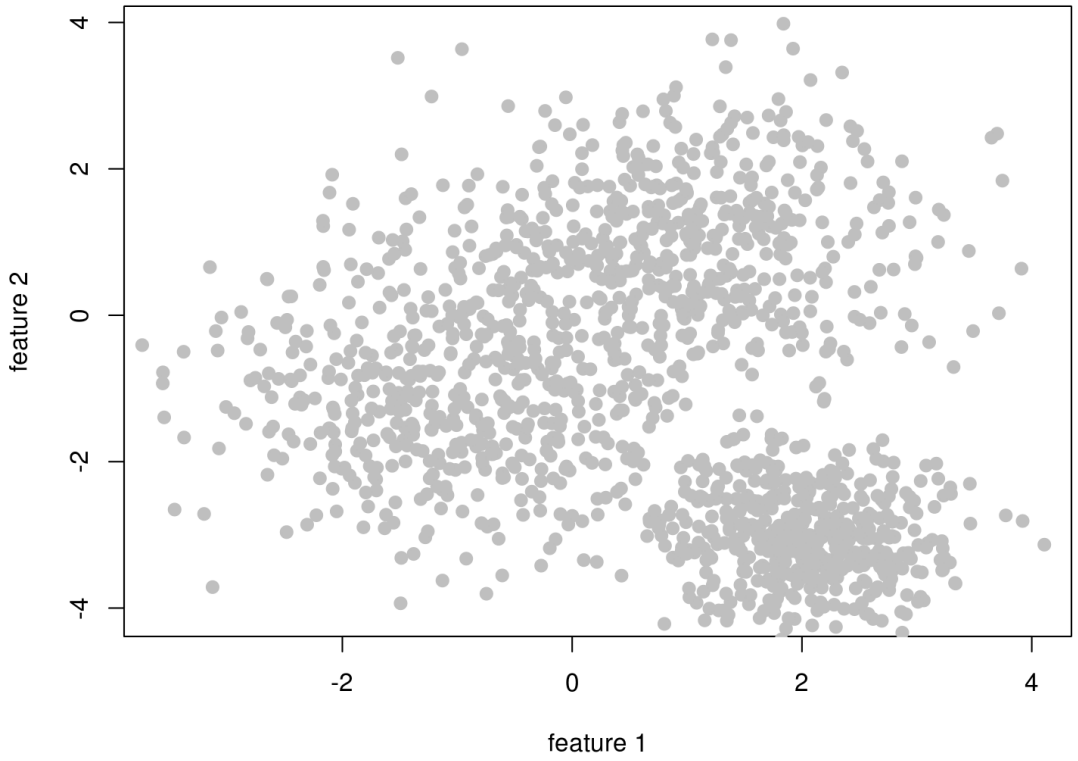


clustering

data

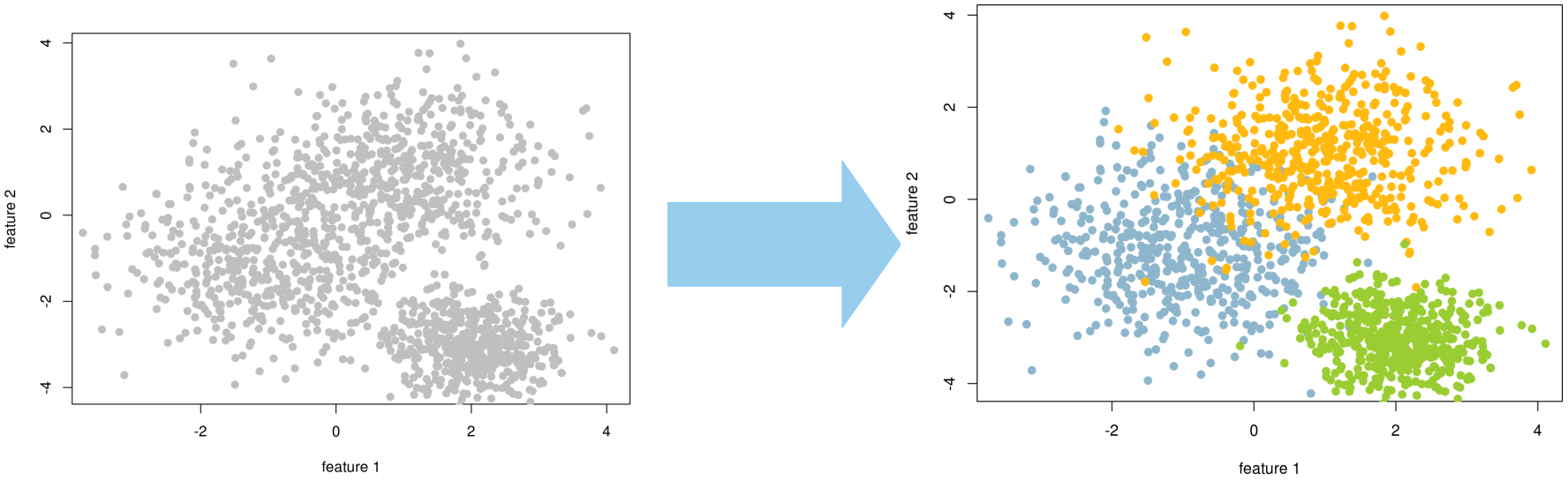
feature1	feature2
3.44	2.66
1.47	-1.26
1.38	2.24
1.09	5.8
1.28	-3.22
2.43	6.72
3.16	0.04
.	.
.	.
.	.
2.13	-0.22

plot



clustering

model



framework

data – transform data into a matrix of numerical feature vectors

optimization – transform problem into an optimization of a loss function over data

generalization – finding models that generalize to unseen data

Optimization framework

optimization

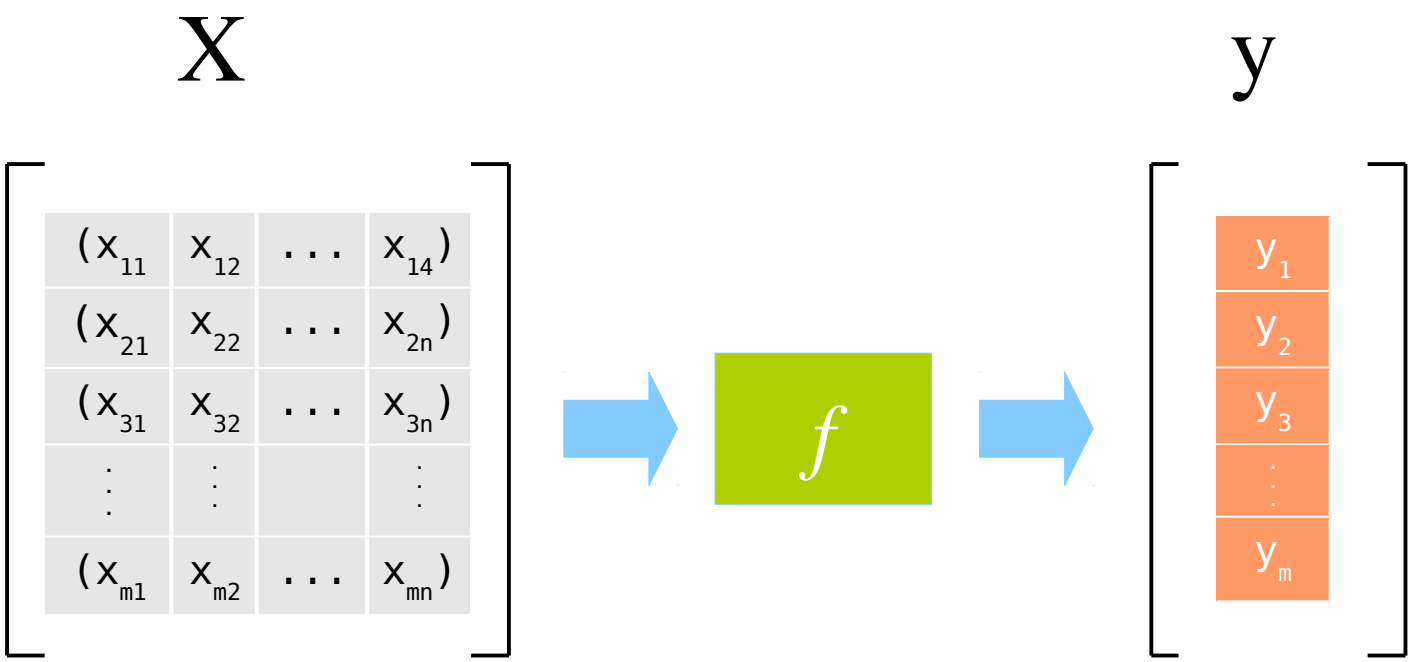
- Find a function that minimizes the error on the data

same as

- Find a function that maximizes the fit on the data

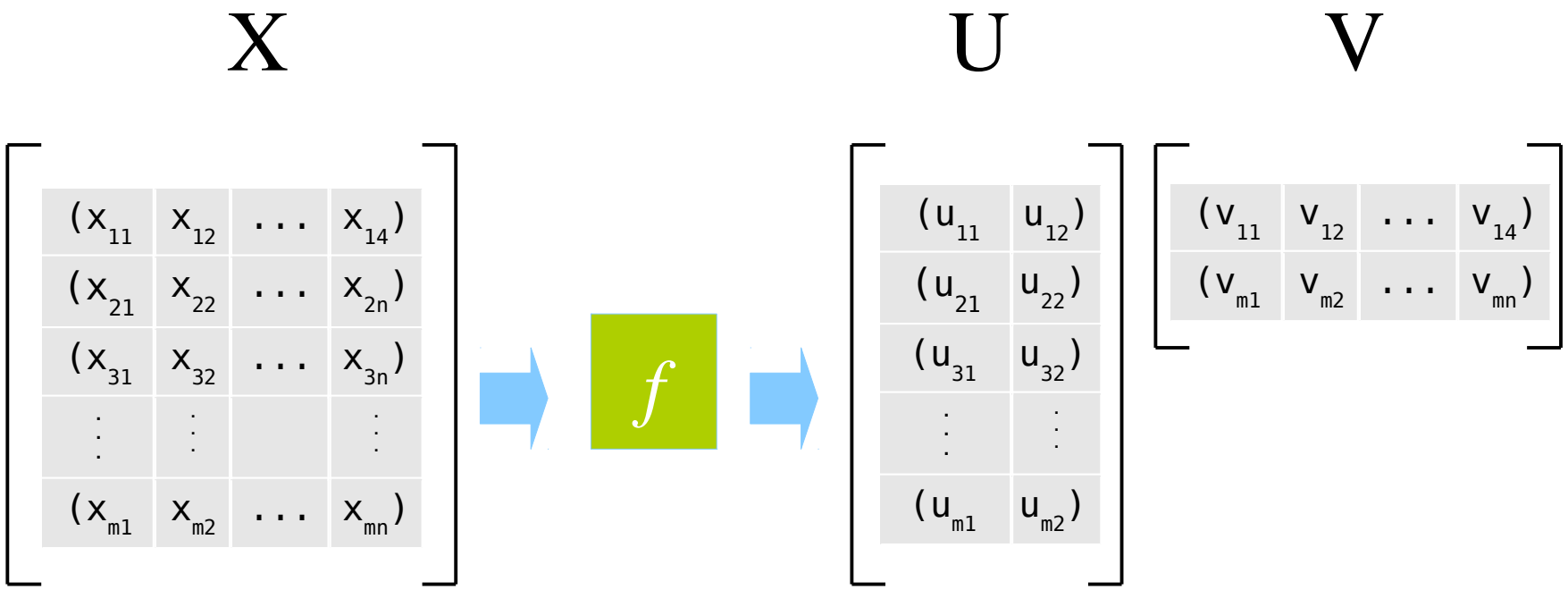
Transform machine learning into a mathematical optimization problem

supervised learning : function approximation



$$f(\mathbf{X}) \approx y$$

unsupervised learning: function approximation



$$X \approx f(X) = U\ V$$

Loss

- Also known as: error, likelihood, risk, cost
- Loss function is chosen to:
  - Represent goals
  - Make optimization more tractable

- **Likelihood**, ℓ , of the data is the most common choice. For a given model and independent identically distributed data (i.i.d):

$$\ell(data|model) = p(data|model) = \prod_{x_i \in data} p(x_i \mid model)$$

minimizing loss

$$w_{min} = \underset{w}{argmin} \sum_{i=1}^{n} Loss(y_i, f_w(x_i))$$

Loss function

learning function

Target variable

parameter weights

surrogate loss

- Notice the 0-1 loss has lots of small flat places. Its is not convex, so using an general purpose optimizer will be very difficult.

- Logistic loss on the other hand is smooth and convex. Gradient based optimizers work well with logistic loss!

- When we do the lab on gradient descent this will make more sense.

- Because of this we use a “surrogate loss” for 0-1loss i.e even though we may be interested in 0-1 loss, we use a smooth convex loss like logistic because it can be solved with existing techniques.

loss functions

model	loss function
linear regression	Quadratic loss : $\sum (f(x) - y)^2$
logistic regression	logistic loss : $\sum \log(1 + e^{-f(x)\cdot y})$
SVM	Hinge loss : $\sum \max(0, 1 - f(x)\cdot y)$
decision tree(CART)	Information gain :
boosting	exponential loss : $\sum e^{-f(x)\cdot y}$

loss functions

All using y ∈ { -1,+1}

Perceptron

$$\boldsymbol{w} = \underset{w}{argmin} \sum_i Max(0 \ , \ -y_i \ <\mathbf{w},\mathbf{x}_i>)$$

$$gradient = \sum_{misclassified} y_i \, \boldsymbol{x}_i$$

Boosting

$$\boldsymbol{w} = \underset{w}{argmin} \sum_i exp(-y_i \ <\mathbf{w},\mathbf{x}_i>)$$

Logistic regression

$$\boldsymbol{w} = \underset{w}{argmin} \sum_i log(1 \ + \ exp(-y_i \ <\mathbf{w},\mathbf{x}_i>))$$

$$gradient = \sum_i \frac{y_i \, \boldsymbol{w}}{(1 \ + \ exp(-y_i \ <\boldsymbol{w},\boldsymbol{x}_i>)}$$

Quadratic loss

$$\boldsymbol{w} = \underset{w}{argmin} \sum_i (y_i \ - \ <\mathbf{w},\mathbf{x}_i>)^2$$

Svm : hinge loss

$$\boldsymbol{w} = \underset{w}{argmin} \sum_i max(0 \ , \ 1 \ - \ y_i \ <\mathbf{w},\mathbf{x}_i>)$$

optimization

- Closed form solution (rare)

  Linear regression, Naive Bayes

- Iterative gradient based (most common)

  linear regression, logistic regression, perceptron

  svm, neural networks

- Expectation maximization

  kmeans, gmm, hmm, bayesian networks

- Heuristic – i.e greedy algorithms

  classification and regression trees, association rules

Generalization Framework

generalization

Two practical ways to help learn models that generalize to new data

Regularization – add regularization to control over-fitting.

Split the data = Train data + Test data. Generalization error will be overestimate of true error if it is based solely on training data

- Fit model on Train, use test to estimate true error rate
- In practice use cross validation or bootstrapping
- With hyper-parameters often we introduce a 3 way split:

data = Train + Validate + Test

regularization

$$w_{min} = \underset{w}{argmin} \sum_{i=1}^{n} Loss(y_i, f_w(x_i)) + \Omega(w)$$

In practice, we add a term to the loss function
To penalize complex models!

regularization for linear models

- ℓ_2 regularization is the most common: *ridge*

$$\Omega(w) = \tfrac{1}{2} \left\| w \right\|^2 = \tfrac{1}{2}\, w \cdot w$$

- ℓ_1 regularization provides sparse solutions : *lasso*

$$\Omega(w) = \sum |w_i|$$

Both ℓ_1 and ℓ_2 control complexity by penalizing large values of w

- Also possible to combine ℓ_1 and ℓ_2: *elastic net*

$$\Omega(w) = \lambda_1 \ell_1 + \lambda_2 \ell_2$$

regularization in nonlinear models

Anything that limits the complexity of the hypothesis space is
Also a form of regularization i.e:

- Degree for polynomial expansions
- Number of hidden neurons in neural networks
- Number of latent factors in Matrix factorizations
- Number of neighbors in neighborhood models
- Tree depth in decision trees
- Number of clusters in clustering models

representer thm

If w can be written as function of margin :

Margin = y_i <$\mathbf{w}$,$\mathbf{x}_i$>

$$w_{min} = \underset{w}{argmin} \sum_{i=1}^{n} Loss(m_i) + ½ \| w \|^2$$

Then :

$$w_{min} = \sum \alpha_i x_i$$

If Loss is differentiable :

$$w_{min} = \frac{1}{\lambda} \sum -y_i \left(\frac{dLoss}{dm} \right) x_i$$

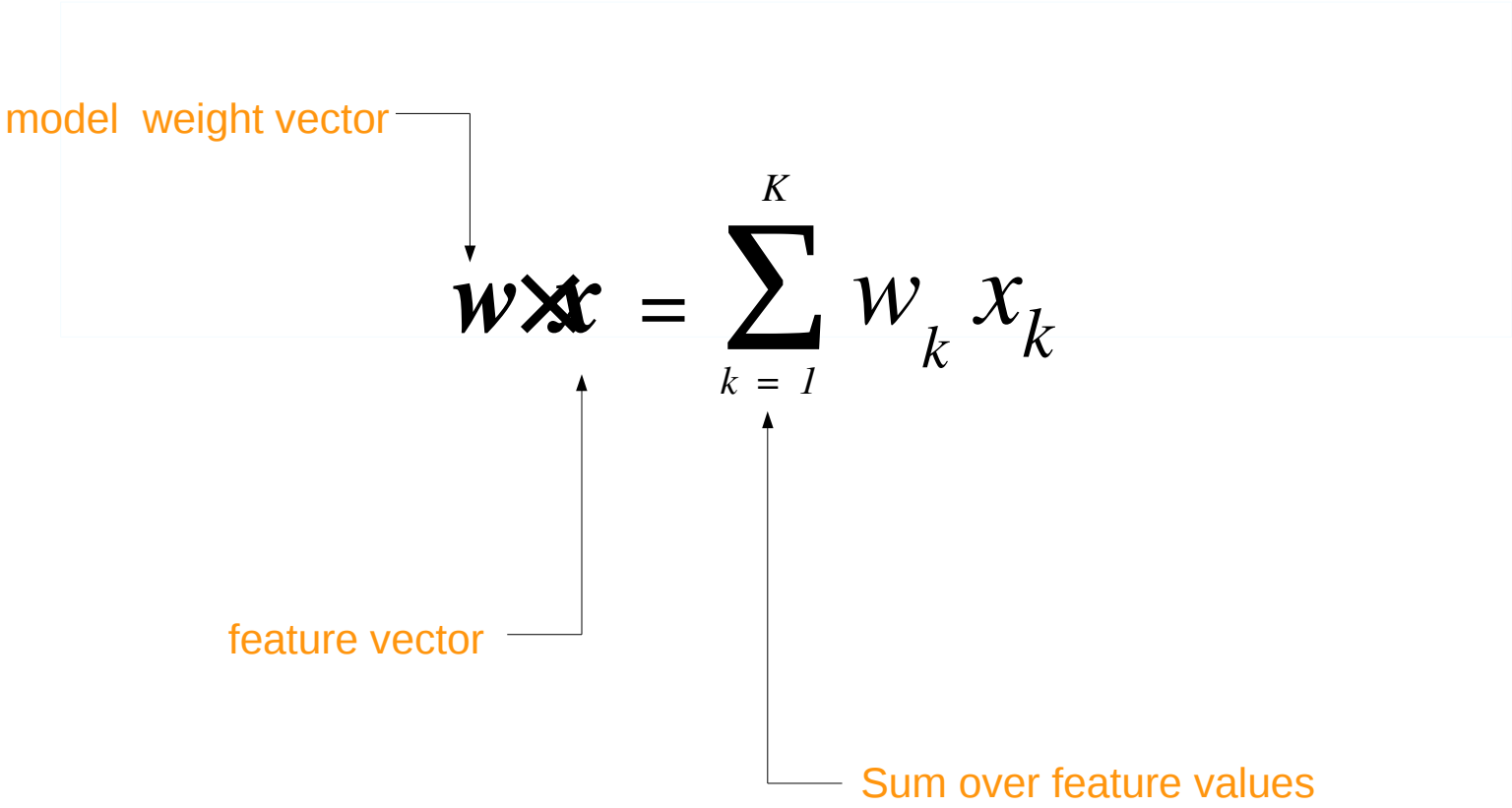
Supervised Learning:Linear Models

linear models

- Most fundamental of ml supervised models
- Continues to be in useful in practice even if in the presence of significant nonlinearity
  - Many nonlinear methods are based on linear models, so its important to really understand them
- With Big Data
  - Linear methods are still the fastest and sometimes the only choice
  - With high dimensional data, often linear models can be on par with nonlinear models

linear models

- Predictions are based on the dot product of weight vector $\boldsymbol{w}$ and feature vector $\boldsymbol{x}$:



linear models

- **Regression**: we can use this directly as estimate of the predicted label

$$f(x) = w \times x$$

- **Classification** : we need to threshold it

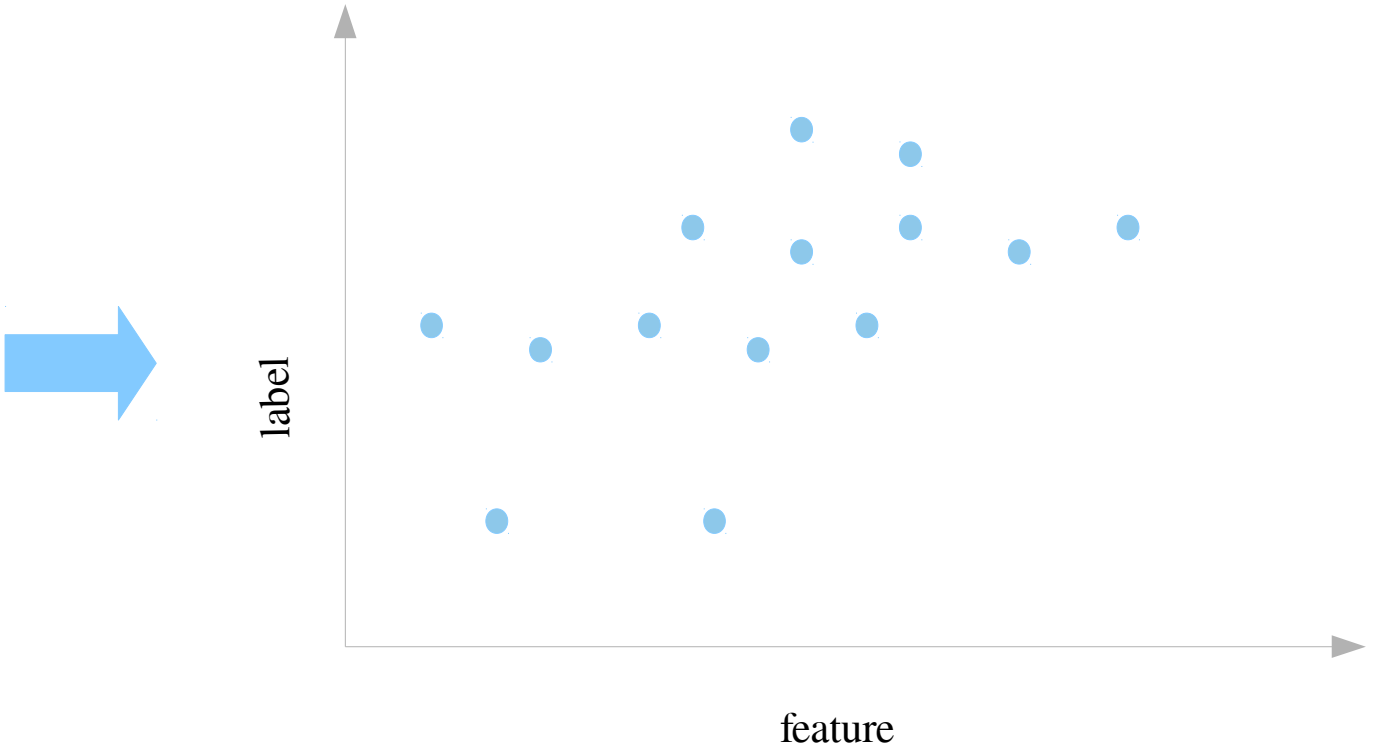
$$f(x) = sign(w \times x) = \begin{cases} 1 \ if \ \ w \times x > 0 \\ 0 \ if \ \ w \times x <= 0 \end{cases}$$

linear regression

Regression problem where we want to predict a continuous label:

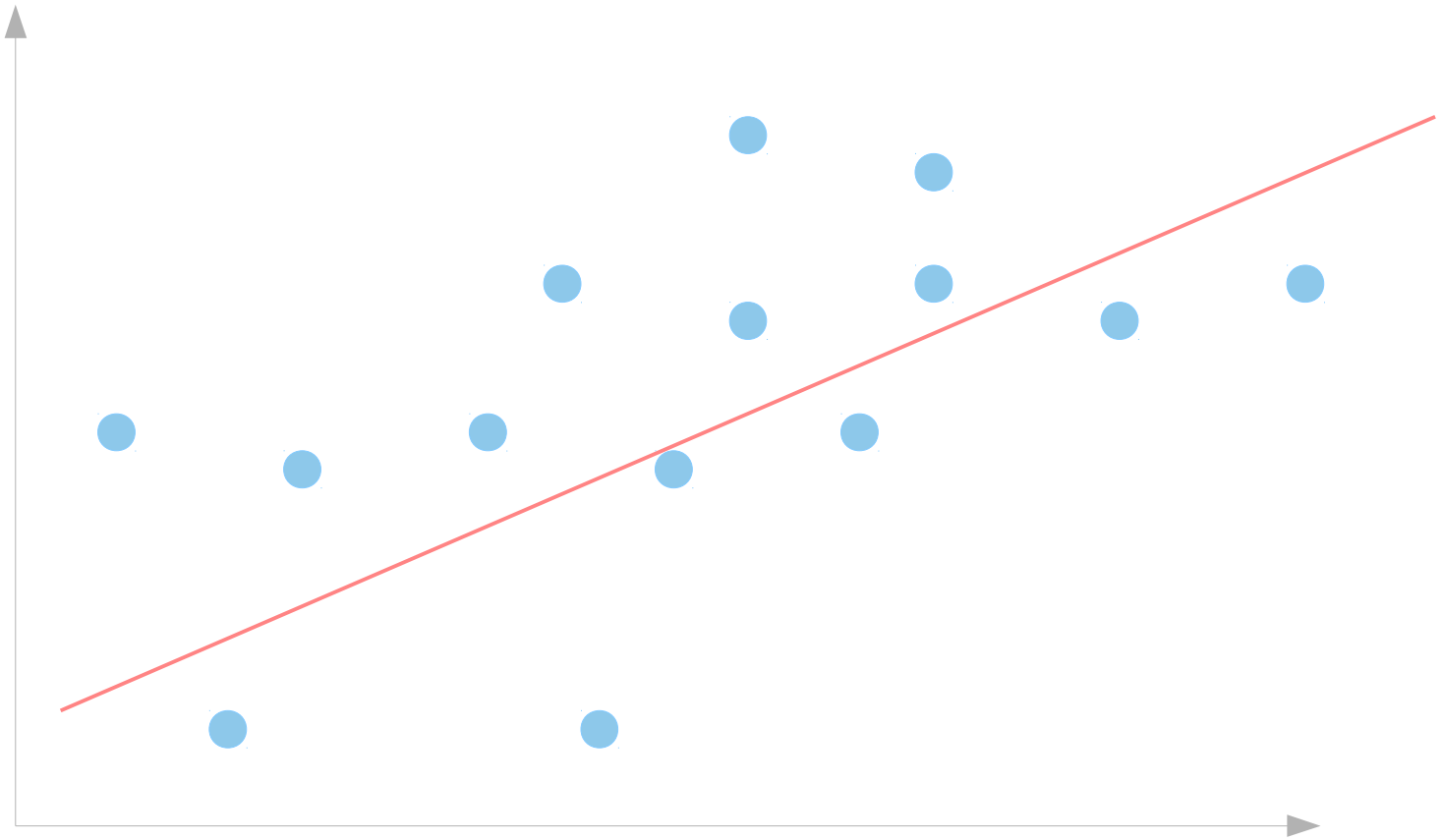
data

feature	label
3.44	16.79
1.47	-4.07
1.38	7.96
1.09	-1.17
1.28	-1.66
2.43	10.52
3.16	14.3
.	.
.	.
.	.
2.11	12.08



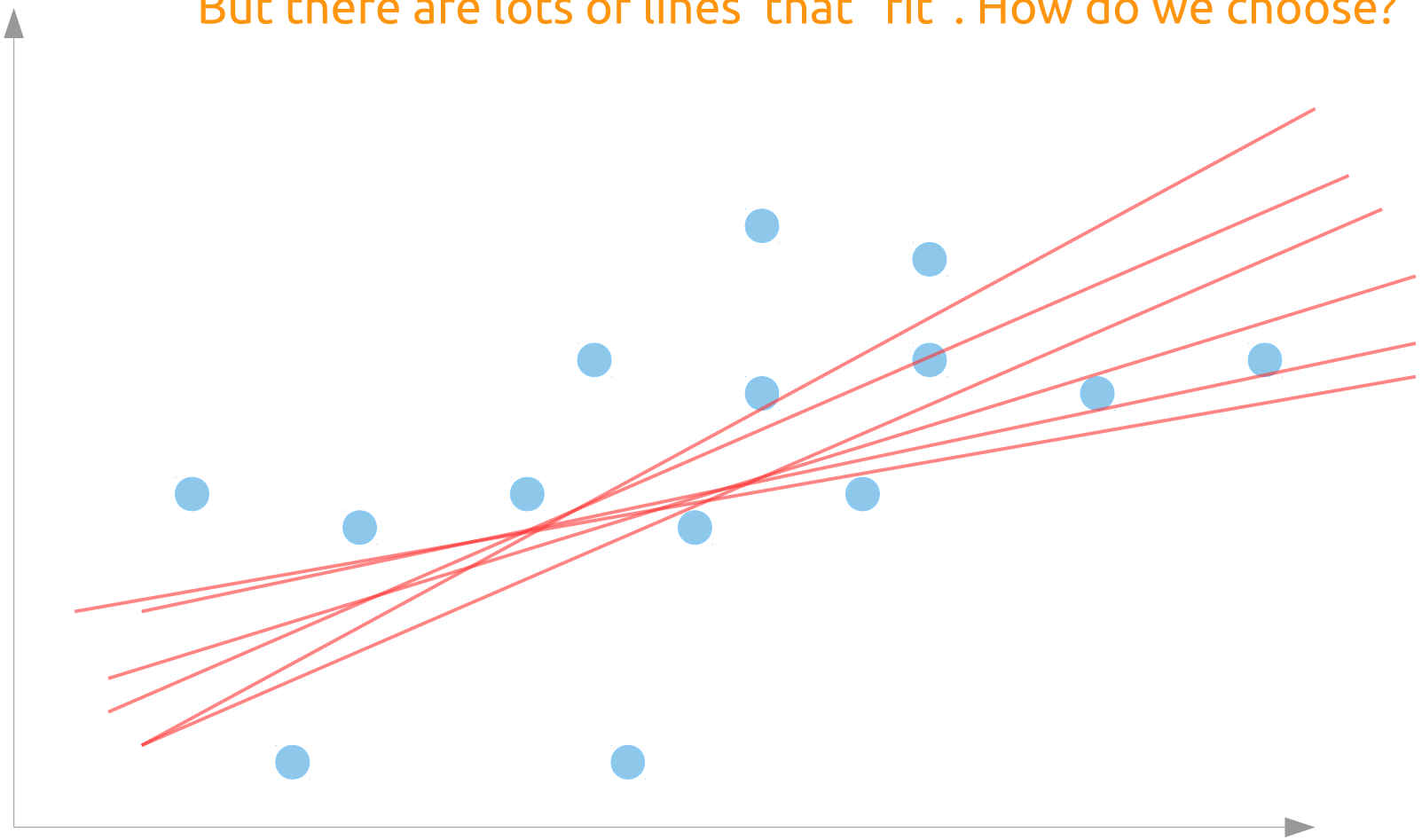
linear regression

We want to find a line that fits our data...



linear regression

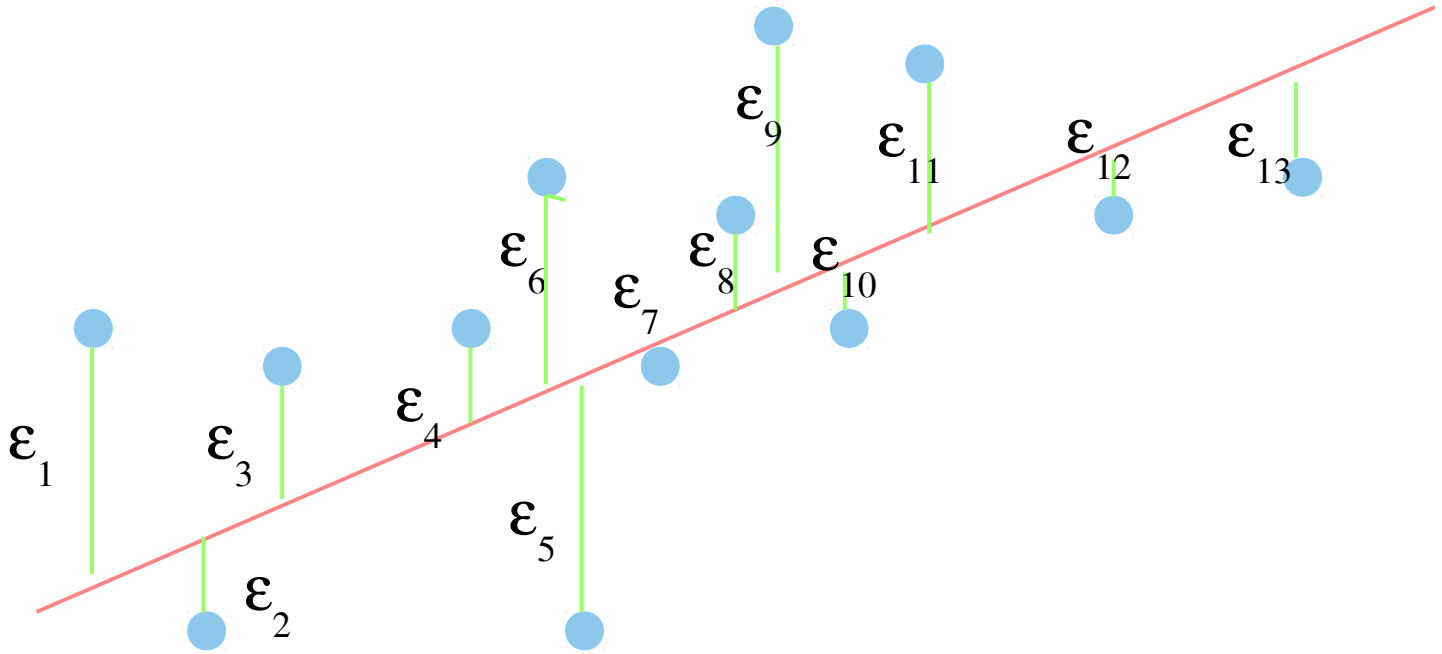
But there are lots of lines that “fit”. How do we choose?



Linear regression

One way is to measure the errors e_i between the line and the points:

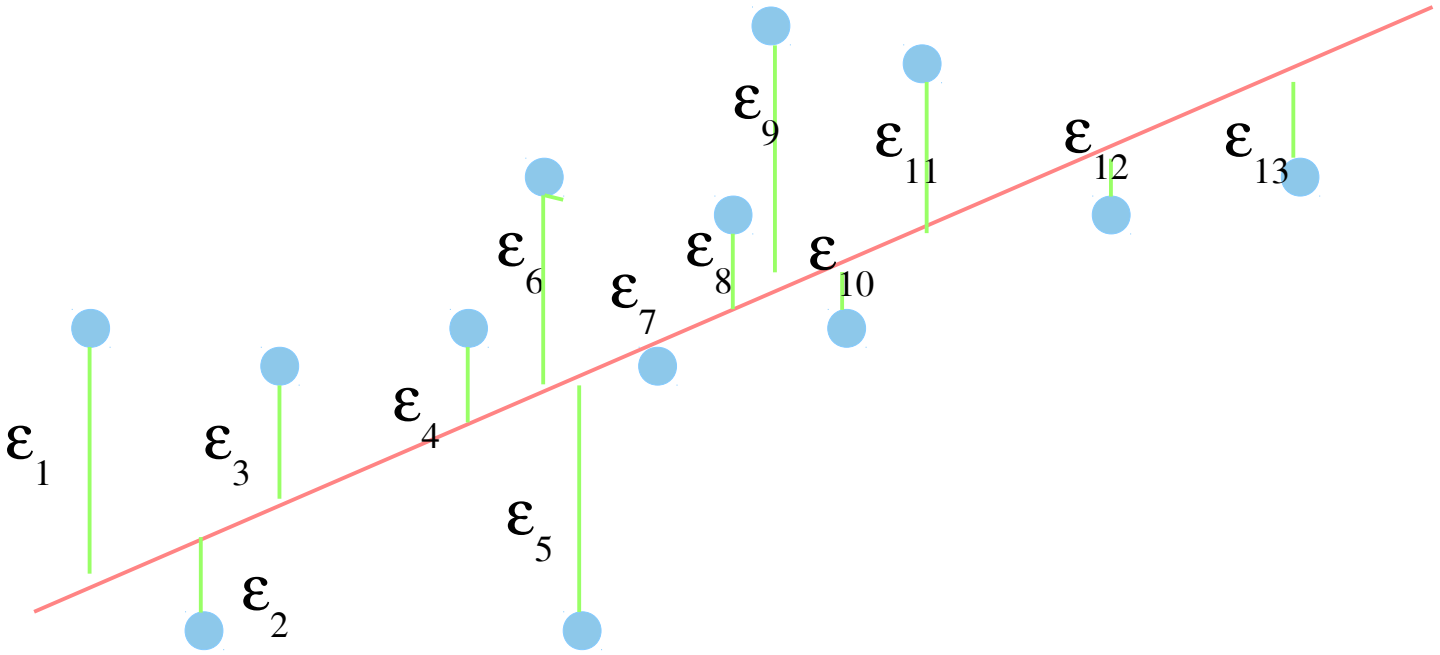
$$\varepsilon_i = wx_i - y_i$$



Linear regression

Define a loss function as the sum of squared errors :

$$loss(w) = \sum_{i=1}^{N} (\varepsilon_i = w \times x_i - y_i)^2$$



linear regression

- Now add the regularization term!

$$loss(w) = \sum_{i=1}^{N} (w \times x_i - y_i)^2 + \lambda \|w\|^2$$

Find least squares estimate of w
Over the training data

Regularization term
Parameter 'λ' needs to be estimated!

linear regression

- Rewrite cost/loss in Matrix form:

$$loss = \mathbf{L}(w) = (\mathbf{X}w - y)^2 + \lambda \|w\|^2$$

- Calculate gradient:

$$\nabla \mathbf{L}(\mathrm{w}) = 2(\mathrm{X}^{\mathrm{t}}(\mathbf{X}w - y) + \lambda w)$$

Sum ov

closed form solution

- Set the gradient equal to zero and solve:

$$\nabla \mathbf{L}(w) = 2(\mathrm{X}^{\mathrm{t}}(\mathrm{X}w - y) + \lambda w) = 0$$

$$w = (\mathrm{X}^{\mathrm{t}}\mathrm{X} + \lambda \mathrm{I}_{\mathrm{K}})^{-1}\mathrm{X}^{\mathrm{t}}\mathrm{y}$$

Ridge regression

$$w = (X^tX + \lambda I_K)^{-1}X^ty$$

- This called the ridge regression solution
- The diagonal term ensures the matrix has full rank and is invertible.
- But to actually compute this we normally don't compute the inverse as it is costly and unstable. Instead solve it using a convenient matrix factorization:
  - QR
  - Cholesky
  - SVD

Iterative solution

- The closed form solution works well and is used in practice, but we will present iterative solution based on gradient descent optimization.

- Advantages of gradient descent over closed form ridge solution:
  - Scales better
  - Can be used in online fashion
  - Introduce a technique that can be used in other ml scenarios

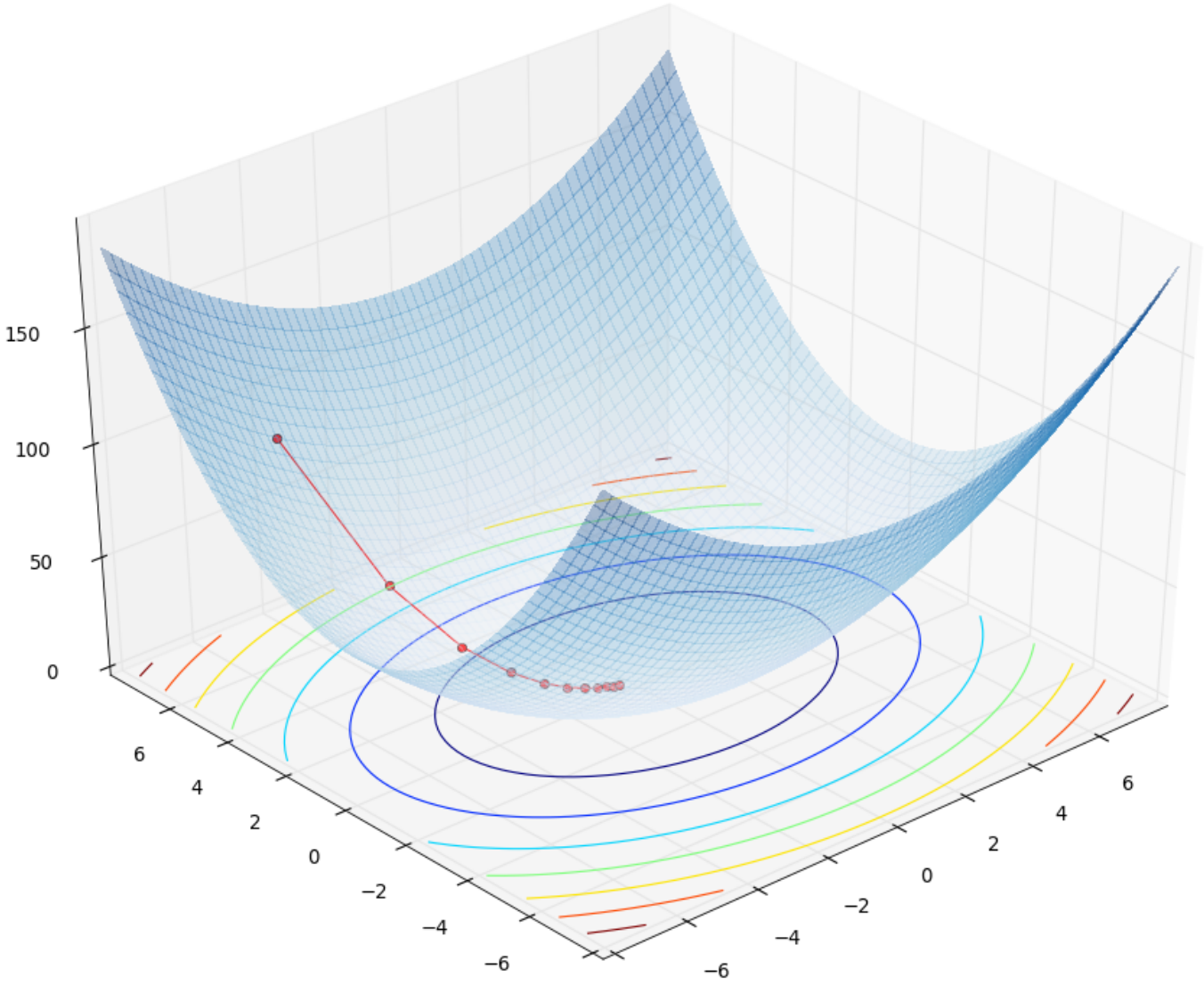
Gradient Descent

gradient descent

Basic idea is simple:

- Start somewhere –-it could be random or all 0s
- Start “climbing down” the cost function i.e update the weight vector in a direction *against* the gradient of the function at that point
- Stop optionally when either:
  - function values are converging
  - gradient is close to zero

gradient descent



gd pseudo-code

η is a learning rate
Step size parameter

$\eta = \eta_0$

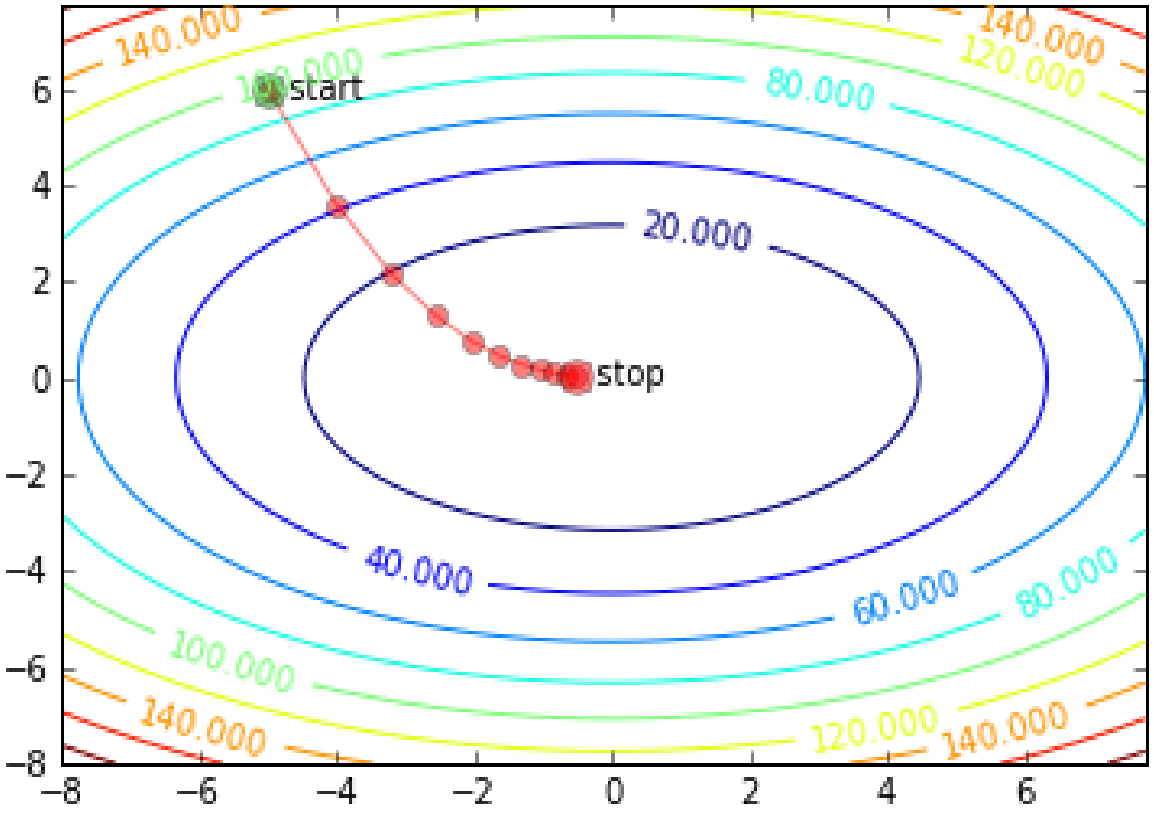
$w = w_0$

for (1 to N) do:

$w = w - \eta \times \nabla f(w)$

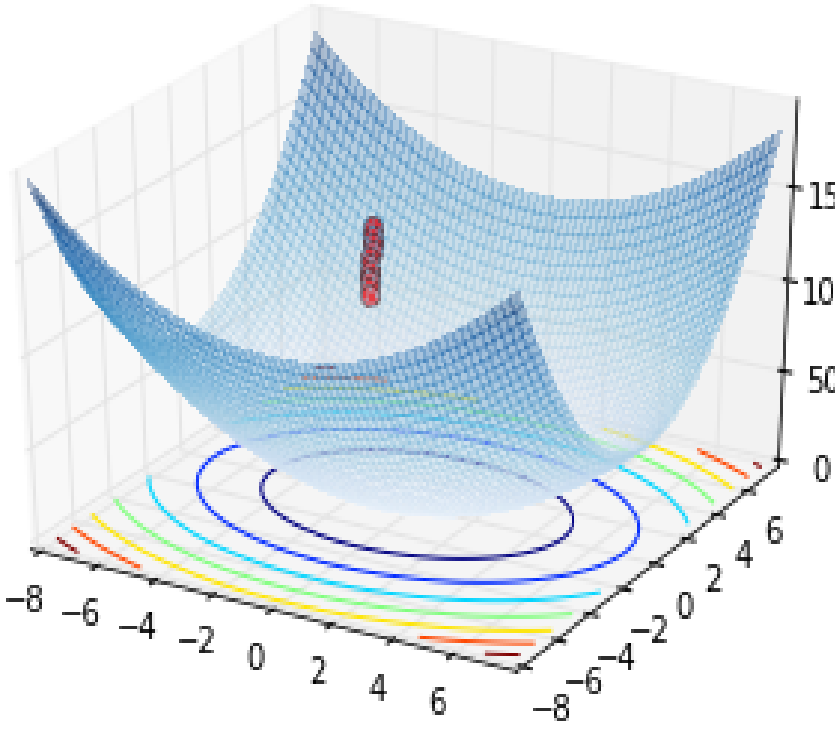
gradient update step

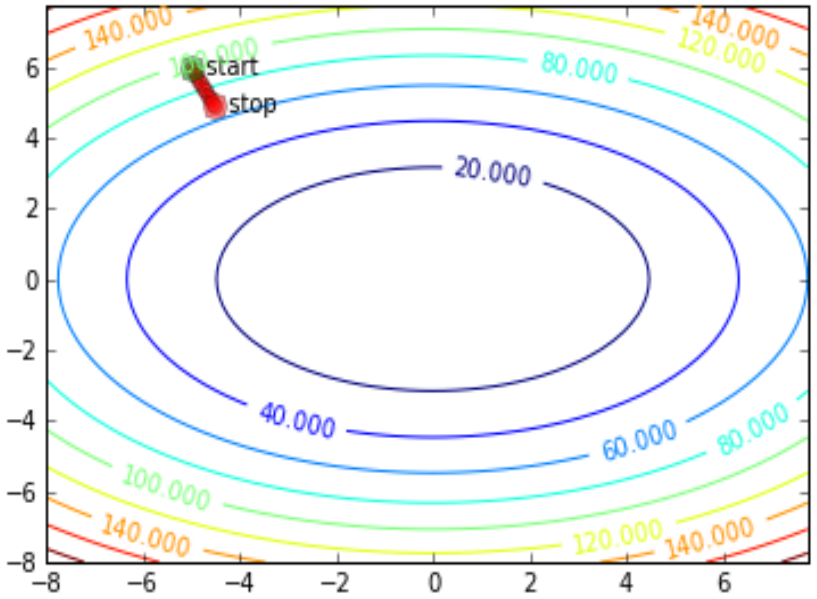
gradient descent



gradient decent gotchas!

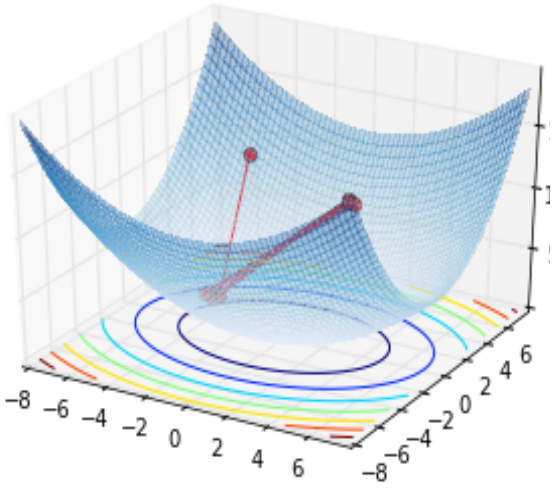
- The learning rate can be problematic. Too small a rate and progress will be very slow:

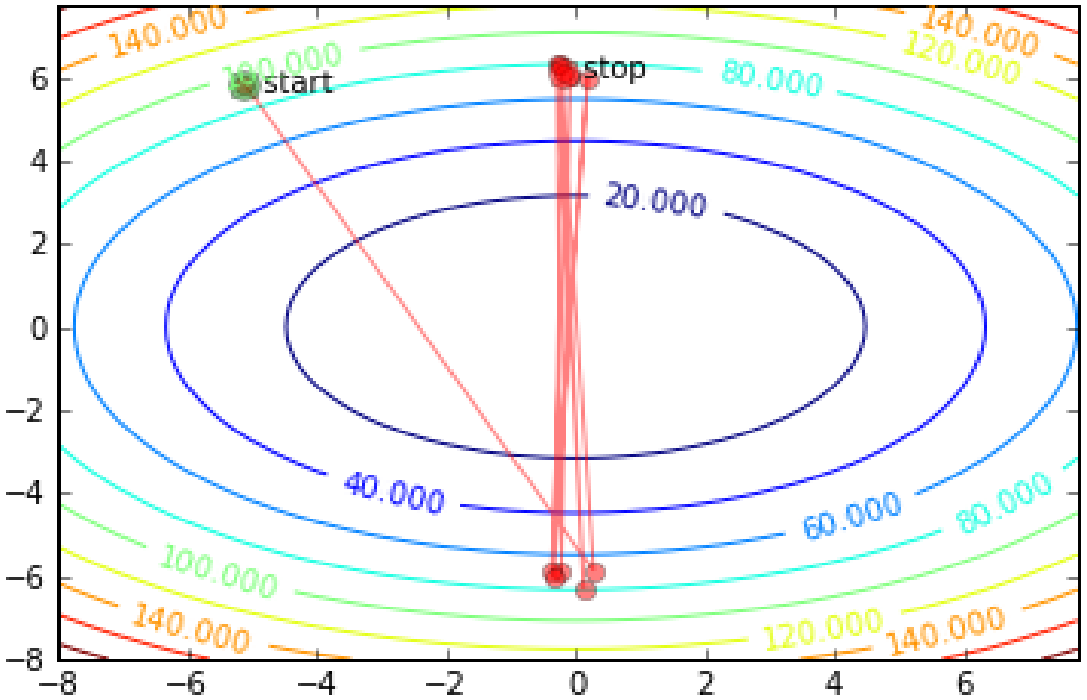




gradient decent gotchas!

- But too large a rate and the iterations may never even converge!





Speeding up gradient descent

- Adaptive step sizes – do a line search to find suitable step size. **Armijo backtracking** is a used in our lab exercises

- **Nesterov acceleration** (see lab notes) has been found to be a very practical speedup as well.

- Even with good step sizes, vanilla gradient descent will waste a lot of time "zig-zagging". Two solutions to this:
    - Adding a "momentum" term that averages in previous search direction
    - **Conjugate-gradient method** is standard fix for this. (see lab notes)

Speeding up gradient descent

- **Newton's method** is a gradient descent method that uses the curvature information in the *hessian*, $H = \nabla^2 f$, to eliminate the need for a step-size parameter as well as converging in a single step for quadratics!

- The update rule is: $w = w - H^{-1}\nabla f(x)$

- The big problem is that it does not scale well. To find a happy medium, **quasi newton** methods exist which approximate the hessian (See **L-BFGS** in the lab exercises)

Back to linear regression...

- Now we can use gradient descent to estimate the model weights :

$$w = w - \eta \left(X^t(\hat{y} - y) + \lambda w \right)$$

learning rate

predicted values = X*w*