CS-671: Deep Learning and its Applications
Lecture: 02 (b)
Basics of Machine Learning: Linear Classification (II)

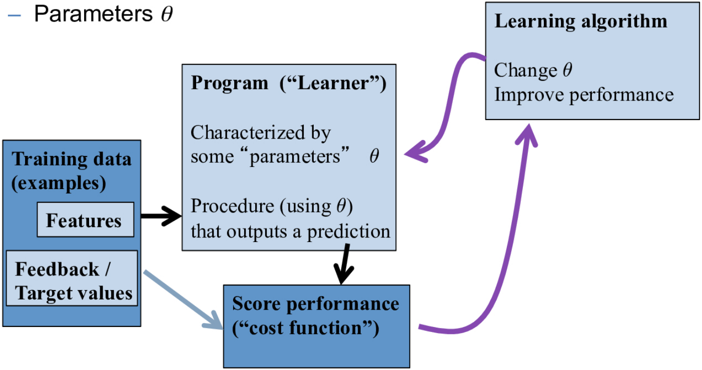
Aditya Nigam, Assistant Professor
School of Computing and Electrical Engineering (SCEE)
Indian Institute of Technology, Mandi
http://faculty.iitmandi.ac.in/ãditya/ aditya@iitmandi.ac.in

Presentation for CS-671@IIT Mandi (21 February, 2019)
(*Slides Credit : Intro to Machine Learning by Alexander Ihler)
https://www.youtube.com/watch?v=qPhMX0vb6D8&list=PLaXDtXvwY-oDvedS3f4HW0b4KxqpJ_imw

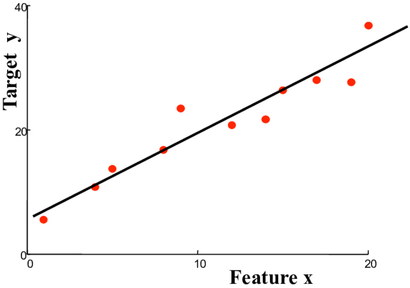
February - May, 2019

Supervised learning

- Notation
  - Features x
  - Targets y
  - Predictions $\hat{y}$
  - Parameters θ



Linear regression



"Predictor":
Evaluate line:

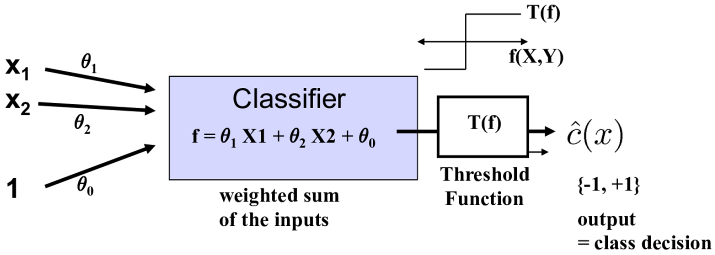
$$r = \theta_0 + \theta_1 x_1$$

return r

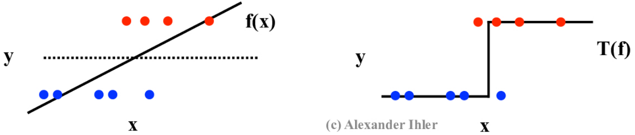
- Contrast with classification
  - Classify: predict discrete-valued target y

(c) Alexander Ihler

Perceptron Classifier (2 features)



Visualizing for one feature "x":



Perceptrons

- Perceptron = a linear classifier
  - The parameters θ are sometimes called weights ("w")
    - real-valued constants (can be positive or negative)
  - Define an additional constant input "1"

- A perceptron calculates 2 quantities:
  - 1. A weighted sum of the input features
  - 2. This sum is then thresholded by the T(.) function

- Perceptron: a simple artificial model of human neurons
    - weights = "synapses"
    - threshold = "neuron firing"

Notation

- Inputs:
  - $x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_n,$
  - $x_1, x_2, \ldots\ldots\ldots\ldots, x_{n-1}, x_n$ are the values of the n features
  - $x_0 = 1$ (a constant input)
  - $\underline{x} = [[x_0, x_1, x_2, \ldots\ldots\ldots\ldots, x_n]]$: feature vector (row vector)
- Weights (parameters):
  - $\theta_0, \theta_1, \theta_2, \ldots\ldots\ldots\ldots, \theta_n,$
  - we have n+1 weights: one for each feature + one for the constant
  - $\underline{\theta} = [[\theta_0, \theta_1, \theta_2, \ldots\ldots\ldots\ldots, \theta_n]]$: parameter vector (row vector)
- Linear response
  - $\theta_0 x_0 + \theta_1 x_1 + \ldots\ \theta_n x_n = \underline{x} \cdot \underline{\theta}'$ then threshold

```
F = X.dot( theta.T );        # compute linear response
Yhat = np.sign(F)            # predict class +1 or -1
Yhat = 2*(F>0)-1             # manual "sign" of F
```

Perceptron Decision Boundary

- The perceptron is defined by the decision algorithm:
- The perceptron represents a hyperplane decision surface in d-dimensional space
  - A line in 2D, a plane in 3D, etc.
- The equation of the hyperplane is given by

This defines the set of points that are on the boundary.

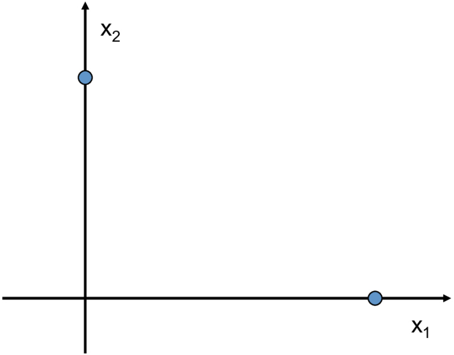
$$o(x_1, x_2, \ldots, x_d, x_{d+1}) \begin{cases} = 1 & (\text{if } \underline{\theta} \cdot \underline{\mathbf{x}}' > 0) \\ = -1 & (\text{otherwise}) \end{cases}$$

$$\underline{\theta} \cdot \underline{\mathbf{x}}' = 0$$

(c) Alexander Ihler

Example, Linear Decision Boundary

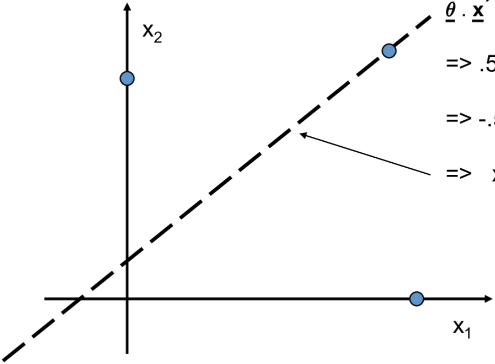
$$\underline{\theta} = (\theta_0, \quad \theta_1, \quad \theta_2) = (1, \quad .5, \quad -.5)$$



From P. Smyth

Example, Linear Decision Boundary

$$\underline{\theta} = (\theta_0, \quad \theta_1, \quad \theta_2) = (1, \quad .5, \quad -.5)$$



$\underline{\theta} \cdot \underline{x}' = 0$

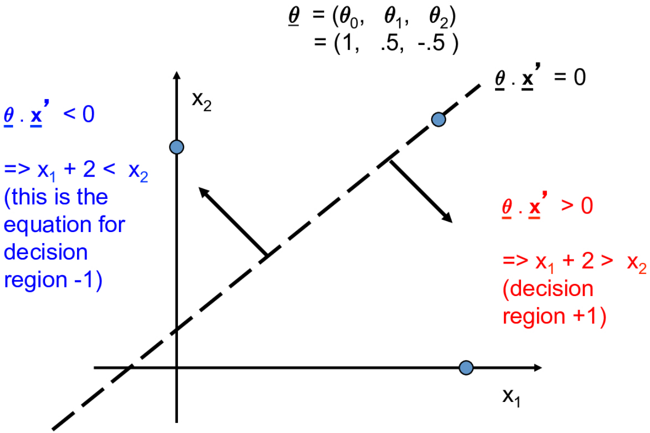
$\Rightarrow .5 \cdot x_1 - .5 \cdot x_2 + 1 \cdot 1 = 0$

$\Rightarrow -.5\, x_2 = -.5\, x_1 - 1$

$\Rightarrow \quad x_2 = x_1 + 2$

From P. Smyth

Example, Linear Decision Boundary

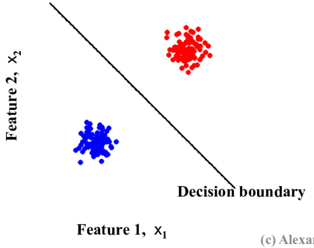


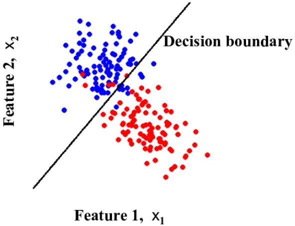
From P. Smyth

Separability

- A data set is separable by a learner if
  - There is some instance of that learner that correctly predicts all the data points
- Linearly separable data
  - Can separate the two classes using a straight line in feature space
  - in 2 dimensions the decision boundary is a straight line

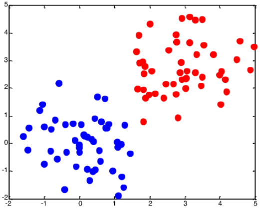
Linearly separable data | Linearly non-separable data





(c) Alexander Ihler

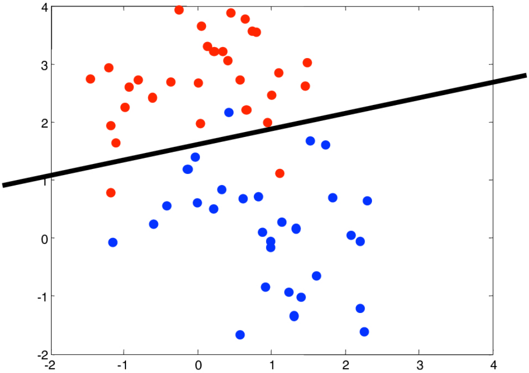
Class overlap



- Classes may not be well-separated
- Same observation values possible under both classes
  - High vs low risk; features {age, income}
  - Benign/malignant cells look similar
  - …
- Common in practice
- May not be able to perfectly distinguish between classes
  - Maybe with more features?
  - Maybe with more complex classifier?
- Otherwise, may have to accept some errors

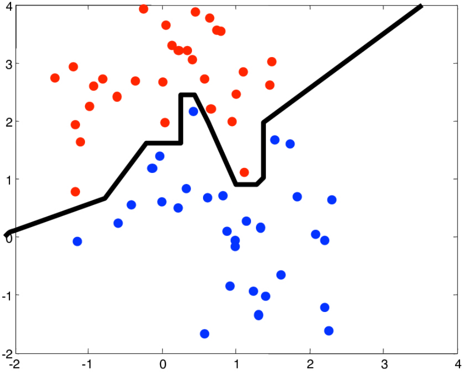
(c) Alexander Ihler

Another example



(c) Alexander Ihler

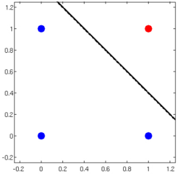
Non-linear decision boundary

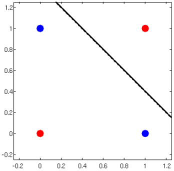


(c) Alexander Ihler

Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)





Adding features

- Linear classifier can't learn some functions

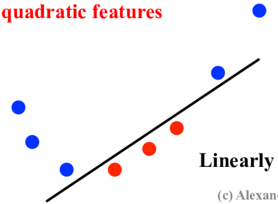
1D example:

$$y = T(\, b\, x + c\,)$$

Not linearly separable

Add quadratic features

$$y = T(\, a\, x^2 + b\, x + c\,)$$



Linearly separable in new features…

(c) Alexander Ihler

Adding features

- Linear classifier can't learn some functions

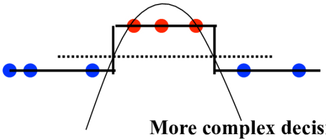
1D example:

$y = T(b\,x + c)$

Not linearly separable

Quadratic features, visualized in original feature space:

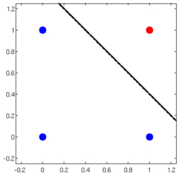
$y = T(a\,x^2 + b\,x + c)$

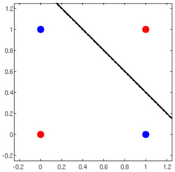


More complex decision boundary: $ax^2+bx+c = 0$

Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)



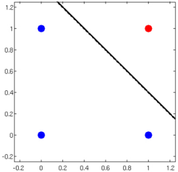


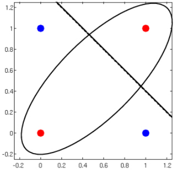
What kinds of functions would we need to learn the data on the right?

(c) Alexander Ihler

Representational Power of Perceptrons

- What mappings can a perceptron represent perfectly?
  - A perceptron is a linear classifier
  - thus it can represent any mapping that is linearly separable
  - some Boolean functions like AND (on left)
  - but not Boolean functions like XOR (on right)





What kinds of functions would we need to learn the data on the right?

Ellipsiodal decision boundary: $a\,x_1^2 + b\,x_1 + c\,x_2^2 + d\,x_2 + e\,x_1x_2 + f = 0$

(c) Alexander Ihler

Feature representations

- Features are used in a linear way
- Learner is dependent on representation

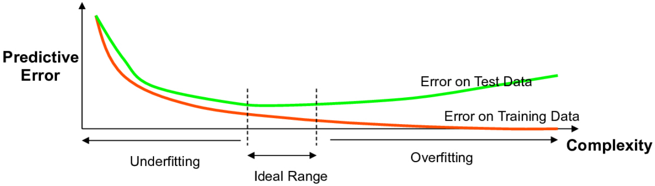
- Ex: discrete features
  - Mushroom surface: {fibrous, grooves, scaly, smooth}
  - Probably not useful to use x = {1, 2, 3, 4}
  - Better: 1-of-K, x = { [1000], [0100], [0010], [0001] }
  - Introduces more parameters, but a more flexible relationship

Effect of dimensionality

- Data are increasingly separable in high dimension – is this a good thing?

- "Good"
  - Separation is easier in higher dimensions (for fixed # of data m)
  - Increase the number of features, and even a linear classifier will eventually be able to separate all the training examples!

- "Bad"
  - Remember training vs. test error? Remember overfitting?
  - Increasingly complex decision boundaries can eventually get all the training data right, but it doesn't necessarily bode well for test data…



Summary

- Linear classifier ⇔ perceptron
- Linear decision boundary
  - Computing and visualizing
- Separability
  - Limits of the representational power of a perceptron
- Adding features
  - Interpretations
  - Effect on separability
  - Potential for overfitting

Linear classification: Learning

Learning the Classifier Parameters

- Learning from Training Data:
  - training data = labeled feature vectors
  - Find parameter values that predict well (low error)
    - error is estimated on the training data
    - "true" error will be on future test data

- Define an objective function $J(\underline{\theta})$:
  - Classifier accuracy (for a given set of weights $\underline{\theta}$ and labeled data)

- Maximize this objective function (or, minimize error)
  - An optimization or search problem over the vector $(\theta_1, \theta_2, \theta_0)$

Training a linear classifier

- How should we measure error?
  - Natural measure = "fraction we get wrong" (error rate)

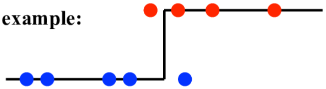
$$\text{err}(\underline{\theta}) = 1/m \sum \delta(\, \hat{y}(i) \neq y(i) \,)$$

where $\delta(\, \hat{y}(i) \neq y(i) \,) = 0$ if $\hat{y}(i) = y(i)$, and 1 otherwise

```
Yhat = np.sign( X.dot( theta.T ) );        # predict class
err = np.mean( Y != Yhat )                 # count errors: empirical error rate
```

- But, hard to train via gradient descent
  - Not continuous
  - As decision boundary moves, errors change abruptly

1D example:

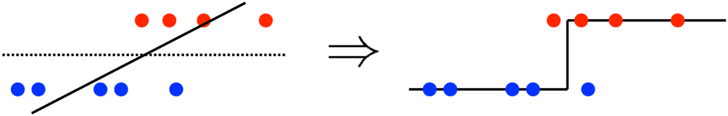


T(f) = -1 if f < 0
T(f) = +1 if f > 0

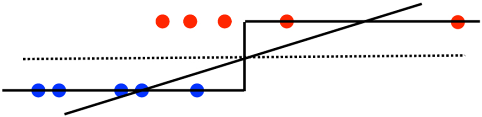
(c) Alexander Ihler

Linear regression?

- Simple option: set θ using linear regression



- In practice, this often doesn't work so well...
  - Consider adding a distant but "easy" point
  - MSE distorts the solution



Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

  While (~done)

  For each data point j:

  $\hat{y}(j) = T(\underline{\theta} * \underline{x}(j))$: predict output for data point j

  $\underline{\theta} \leftarrow \underline{\theta} + \alpha\,(y(j) - \hat{y}(j))\,\underline{x}(j)$: "gradient-like" step

- Compare to linear regression + MSE cost
  - Identical update to SGD for MSE except error uses thresholded $\hat{y}(j)$ instead of linear response $\underline{\theta}\,x'$ so:
  - (1) For correct predictions, $y(j) - \hat{y}(j) = 0$
  - (2) For incorrect predictions, $y(j) - \hat{y}(j) = \pm 2$

"adaptive" linear regression: correct predictions stop contributing

Perceptron algorithm

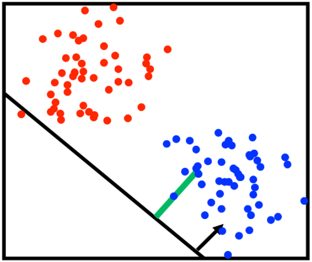
- Perceptron algorithm: an SGD-like algorithm

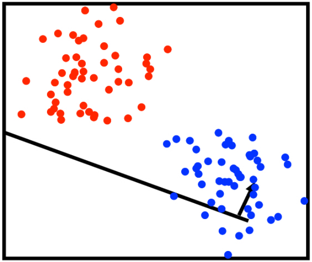
  While (~done)

  For each data point j:

  $\hat{y}(j) = T(\underline{\theta} * \underline{x}(j))$: predict output for data point j

  $\underline{\theta} \leftarrow \underline{\theta} + \alpha\,(y(j) - \hat{y}(j))\,\underline{x}(j)$: "gradient-like" step





(c) Alexander Ihler

Perceptron algorithm

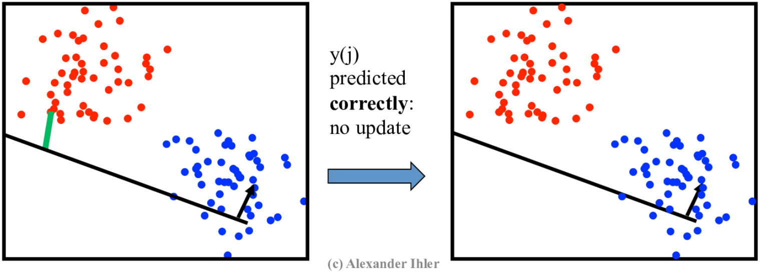
- Perceptron algorithm: an SGD-like algorithm

  While (~done)

  For each data point j:

  $\hat{y}(j) = T(\underline{\theta} * \underline{x}(j))$: predict output for data point j

  $\underline{\theta} \leftarrow \underline{\theta} + \alpha\,(y(j) - \hat{y}(j))\,\underline{x}(j)$: "gradient-like" step



Perceptron algorithm

- Perceptron algorithm: an SGD-like algorithm

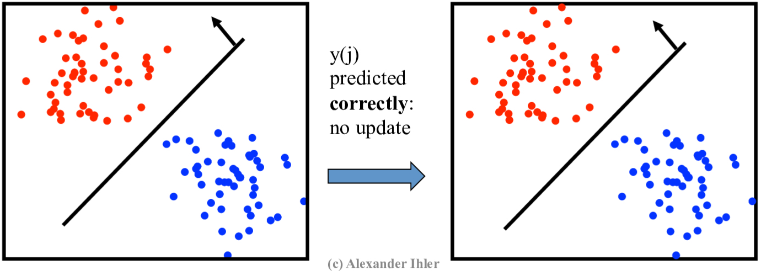
  While (~done)

  For each data point j:

  $\hat{y}(j) = T(\underline{\theta} * \underline{x}(j))$: predict output for data point j

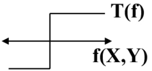
  $\underline{\theta} \leftarrow \underline{\theta} + \alpha\,(y(j) - \hat{y}(j))\,\underline{x}(j)$: "gradient-like" step

  (Converges if data are linearly separable)

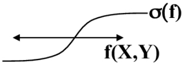


Surrogate loss functions

- Another solution: use a "smooth" loss
  - e.g., approximate the threshold function



  - Usually some smooth function of distance
    - Example: "sigmoid", looks like an "S"



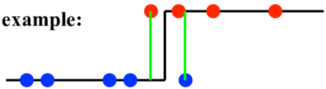
  - Now, measure e.g. MSE

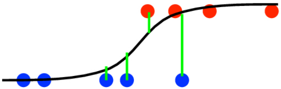
$$J(\underline{\theta}) = \frac{1}{m} \sum_j \left(\sigma(f(x^{(j)})) - y^{(j)} \right)^2$$

Class y = {0, 1} …

  - Far from the decision boundary: |f(.)| large, small error
  - Nearby the boundary: |f(.)| near 1/2, larger error

1D example:



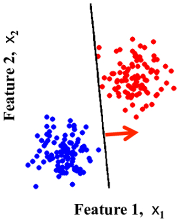


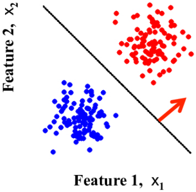
Classification error = 2/9

MSE = $(0^2 + 1^2 + .2^2 + .25^2 + .05^2 + \ldots)/9$

Beyond misclassification rate

- Which decision boundary is "better"?
  - Both have zero training error (perfect training accuracy)
  - But, one of them seems intuitively better…



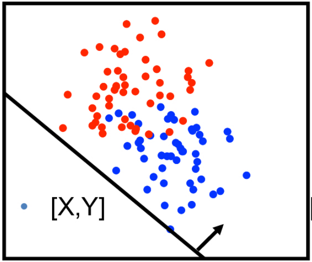


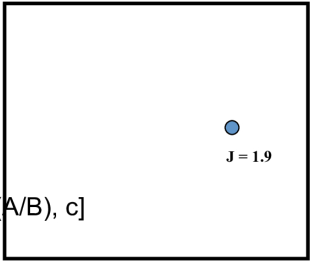
- Side benefit of "smoothed" error function
  - Encourages data to be far from the decision boundary
  - See more examples of this principle later...

Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of f(X1,X2) = a*X1 + b*X2 + c

- Example: 2D feature space ⇔ parameter space





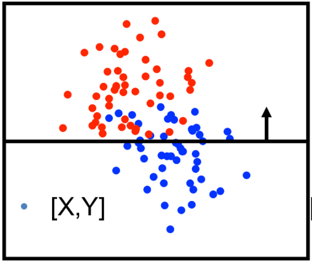
- [X,Y] [arctan(A/B), c]

(c) Alexander Ihler

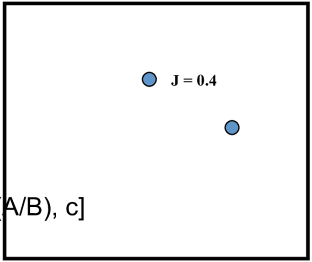
Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of f(X1,X2) = a*X1 + b*X2 + c

- Example: 2D feature space ⇔ parameter space



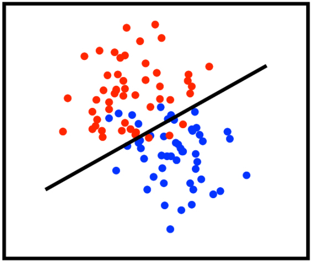
- [X,Y] [arctan(A/B), c]

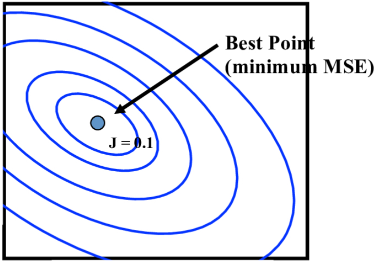


(c) Alexander Ihler

Training the Classifier

- Once we have a smooth measure of quality, we can find the "best" settings for the parameters of
  f(X1,X2) = a*X1 + b*X2 + c
- Finding the minimum loss J(.) in parameter space…





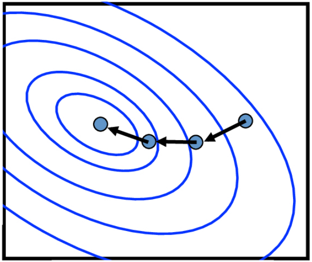
(c) Alexander Ihler

Finding the Best MSE

- As in linear regression, this is now just optimization

- Methods:
  - Gradient descent
    - Improve loss by small changes in parameters ("small" = learning rate)
  - Or, substitute your favorite optimization algorithm…
    - Coordinate descent
    - Stochastic search
    - Genetic algorithms

Gradient Descent



Gradient Equations

- MSE (note, depends on function $\sigma(.)$)

$$J(\underline{\theta} = [a, b, c]) = \frac{1}{m} \sum_i (\sigma(ax_1^{(i)} + bx_2^{(i)} + c) - y^{(i)})^2$$

- What's the derivative with respect to one of the parameters?

$$\frac{\partial J}{\partial a} = \frac{1}{m} \sum_i 2 \boxed{(\sigma(\theta \cdot x^{(i)}) - y^{(i)})} \boxed{\partial \sigma(\theta \cdot x^{(i)}) \; x_1^{(i)}}$$

Error between class and prediction

Sensitivity of prediction to changes in parameter "a"

- Similar for parameters b, c [replace x_1 with x_2 or 1 (constant)]

Saturating Functions

- Many possible "saturating" functions
- "Logistic" sigmoid (scaled for range [0,1]) is

$$\sigma(z) = 1 / (1 + \exp(-z))$$

- Derivative is

$$\partial\sigma(z) = \sigma(z)\,(1-\sigma(z))$$

(to predict:
threshold z at 0 or
threshold $\boldsymbol{\sigma}$ (z) at ½)

- Python Implementation:

```
def sig(z):                                  # logistic sigmoid
    return  1.0 / (1.0 + np.exp(-z) ) #   in [0,1]

def dsig(z):                                 # its derivative at z
    return  sig(z) * (1-sig(z))
```

For range [-1 , +1]:

$$\rho(z) \;\; = 2\,\sigma(z) - 1$$

$$\partial\rho(z) = 2\,\sigma(z)\,(1-\sigma(z))$$

Predict: threshold z or ρ at zero

Logistic regression

- Intepret $\sigma(\underline{\theta}$ x') as a probability that y = 1
- Use a negative log-likelihood loss function
  - If y = 1, cost is - log Pr[y=1] = - log $\sigma(\underline{\theta}$ x')
  - If y = 0, cost is - log Pr[y=0] = - log (1 - $\sigma(\underline{\theta}$ x'))

- Can write this succinctly:

$$J(\underline{\theta}) = -\frac{1}{m}\sum_i \underbrace{y^{(i)} \log \sigma(\theta \cdot x^{(i)})}_{\text{Nonzero only if y=1}} + \underbrace{(1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))}_{\text{Nonzero only if y=0}}$$

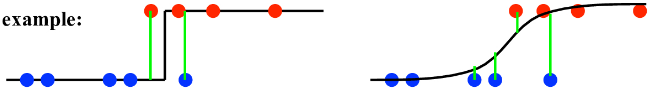
Logistic regression

- Intepret $\sigma(\underline{\theta}\, x')$ as a probability that y = 1
- Use a negative log-likelihood loss function
  - If y = 1, cost is - log Pr[y=1] = - log $\sigma(\underline{\theta}\, x')$
  - If y = 0, cost is - log Pr[y=0] = - log (1 - $\sigma(\underline{\theta}\, x')$)
- Can write this succinctly:

$$J(\underline{\theta}) = -\frac{1}{m}\sum_i y^{(i)} \log \sigma(\theta \cdot x^{(i)}) + (1-y^{(i)}) \log(1-\sigma(\theta \cdot x^{(i)}))$$

- Convex! Otherwise similar: optimize J(θ) via …

1D example:



Classification error = MSE = 2/9

NLL = - (log(.99) + log(.97) + …)/9

Gradient Equations

- Logistic neg-log likelihood loss:

$$J(\underline{\theta}) = -\frac{1}{m} \sum_i y^{(i)} \log \sigma(\theta \cdot x^{(i)}) + (1 - y^{(i)}) \log(1 - \sigma(\theta \cdot x^{(i)}))$$

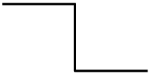
- What's the derivative with respect to one of the parameters?

$$\frac{\partial J}{\partial a} = -\frac{1}{m} \sum_i y^{(i)} \frac{1}{\sigma(\theta \cdot x^{(i)})} \; \partial\sigma(\theta \cdot x^{(i)}) \; x_1^{(i)} + (1 - y(i)) \ldots$$

$$= -\frac{1}{m} \sum_i y^{(i)} (1 - \sigma(\theta \cdot x^{(i)})) \; x_1^{(i)} - (1 - y^{(i)}) \ldots$$

Surrogate loss functions

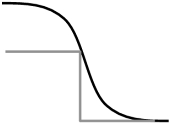
- Replace 0/1 loss
  with something easie $\Delta_i(\theta) = \delta\big(T(\theta x^{(i)}) \neq y^{(i)}\big)$



0/1 Loss

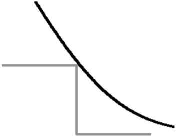
- Logistic MSE

$$J_i(\theta) = 4\big(\sigma(\theta x^{(i)}) - y^{(i)}\big)^2$$



- Logistic Neg Log Likelihood

$$J_i(\underline{\theta}) = -\frac{y^{(i)}}{\log 2} \log \sigma(\theta \cdot x^{(i)}) + \ldots$$



Summary

- Linear classifier ⇔ perceptron
- Measuring quality of a decision boundary
  - Error rate (0/1 loss)
  - Logistic sigmoid + MSE criterion
  - Logistic Regression
- Learning the weights of a linear classifer from data
  - Reduces to an optimization problem
  - Perceptron algorithm
  - For MSE or Logistic NLL, we can do gradient descent
  - Gradient equations & update rules

Multiclass linear models

- Define a generic linear classifier by

$$f(x\,;\theta) = \arg\max_y \;\; \theta \cdot \Phi(x, y)$$

- Example: y ∈ {-1, +1}

$$\Phi(x, y) = y \; [1 \;\; x \;\; x^2 \;\; \ldots]$$

$$f(x\,;\theta) = \begin{cases} +1 & \theta \cdot [1 \;\; x \;\; x^2 \ldots] > -\theta \cdot [1 \;\; x \;\; x^2 \ldots] \\ -1 & \text{o.w.} \end{cases}$$

(Standard perceptron rule)

Multiclass linear models

- Define a generic linear classifier by

$$f(x\,;\theta) = \arg\max_y \;\theta \cdot \Phi(x,y)$$

- Example: y ∈ {0,1,2,…}

$$\Phi(x,y) = [\;\mathbb{1}[y=0][1\;x\;x^2\;\;\ldots]\;\;\mathbb{1}[y=1][1\;x\;x^2\ldots]\ldots]$$
$$\theta = [\;\;[\theta_{00}\;\;\theta_{01}\;\;\theta_{02}\ldots]\;\;\;\;[\theta_{10}\;\;\theta_{11}\;\;\theta_{12}\ldots]\;\;\;\ldots]$$

(parameters for each class c)

$$f(x\,;\theta) = \arg\max_c \;\theta_{c\cdot} \cdot [1\;x\;x^2\;\;\ldots]$$

(predict class with largest linear response)

Training multiclass perceptrons

- Multi-class perceptron algorithm
  - Straightforward generalization of perceptron alg
- Multilogistic regression
  - Take $p(c \mid x) \propto \exp[\, \theta\, \Phi(x,c) \,]$
  - Normalize by sum over classes c
  - Straightforward generalization of logistic regression