CSE419 – Artificial Intelligence and Machine Learning 2020

PhD Furkan Gözükara, Toros University

https://github.com/FurkanGozukara/CSE419-Artificial-Intelligence-and-Machine-Learning-2020

Lecture 11 Part 1
Gradient Descent

Based on Asst. Prof. Dr. David Kauchak (Pomona College) Lecture Slides

What is Gradient Descent

- Gradient descent is a first-order iterative optimization algorithm for finding the minimum of a function.

- To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

- Using the cost function in in conjunction with GD is called linear regression.

- [Understanding and Calculating the Cost Function for Linear Regression](#)

- [Machine Learning week 1: Cost Function, Gradient Descent and Univariate Linear Regression](#)

Linear models

A strong high-bias assumption is *linear separability*:

- in 2 dimensions, can separate classes by a line
- in higher dimensions, need hyperplanes

A *linear model* is a model that assumes the data is linearly separable
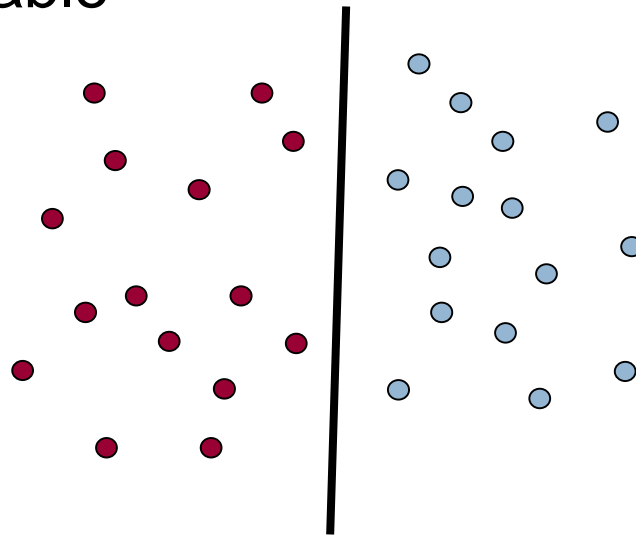
Linear models

A linear model in *n*-dimensional space (i.e. *n* features) is define by *n*+1 weights:

In two dimensions, a line:

$$0 = w_1 f_1 + w_2 f_2 + b \qquad \text{(where b = -a)}$$

In three dimensions, a plane:

$$0 = w_1 f_1 + w_2 f_2 + w_3 f_3 + b$$

In *m*-dimensions, a *hyperplane*

$$0 = b + \sum_{j=1}^{m} w_j f_j$$

Perceptron learning algorithm

repeat until convergence (or for some # of iterations):

for each training example (f_1, f_2, ..., f_m, label):
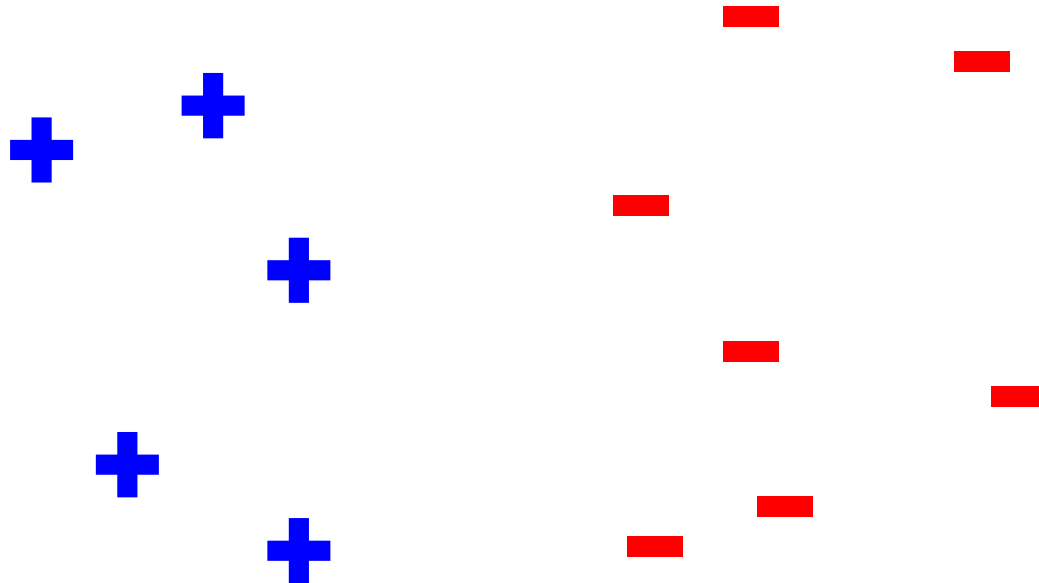
$$prediction = b + \sum_{j=1}^{m} w_j f_j$$

if *prediction * label* ≤ 0: // they don't agree
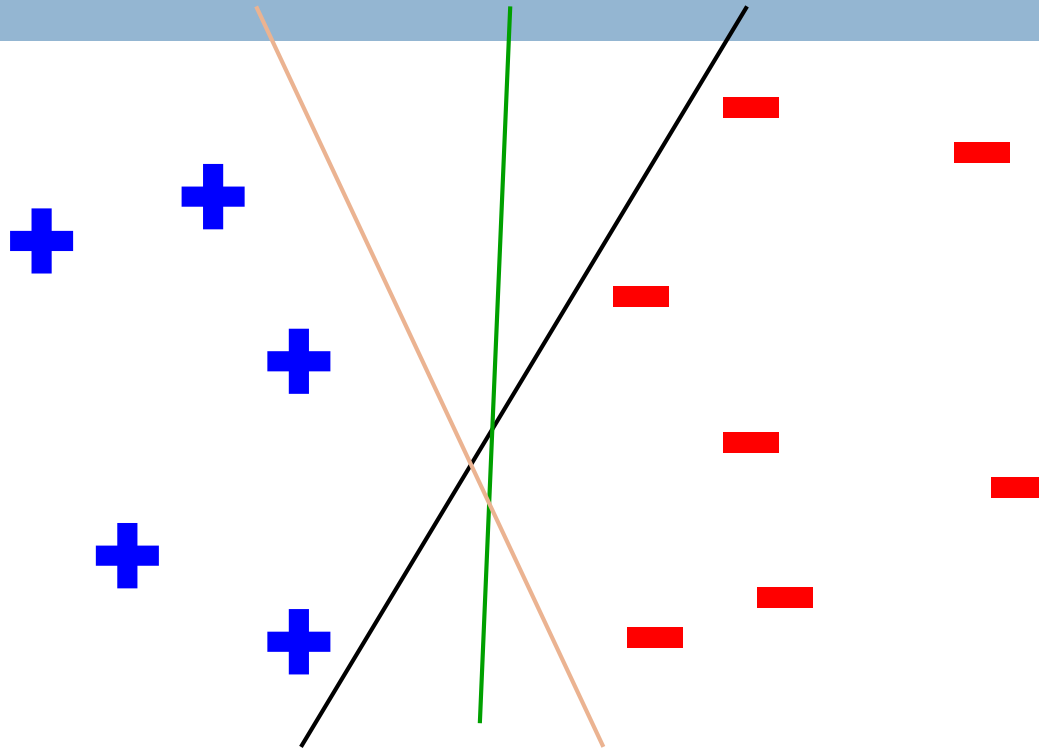
for each w_j:

$w_j = w_j + f_j^*$label

$b = b$ + label

Which line will it find?

Which line will it find?

Only guaranteed to find *some* line that separates the data

Linear models

Perceptron algorithm is one example of a linear classifier

Many, many other algorithms that learn a line (i.e. a setting of a linear combination of weights)

Goals:
- Explore a number of linear training algorithms
- Understand *why these algorithms work*

Perceptron learning algorithm

repeat until convergence (or for some # of iterations):

for each training example (f_1, f_2, …, f_m, label):

$$prediction = b + \sum_{j=1}^{m} w_j f_j$$

if *prediction * label* ≤ 0: // they don't agree

for each w_i:

$w_i = w_i + f_i *$label

$b = b +$ label

A closer look at why we got it wrong

w_1 w_2

$$(-1, -1, \text{positive})$$

$$0 * f_1 + 1 * f_2 =$$

$$0 * -1 + 1 * -1 = -1$$

We'd like this value to be positive since it's a positive value

didn't contribute, but could have

contributed in the wrong direction

Intuitively these make sense
Why change by 1?
Any other way of doing it?

decrease

decrease

0 -> -1

1 -> 0

Model-based machine learning

1. pick a model

   - e.g. a hyperplane, a decision tree,…

   - A model is defined by a collection of parameters

   What are the parameters for DT? Perceptron?

Model-based machine learning

1. pick a model
   - e.g. a hyperplane, a decision tree,…
   - A model is defined by a collection of parameters

2. pick a criteria to optimize (aka objective function)

What criterion do decision tree learning and perceptron learning optimize?

Model-based machine learning

1. pick a model
   - e.g. a hyperplane, a decision tree,…
   - A model is defined by a collection of parameters

2. pick a criteria to optimize (aka objective function)
   - e.g. training error

3. develop a learning algorithm
   - the algorithm should try and minimize the criteria
   - sometimes in a heuristic way (i.e. non-optimally)
   - sometimes explicitly

Linear models in general

1. pick a model

$$0 = b + \sum_{j=1}^{m} w_j f_j$$

These are the parameters we want to learn

2. pick a criteria to optimize (aka objective function)

Some notation: indicator function

$$1[x] = \begin{cases} 1 & if \ x = True \\ 0 & if \ x = False \end{cases}$$

Convenient notation for turning T/F answers into numbers/counts:

$$drinks\_to\_bring\_for\_class = \sum_{x \in class} 1[x >= 21]$$

Some notation: dot-product

Sometimes it is convenient to use vector notation

We represent an example $f_1, f_2, \ldots, f_m$ as a single vector, x

Similarly, we can represent the weight vector $w_1, w_2, \ldots, w_m$ as a single vector, w

The dot-product between two vectors a and b is defined as:

$$a \times b = \sum_{j=1}^{m} a_j b_j$$

Linear models

1. pick a model

$$0 = b + \mathring{a}_{j=1}^{n} w_j f_j$$

These are the parameters we want to learn

2. pick a criteria to optimize (aka objective function)

$$\mathring{a}_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0\right]$$

What does this equation say?

0/1 loss function

$$\sum_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0 \right]$$

$$\text{distance} = b + \sum_{j=1}^{m} w_j x_j = w \times x + b$$ distance from hyperplane

$$incorrect = y_i(w \times x_i + b) \pounds 0$$ whether or not the prediction and label agree

$$0/1 \text{ loss} = \sum_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0 \right]$$ total number of mistakes, aka 0/1 loss

Model-based machine learning

1. pick a model

$$0 = b + \mathop{\text{\AA}}_{j=1}^{m} w_j f_j$$

2. pick a criteria to optimize (aka objective function)

$$\mathop{\text{\AA}}_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0 \right]$$

3. develop a learning algorithm

$$\text{argmin}_{w,b} \mathop{\text{\AA}}_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0 \right]$$

Find w and b that minimize the 0/1 loss

Minimizing 0/1 loss

$$\text{argmin}_{w,b} \sum_{i=1}^{n} 1\left[y_i (w \times x_i + b) \pounds\ 0 \right]$$

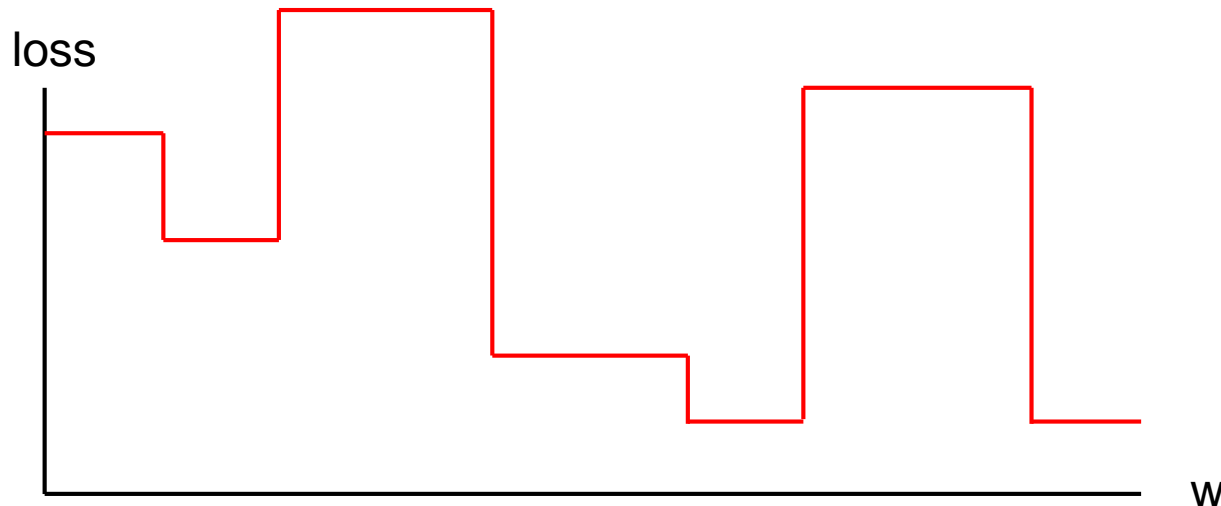Find w and b that minimize the 0/1 loss

How do we do this?
How do we *minimize* a function?
Why is it hard for this function?
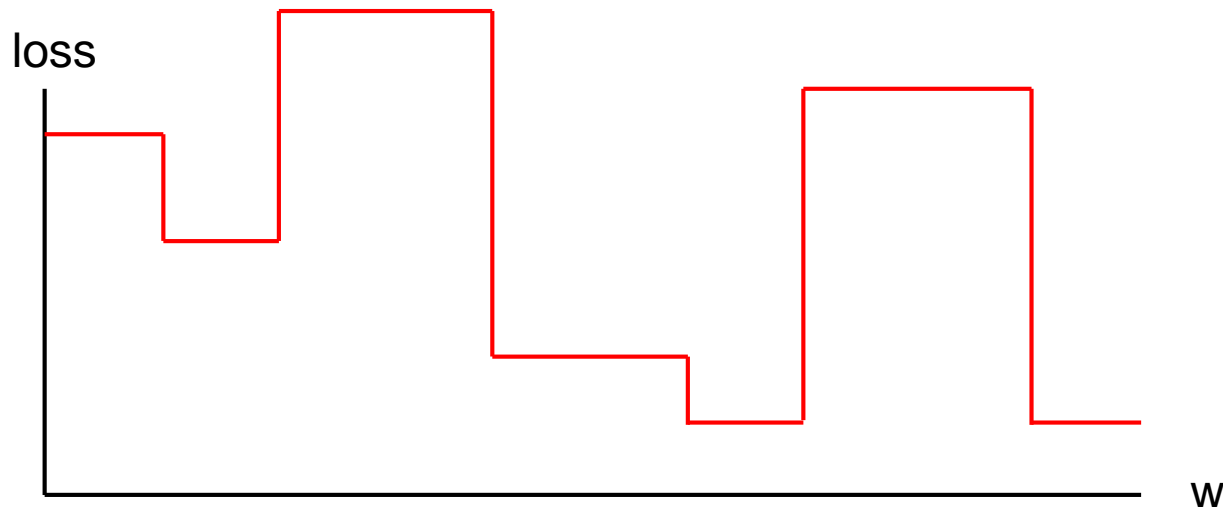
Minimizing 0/1 in one dimension

$$\sum_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds\, 0\right]$$

loss

w

Each time we change w such that the example is right/wrong the loss will increase/decrease

Minimizing 0/1 over all w

$$\sum_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds 0 \right]$$

loss

w

Each new feature we add (i.e. weights) adds another dimension to this space!

Minimizing 0/1 loss

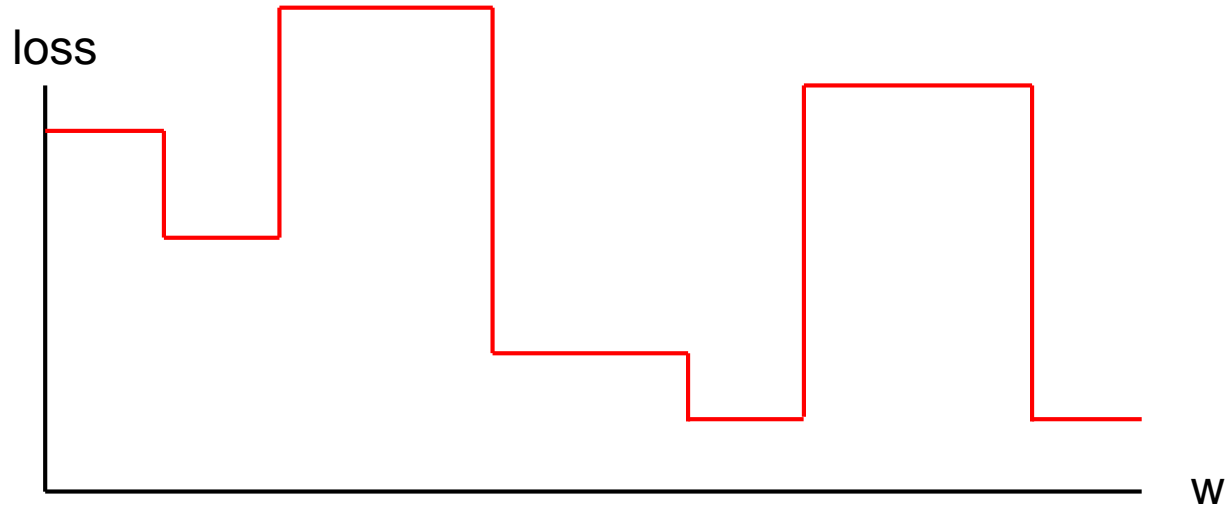$$\text{argmin}_{w,b} \sum_{i=1}^{n} 1\left[y_i(w \times x_i + b) \pounds\, 0\right]$$

Find w and b that minimize the 0/1 loss

This turns out to be hard (in fact, NP-HARD ☹)

Challenge:
- small changes in any w can have large changes in the loss (the change isn't continuous)
- there can be many, many local minima
- at any give point, we don't have much information to direct us towards any minima

More manageable loss functions

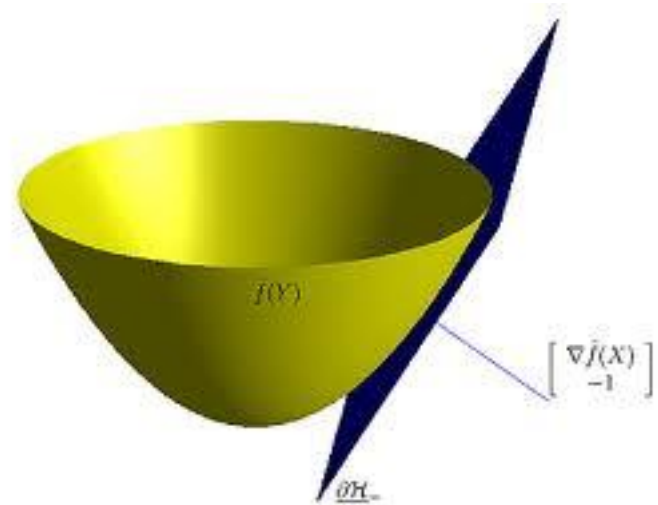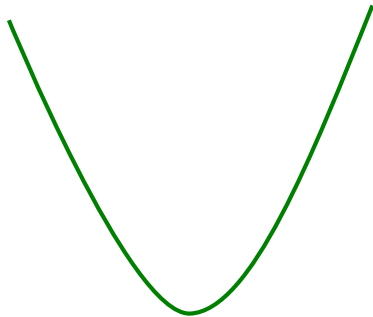What property/properties do we want from our loss function?

More manageable loss functions

loss

w
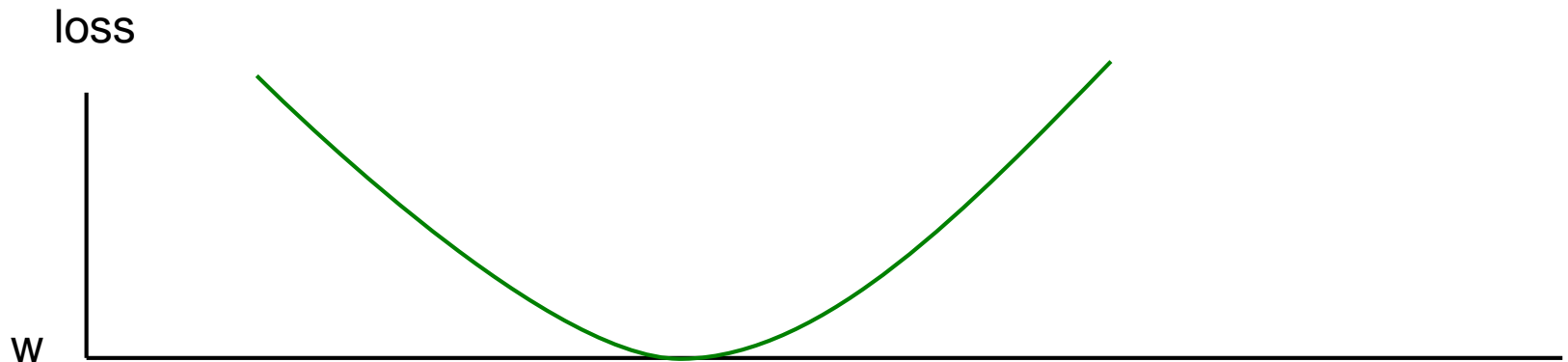
- Ideally, continues (i.e. differentiable) so we get an indication of direction of minimization
- Only one minima

Convex functions

Convex functions look something like:

One definition: The line segment between any two points on the function is *above* the function

Surrogate loss functions

For many applications, we really would like to minimize the 0/1 loss

A surrogate loss function is a loss function that provides an upper bound on the actual loss function (in this case, 0/1)

We'd like to identify convex surrogate loss functions to make them easier to minimize

Key to a loss function is how it scores the difference between the actual label y and the predicted label y'

Surrogate loss functions

0/1 loss: $$l(y, y') = 1[yy' £ 0]$$

Ideas?
Some function that is a proxy for error, but is continuous and convex

Surrogate loss functions

0/1 loss: $\quad l(y, y') = 1\left[yy' \pounds 0\right]$

Hinge: $\quad l(y, y') = \max(0, 1 - yy')$

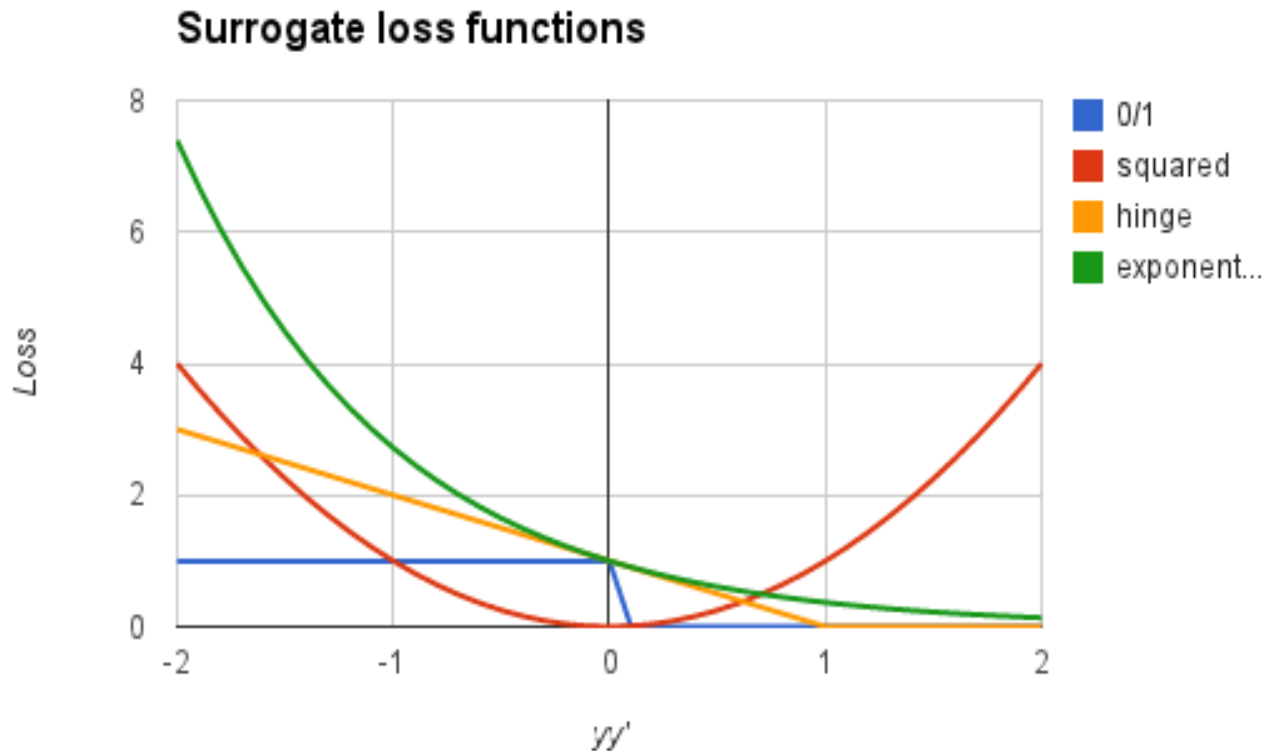Exponential: $\quad l(y, y') = \exp(-yy')$

Squared loss: $\quad l(y, y') = (y - y')^2$

Why do these work? What do they penalize?

Surrogate loss functions

0/1 loss: $l(y, y') = 1[yy' £ 0]$ Hinge: $l(y, y') = \max(0, 1 - yy')$

Squared loss: $l(y, y') = (y - y')^2$ Exponential: $l(y, y') = \exp(-yy')$

Surrogate loss functions

Model-based machine learning

1. pick a model

$$0 = b + \sum_{j=1}^{m} w_j f_j$$

2. pick a criteria to optimize (aka objective function)

$$\sum_{i=1}^{n} \exp(-y_i(w \times x_i + b))$$

use a convex surrogate loss function

3. develop a learning algorithm

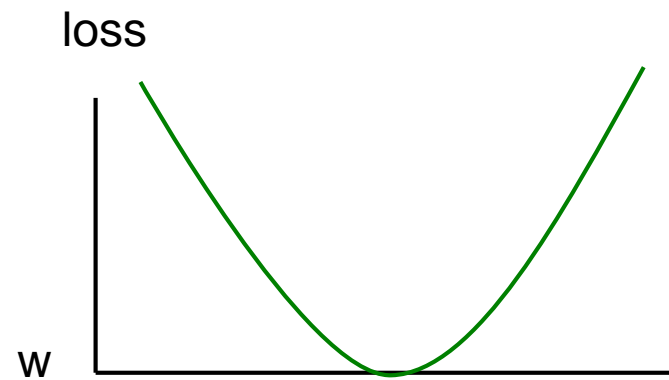$$\arg\min_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b))$$

Find w and b that minimize the surrogate loss

Finding the minimum

You're blindfolded, but you can see out of the bottom of the blindfold to the ground right by your feet. I drop you off somewhere and tell you that you're in a convex shaped valley and escape is at the bottom/minimum. How do you get out?
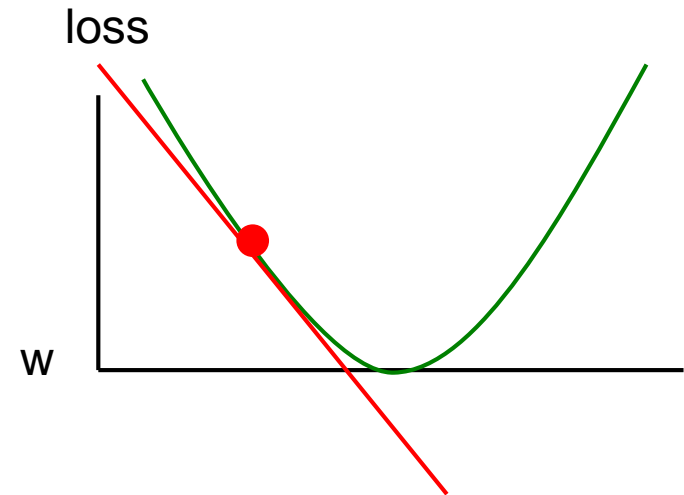
Finding the minimum

loss

w

How do we do this for a function?

One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

Approach:
- pick a starting point (w)
- repeat:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)
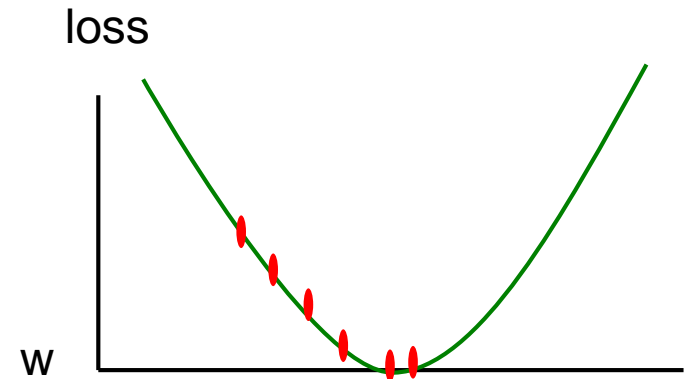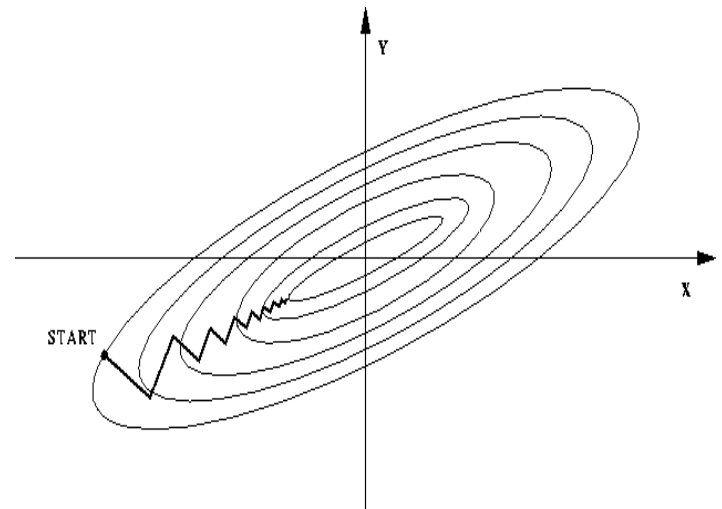
loss

w

One approach: gradient descent

Partial derivatives give us the slope (i.e. direction to move) in that dimension

Approach:
- ☐ pick a starting point (w)
- ☐ repeat:
  - ■ pick a dimension
  - ■ move a small amount in that dimension towards decreasing loss (using the derivative)

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_j = w_j - \eta \frac{d}{dw_j} loss(w)$$

What does this do?

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_j = w_j - \hbar \frac{d}{dw_j} loss(w)$$

learning rate (how much we want to move in the error direction, often this will change over time)

Some maths

$$\frac{d}{dw_j} loss = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b))$$

$$= \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b)) \frac{d}{dw_j} - y_i(w \times x_i + b)$$

$$= \sum_{i=1}^{n} -y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_j = w_j + h \sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

What is this doing?

Exponential update rule

$$w_j = w_j + h\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

for each example x_i:

$$w_j = w_j + h y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

Does this look familiar?

Perceptron learning algorithm!

repeat until convergence (or for some # of iterations):

  for each training example (f_1, f_2, …, f_m, label):

$$prediction = b + \sum_{j=1}^{m} w_j f_j$$

  if *prediction * label* ≤ 0: // they don't agree

    for each w_j:

      $w_j = w_j + f_j$*label

    $b = b +$ label

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

$$\text{or}$$

$$w_j = w_j + x_{ij} y_i c \quad \text{where} \quad c = \eta \exp(-y_i(w \times x_i + b))$$

The constant

$$c = h\exp(-y_i(w \times x_i + b))$$

learning rate   label   prediction

When is this large/small?
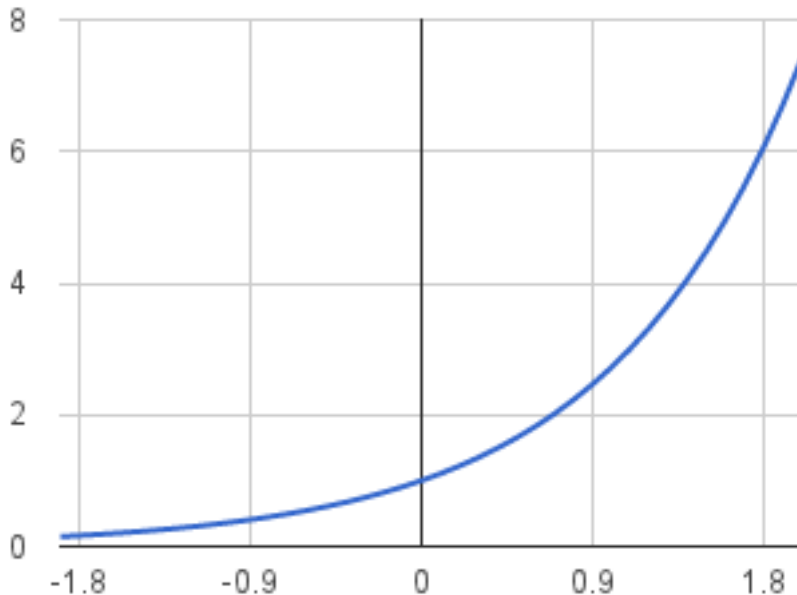
The constant

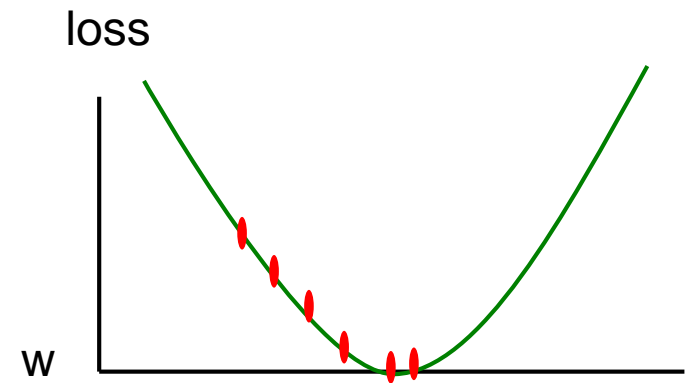$$c = h\exp(-y_i(w{\times}x_i + b))$$

label   prediction

If they're the same sign, as the predicted gets larger there update gets smaller

If they're different, the more different they are, the bigger the update

One concern

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b))$$

loss

What is this calculated on?
Is this what we want to optimize?

w

Perceptron learning algorithm!

repeat until convergence (or for some # of iterations):

for each training example (f_1, f_2, …, f_m, label):

$$prediction = b + \overset{m}{\underset{j=1}{\mathring{a}}} w_j f_j$$

~~if *prediction * label* ≤ 0: // they don't agree~~

for each w_j:

$w_j = w_j + f_j$*label

$b = b +$ label

Note: for gradient descent, we always update

$$w_j = w_j + h y_i x_{ij} \exp(-y_i(w \times x_i + b))$$

or

$$w_j = w_j + x_{ij} y_i c \quad \text{where} \quad c = h \exp(-y_i(w \times x_i + b))$$

One concern

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b))$$

We're calculating this on the **training set**

We still need to be careful about overfitting!

The min w,b on the training set is generally NOT the min for the test set

How did we deal with this for the perceptron algorithm?

loss

w
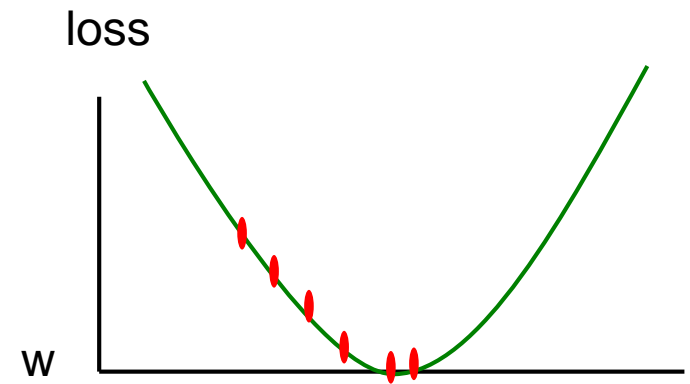
Summary

Model-based machine learning:

- define a model, objective function (i.e. loss function), minimization algorithm

Gradient descent minimization algorithm

- require that our loss function is convex
- make small updates towards lower losses

Perceptron learning algorithm:

- gradient descent
- exponential loss function (modulo a learning rate)

CSE419 – Artificial Intelligence and Machine Learning 2020

PhD Furkan Gözükara, Toros University

Lecture 11 Part 2
Regularization

Based on Asst. Prof. Dr. David Kauchak (Pomona College) Lecture Slides

Overfitting revisited: regularization

A regularizer is an additional criteria to the loss function to make sure that we don't overfit

It's called a regularizer since it tries to keep the parameters more normal/regular

It is a bias on the model forces the learning to prefer certain types of weights over others

$$\operatorname{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

Regularizers

$$0 = b + \sum_{j=1}^{n} w_j f_j$$

Should we allow all possible weights?

Any preferences?

What makes for a "simpler" model for a linear model?

Regularizers

$$0 = b + \overset{n}{\underset{j=1}{\mathring{a}}} w_j f_j$$

Generally, we don't want huge weights

If weights are large, a small change in a feature can result in a large change in the prediction

Also gives too much weight to any one feature

Might also prefer weights of 0 for features that aren't useful

How do we encourage small weights? or penalize large weights?

Regularizers

$$0 = b + \mathring{a}_{j=1}^{n} w_j f_j$$

How do we encourage small weights? or penalize large weights?

$$\text{argmin}_{w,b} \mathring{a}_{i=1}^{n} loss(yy') + l \; regularizer(w,b)$$

Common regularizers

sum of the weights

$$r(w, b) = \sum_{w_j} |w_j|$$

sum of the squared weights

$$r(w, b) = \sqrt{\sum_{w_j} |w_j|^2}$$

What's the difference between these?

Common regularizers

sum of the weights

$$r(w, b) = \sum_{w_j} |w_j|$$

sum of the squared weights

$$r(w, b) = \sqrt{\sum_{w_j} |w_j|^2}$$

Squared weights penalizes large values more
Sum of weights will penalize small values more

p-norm

sum of the weights (1-norm)

$$r(w,b) = \sum_{w_j} |w_j|$$

sum of the squared weights (2-norm)
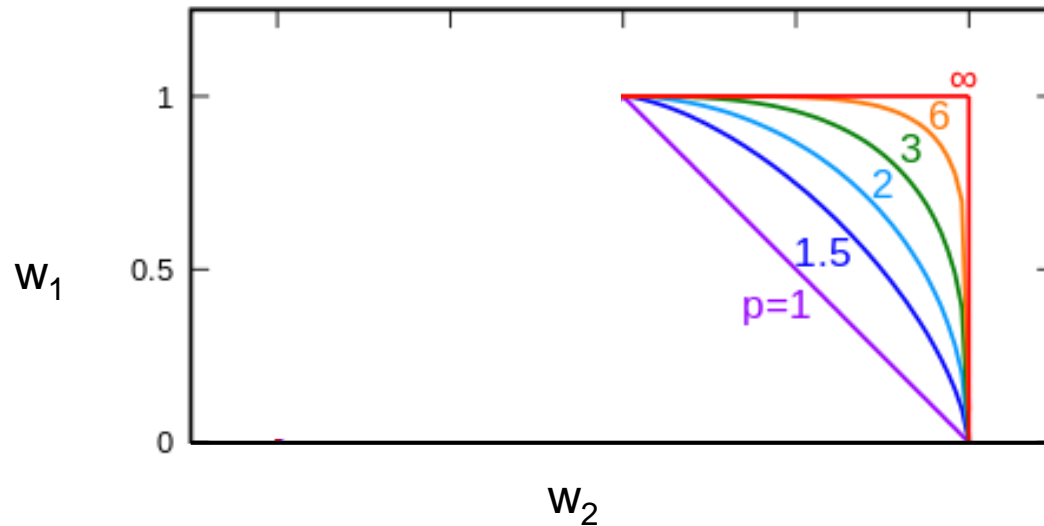
$$r(w,b) = \sqrt{\sum_{w_j} |w_j|^2}$$

p-norm

$$r(w,b) = \sqrt[p]{\sum_{w_j} |w_j|^p} = \|w\|^p$$

Smaller values of p (p < 2) encourage sparser vectors
Larger values of p discourage large weights more

p-norms visualized

w_1

w_2

lines indicate penalty = 1

For example, if w_1 = 0.5

p	w_2
1	0.5
1.5	0.75
2	0.87
3	0.95
∞	1

p-norms visualized
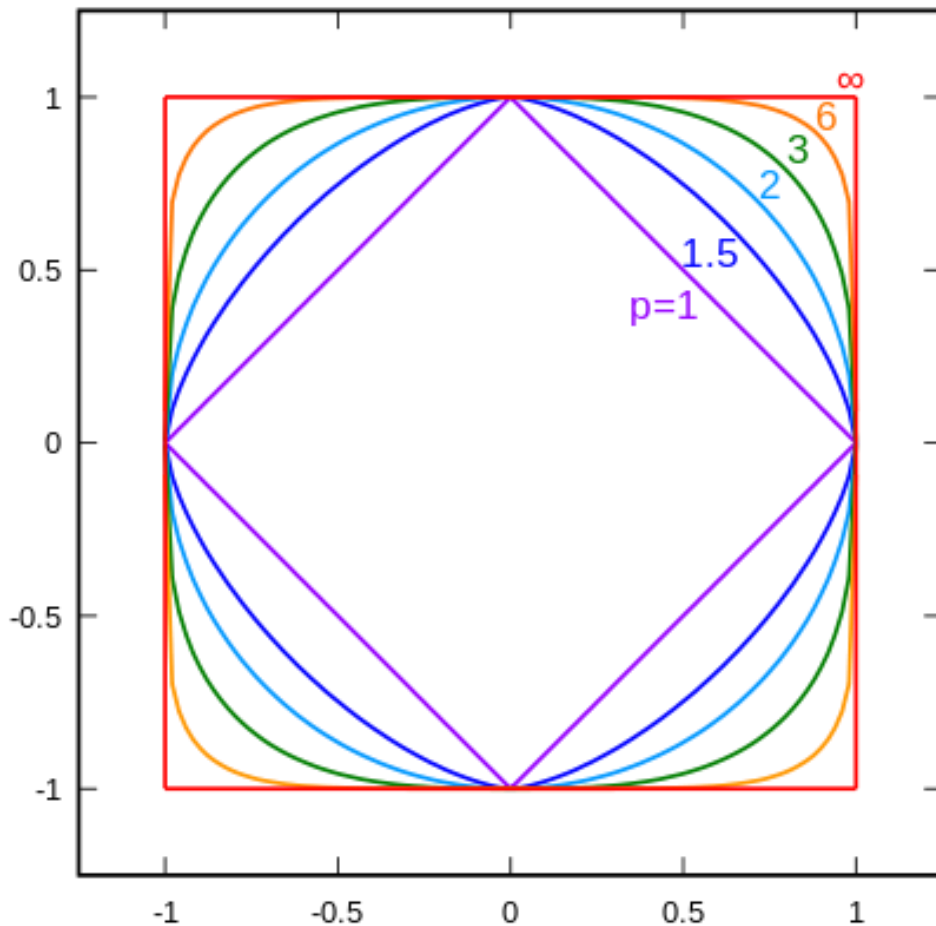
all p-norms penalize larger weights

p < 2 tends to create sparse (i.e. lots of 0 weights)

p > 2 tends to like similar weights

Model-based machine learning

1. pick a model

$$0 = b + \mathring{a}_{j=1}^{n} w_j f_j$$

2. pick a criteria to optimize (aka objective function)

$$\mathring{a}_{i=1}^{n} loss(yy') + \lambda\, regularizer(w)$$

3. develop a learning algorithm

$$\operatorname{argmin}_{w,b} \mathring{a}_{i=1}^{n} loss(yy') + \lambda\, regularizer(w)$$

Find w and b that minimize

Minimizing with a regularizer
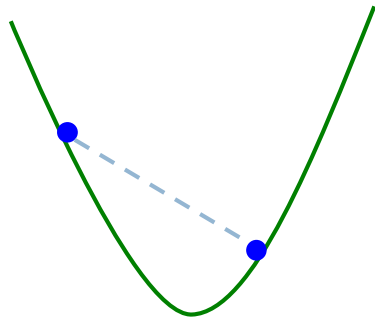
We know how to solve convex minimization problems using gradient descent:

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy')$$

If we can ensure that the loss + regularizer is convex then we could still use gradient descent:

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \, regularizer(w)$$

make convex

Convexity revisited

One definition: The line segment between any two points on the function is *above* the function

Mathematically, *f* is convex if for all x_1, x_2:

$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2) \quad \forall \quad 0 < t < 1$$

the value of the function at some point between x_1 and x_2

the value at some point on the **line segment** between x_1 and x_2

Adding convex functions

Claim: If *f* and *g* are convex functions then so is the function z=*f+g*

Prove:

$$z(tx_1 + (1-t)x_2) \le tz(x_1) + (1-t)z(x_2) \quad " \quad 0 < t < 1$$

Mathematically, *f* is convex if for all *x₁, x₂*:

$$f(tx_1 + (1-t)x_2) \le tf(x_1) + (1-t)f(x_2) \quad " \quad 0 < t < 1$$

Adding convex functions

By definition of the sum of two functions:

$$z(tx_1 + (1-t)x_2) = f(tx_1 + (1-t)x_2) + g(tx_1 + (1-t)x_2)$$

$$tz(x_1) + (1-t)z(x_2) = tf(x_1) + tg(x_1) + (1-t)f(x_2) + (1-t)g(x_2)$$
$$= tf(x_1) + (1-t)f(x_2) + tg(x_1) + (1-t)g(x_2)$$

Then, given that:

$$f(tx_1 + (1-t)x_2) \pounds\, tf(x_1) + (1-t)f(x_2)$$
$$g(tx_1 + (1-t)x_2) \pounds\, tg(x_1) + (1-t)g(x_2)$$

We know:

$$f(tx_1 + (1-t)x_2) + g(tx_1 + (1-t)x_2) \pounds\, tf(x_1) + (1-t)f(x_2) + tg(x_1) + (1-t)g(x_2)$$

So: $z(tx_1 + (1-t)x_2) \pounds\, tz(x_1) + (1-t)z(x_2)$

Minimizing with a regularizer

We know how to solve convex minimization problems using gradient descent:

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy')$$

If we can ensure that the loss + regularizer is convex then we could still use gradient descent:

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \, regularizer(w)$$

convex as long as both loss and regularizer are convex

p-norms are convex

$$r(w, b) = \sqrt[p]{\sum_{w_j} |w_j|^p} = \|w\|^p$$

p-norms are convex for p >= 1

Model-based machine learning

1. pick a model

$$0 = b + \overset{n}{\underset{j=1}{\mathring{\textstyle\sum}}} w_j f_j$$

2. pick a criteria to optimize (aka objective function)

$$\overset{n}{\underset{i=1}{\mathring{\textstyle\sum}}} \exp(-y_i(w \times x_i + b)) + \frac{l}{2}\|w\|^2$$

3. develop a learning algorithm

$$\operatorname{argmin}_{w,b} \overset{n}{\underset{i=1}{\mathring{\textstyle\sum}}} \exp(-y_i(w \times x_i + b)) + \frac{l}{2}\|w\|^2$$

Find w and b that minimize

Our optimization criterion

$$\text{argmin}_{w,b} \overset{n}{\underset{i=1}{\mathring{a}}} \exp(-y_i(w \times x_i + b)) + \frac{l}{2}\|w\|^2$$

Loss function: penalizes examples where the prediction is different than the label

Regularizer: penalizes large weights

Key: this function is convex allowing us to use gradient descent

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - \hbar \frac{d}{dw_i}(loss(w) + regularizer(w, b))$$

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b)) + \frac{\lambda}{2}\|w\|^2$$

Some more maths

$$\frac{d}{dw_j} objective = \frac{d}{dw_j} \mathring{\mathrm{a}}_{i=1}^{n} \exp(-y_i(w \times x_i + b)) + \frac{l}{2}\|w\|^2$$

$\vdots$ (some math happens)

$$= -\mathring{\mathrm{a}}_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \times x_i + b)) + l\, w_j$$

Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - h\frac{d}{dw_i}(loss(w) + regularizer(w,b))$$

$$w_j = w_j + h\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \times x_i + b)) - h/w_j$$

The update

$$w_j = w_j + hy_ix_{ij}\exp(-y_i(w \times x_i + b)) - h/w_j$$

learning rate direction
to update

constant: how far from wrong

regularization

What effect does the regularizer have?

The update

$$w_j = w_j + \hbar y_i x_{ij} \exp(-y_i(w \times x_i + b)) - \hbar / w_j$$

learning rate direction
to update

constant: how far from wrong

regularization

If w_j is positive, reduces w_j
If w_j is negative, increases w_j

moves w_j towards 0

L1 regularization

$$\text{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b)) + \|w\|$$

$$\frac{d}{dw_j} objective = \frac{d}{dw_j} \sum_{i=1}^{n} \exp(-y_i(w \times x_i + b)) + \lambda \|w\|$$

$$= -\sum_{i=1}^{n} y_i x_{ij} \exp(-y_i(w \times x_i + b)) + \lambda \, sign(w_j)$$

L1 regularization

$$w_j = w_j + \hbar y_i x_{ij} \exp(-y_i(w \times x_i + b)) - \hbar l \, sign(w_j)$$

learning rate direction
to update

regularization

constant: how far from wrong

What effect does the regularizer have?

L1 regularization

$$w_j = w_j + \eta y_i x_{ij} \exp(-y_i(w \times x_i + b)) - \eta \lambda\, sign(w_j)$$

learning rate direction
to update

regularization

constant: how far from wrong

If w_j is positive, reduces by a constant
If w_j is negative, increases by a constant

moves w_j towards 0
regardless of magnitude

Regularization with p-norms

L1:

$$w_j = w_j + \eta(loss\_correction - \lambda \, sign(w_j))$$

L2:

$$w_j = w_j + \eta(loss\_correction - \lambda \, w_j)$$

Lp:

$$w_j = w_j + \eta(loss\_correction - \lambda \, cw_j^{p-1})$$

How do higher order norms affect the weights?

Regularizers summarized

L1 is popular because it tends to result in sparse solutions (i.e. lots of zero weights)

> However, it is not differentiable, so it only works for gradient descent solvers

L2 is also popular because for some loss functions, it can be solved directly (no gradient descent required, though often iterative solvers still)

Lp is less popular since they don't tend to shrink the weights enough

The other loss functions

Without regularization, the generic update is:

$$w_j = w_j + hy_i x_{ij} c$$

where

$$c = \exp(-y_i(w \times x_i + b)) \qquad \text{exponential}$$

$$c = 1[yy' < 1] \qquad \text{hinge loss}$$

$$w_j = w_j + h(y_i - (w \times x_i + b)x_{ij}) \quad \text{squared error}$$

Many tools support these different combinations

Look at scikit learning package:

http://scikit-learn.org/stable/modules/sgd.html

Common names

(Ordinary) Least squares: squared loss

Ridge regression: squared loss with L2 regularization

Lasso regression: squared loss with L1 regularization

Elastic regression: squared loss with L1 AND L2 regularization

Logistic regression: logistic loss