GRADIENT DESCENT

David Kauchak
CS 158 – Fall 2019

Admin

Assignment 3 almost graded

Assignment 5
- Course feedback

An aside: text classification

Raw data | labels

Chardonnay

Pinot Grigio

Zinfandel

Text: raw data

Raw data | labels | Features?

Chardonnay

Pinot Grigio

Zinfandel

Feature examples

Raw data | labels | Features

Chardonnay — Clinton said pinot repeatedly last week on tv, "pinot, pinot, pinot"

Pinot Grigio — (1, 1, 1, 0, 0, 1, 0, 0, …)

pinot clinton said california across tv wrong capital

Zinfandel — Occurrence of words

Feature examples

Raw data | labels | Features

Chardonnay — Clinton said pinot repeatedly last week on tv, "pinot, pinot, pinot"

Pinot Grigio — (4, 1, 1, 0, 0, 1, 0, 0, …)

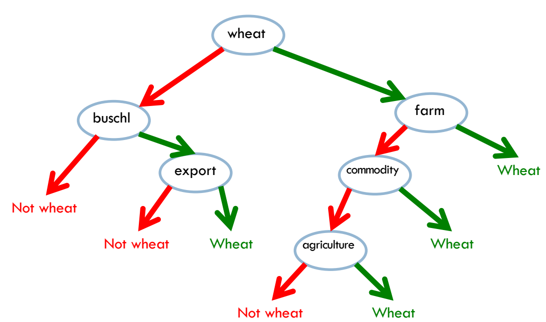
pinot clinton said california across tv wrong capital

Zinfandel — Frequency of word occurrences

This is the representation we're using for assignment 5

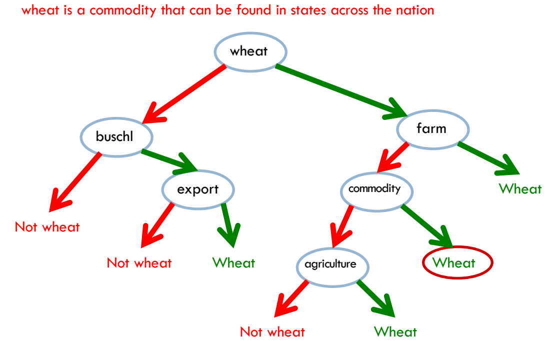
Decision trees for text

Each internal node represents whether or not the text has a particular word



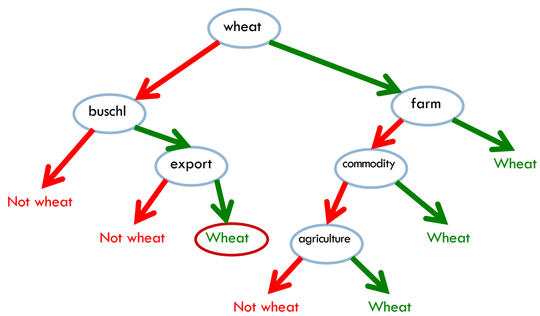
Decision trees for text

wheat is a commodity that can be found in states across the nation

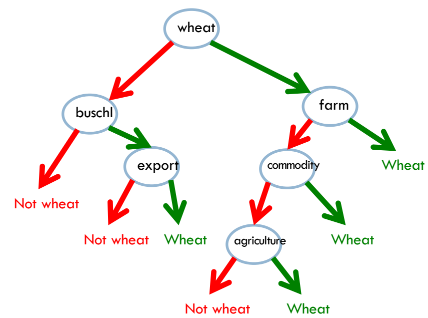


Decision trees for text

The US views technology as a commodity that it can export by the buschl.

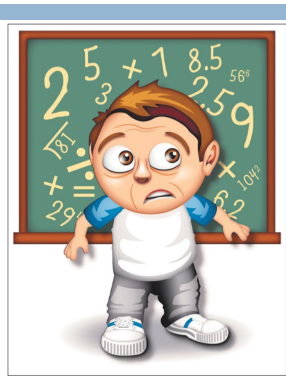


Printing out decision trees




```
(wheat
  (buschl
    predict=not wheat
    (export
      predict=not wheat
      predict=wheat))
  (farm
    (commodity
      (agriculture
        predict=not wheat
        predict=wheat)
      predict=wheat)
    predict=wheat))
```

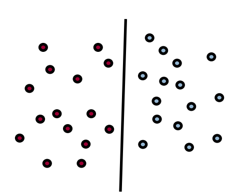
Some math today (but don't worry!)



Linear models

A 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:

$$0 = w_1 f_1 + w_2 f_2 + b$$ (where b = -a)

In three dimensions, a plane:

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

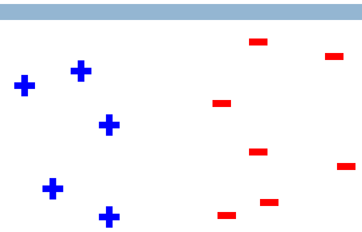
In *m*-dimensions, a *hyperplane*

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

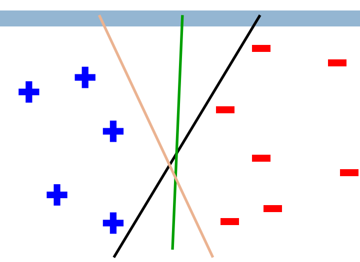
Perceptron learning algorithm

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_i:
        $w_i = w_i + f_i$*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 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, 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

decrease

0 -> -1

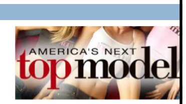
decrease

1 -> 0

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

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

DT: the structure of the tree, which features each node splits on, the predictions at the leaves

perceptron: the weights and the b value

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 criterion to optimize (aka objective function)

What criteria do decision tree learning and perceptron learning optimizing?

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 criterion 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 exactly

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 criterion to optimize (aka objective function)

Some notation: indicator function

$$1[x] = \left\{ \begin{array}{ll} 1 & if\ x = True \\ 0 & if\ x = False \end{array} \right\}$$

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

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

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

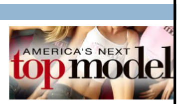
- j subscript will indicate feature indexing, i.e., x_j
- i subscript will indicate examples indexing over a dataset, i.e., x_i or sometimes x_{ij}

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:

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

Linear models



1. pick a model

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

These are the parameters we want to learn

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

$$\sum_{i=1}^{n} 1[y_i(w \cdot x_i + b) \le 0]$$

What does this equation say?

0/1 loss function

$$\sum_{i=1}^{n} 1[y_i(w \cdot x_i + b) \le 0]$$

whether or not the prediction and label agree, true if ***they don't***

- distance from hyperplane
- sign is prediction

total number of mistakes, aka 0/1 loss

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)

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

3. develop a learning algorithm

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

Find w and b that minimize the 0/1 loss (i.e. training error)

Minimizing 0/1 loss

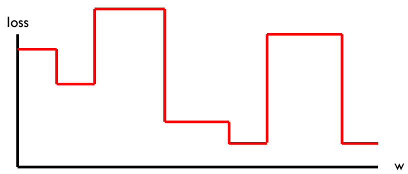
$$\operatorname{argmin}_{w,b} \sum_{i=1}^{n} 1\left[y_i(w \cdot x_i + b) \le 0 \right]$$

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

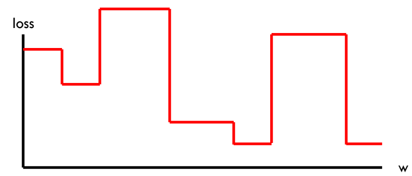
$$\sum_{i=1}^{n} 1\left[y_i(w \cdot x_i + b) \le 0 \right]$$



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

Minimizing 0/1 over all w

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



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

Minimizing 0/1 loss

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

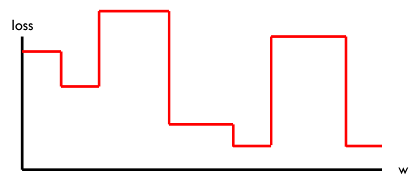
Find w and b that minimize the 0/1 loss

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

Challenge:

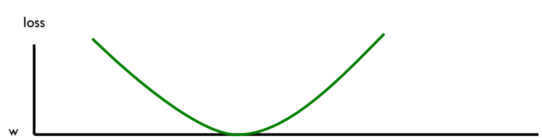
- small changes in any w can have large changes in the loss (the change isn't continuous)
- there can be many, many local minima
- at any given point, we don't have much information to direct us towards any minima

More manageable loss functions



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

More manageable loss functions



- Ideally, continuous (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: how it scores the difference between the actual label ***y*** and the predicted label ***y'***

Surrogate loss functions

0/1 loss: $l(y, y') = 1[yy' \leq 0]$

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

Surrogate loss functions

0/1 loss: $l(y, y') = 1[yy' \leq 0]$

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

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

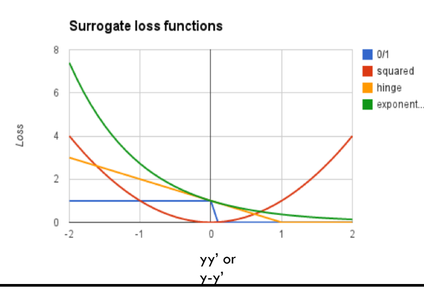
Squared loss: $l(y, y') = (y - y')^2$

Why do these work? What do they penalize?

Surrogate loss functions

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

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



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)

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

use a convex surrogate loss function

3. develop a learning algorithm

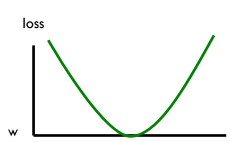
$$\operatorname{argmin}_{w,b} \sum_{i=1}^{n} \exp(-y_i(w \cdot x_i + b))$$

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?

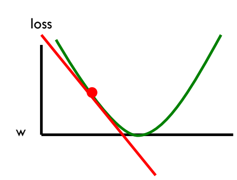
Finding the minimum



How do we do this for a function?

One approach: gradient descent

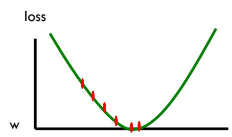
Partial derivatives give us the slope (i.e. direction to move) in that dimension



One approach: gradient descent

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

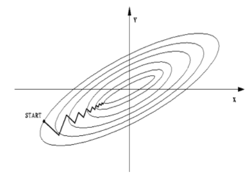
Approach:
- pick a starting point (w)
- repeat:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)



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 any dimension:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

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

Why negative?

Gradient descent

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

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

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

Some math

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

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

Some math

$$-\frac{d}{dw_j} y_i(w \cdot x_i + b) = -\frac{d}{dw_j} y_i(\textstyle\sum_{j=1}^{m} w_j x_{ij} + b)$$

$$= -\frac{d}{dw_j} y_i(w_1 x_{i1} + w_2 x_{i2} + \ldots + w_m x_{im} + b)$$

$$= -\frac{d}{dw_j} y_i w_1 x_{i1} + y_i w_2 x_{i2} + \ldots + y_i w_m x_{im} + y_i b)$$

$$= -y_i x_{ij}$$

Some math

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

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

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

Gradient descent

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

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

What is this doing?

Exponential update rule

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

for each example x_i:

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

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_i:

$w_i = w_i + f_i$*label

b = b + label

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

or

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

The constant

$$c = \eta \exp(-y_i(w \cdot x_i + b))$$

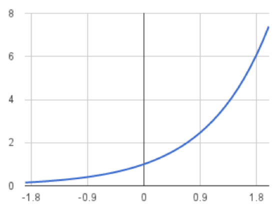
learning rate label prediction

When is this large/small?

The constant

$$c = \eta \boxed{\exp(-y_i(w \cdot x_i + b))}$$

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

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

Note: for gradient descent, we always update

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

or

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

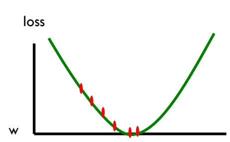
One concern

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

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?

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)