**Classification**

example of multiclass classifier: web page classifier

most common classifier: spam filtering

more complex classifier: image classifier

prospective application: disease classifier- personalized medical diagnosis

**Week 1 Linear classifiers: Logistic regression**

(1) Classifier

example: restaurant review system

input: sentence from review

output: predicted class

y-hat = +1: positive review

y-hat = -1: negative review

a linear classifier will use training data to learn a weight or coefficient for each word

e.g. good:1.0, great:1.4, terrible:-2.1

then compute a score for an input: score(**x**)=weighted count of words in sentence

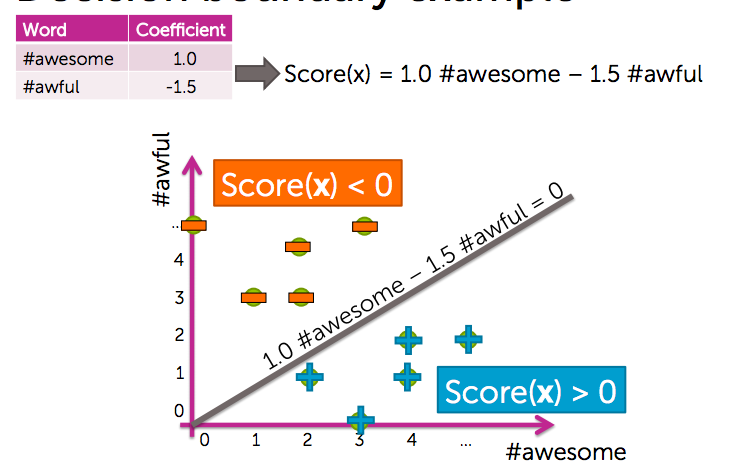
if score(**x**) > 0: y-hat = +1

(2) Decision boundaries

boundary between positive predictions and negative predictions

e.g. awesone:1.0, awful:-1.5

score(**x**) = 1.0#awesome – 1.5#awful



when 2 coefficients are non-zero: line

when 3 coefficients are non-zero: plane

when many coefficients are non-zero: hyperplane

(2) Notation

output: y {-1, +1}

inputs: **x** = (**x**[1], **x**[2], …, **x**[d])

d-dim vector

notational conventions:

**x**[j] = jth input(scalar)

hj(**x**) = jth feature (scalar)

**x**i = input of ith data point (vector)

**x**i[j] = jth input of ith data point (scalar)

Model:

feature 1 = 1

feature 2 = **x**[1] e.g. # awesome

…

feature d+1 = **x**[d] e.g. # awful

→more genetic features- D-dimensional hyperplane

Model:

features: h0(**x**), …, hD(**x**)

(3) Predicting class probabilities

Some reviews may not be definitely positive/ negative

e.g. probability a review with 3 “awesome” and 1 “awful” is positive is 0.9

many classifiers provide a degree of certainty: P(y|**x**)

**x** is the input sentence

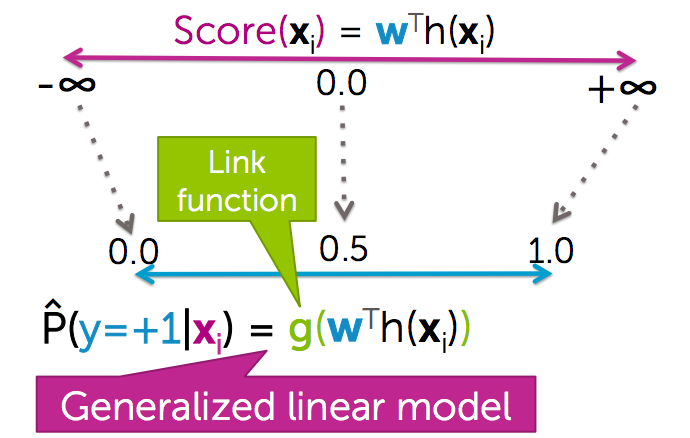
y is the output label (positive/ negative)

goal: learn conditional probability from data, find best model P-hat by finding best w-hat

→ predicting class probabilities with (generalized) linear models:

for Score(**x**i): range from –infinity(y-hat = -1) to infinity(y-hat = 1)

use a link function to generalize linear model



(4) Logistic regression classifier

linear score with logistic link function

**logistic function(sigmoid, logit):**

(5) Categorical inputs

inputs categories:

①numeric inputs

e.g. number of “awesome”, “age”, “salary”, …

intuitive when multiplied by coefficient

②categorical inputs

e.g. gender – male/female, country of birth, zipcode

encoding categories as numeric features:

**x** = country of birth (196 categories)

1-hot encoding:

create one feature for every possible country

only one these features will have the value 1, all the others are 0

encoding a text input- bag of words:

each feature is the number of certain work

(6) Multi-class classification

**1 versus all approach**

C possible classes: y can be 1, 2, …, C (maybe□,△,○)

N data points with their outputs be 1,2,…, or C

we need to learn P(□|x), P(△|x), …, P(○|x)

we can estimate using 2-class model:

+1 class: points with yi = □

-1 class: points with yi = △ or ○

→train classifier

we will learn three model:

=

=

=

for any input, which model outputs a higher probability wins

**Week 2 Logistic Regression Learning**

(1) Maximum likelihood estimation (MLE)

Quality metric for logistic regression

when training data, we need to optimize quality metric:

for each observation with y=+1, P(y=+1|**x**i, **w**) = 1

for each observation with y=-1, P(y=-1|**x**i, **w**) = 0

usually, no w-hat achieves perfect predictions as above

Likelihood l(**w**): measures quality of fit for model with coefficients **w**

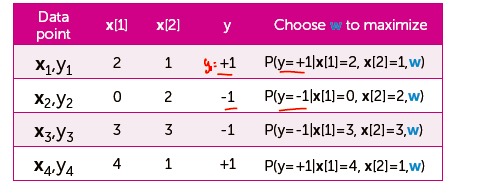
the higher (the closer you get to 1), the better

Gradient Ascent can be used to find w-hat

(2) Data likelihood

quality metric: probability of data

example:



l(**w**) = P(y=+1|**x**[1]=1,**x**[2]=1, **w**) \* P(y=-1|**x**[1]=0,**x**[2]=2, **w**) \*

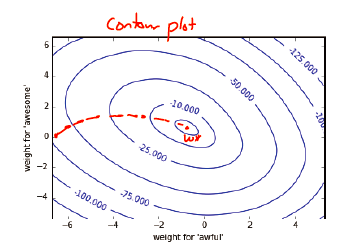
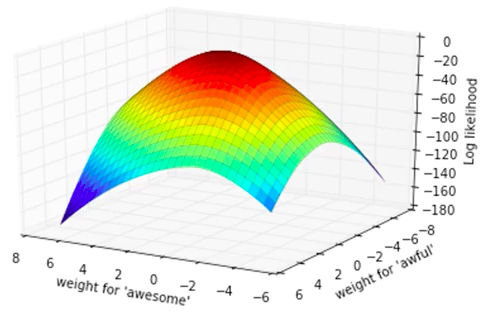
P(y=-1|**x**[1]=3,**x**[2]=3, **w**) \* P(y=+1|**x**[1]=4,**x**[2]=1, **w**)

= P(y1|**x**1, **w**) \* P(y2|**x**2, **w**) \* P(y3|**x**3, **w**) \* P(y4|**x**4, **w**)

→

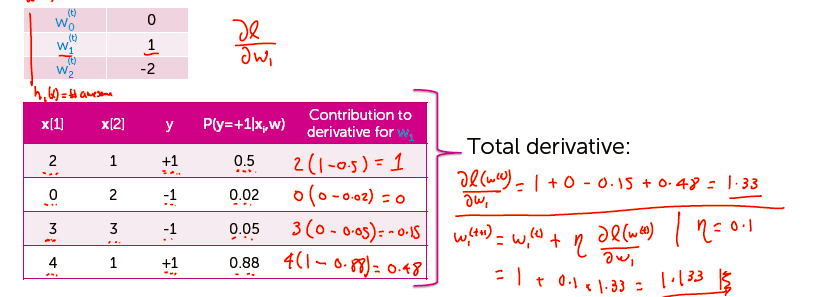
N: number of data points

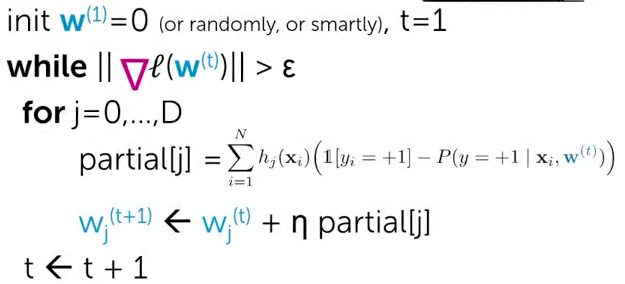
(3) Derivative (gradient ascent)



find :

D+1 dimension vector



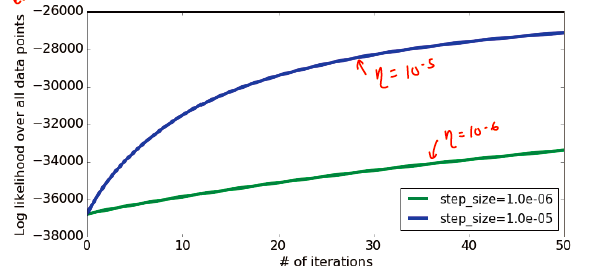


(4) choosing stepsize η

building a **learning curve**

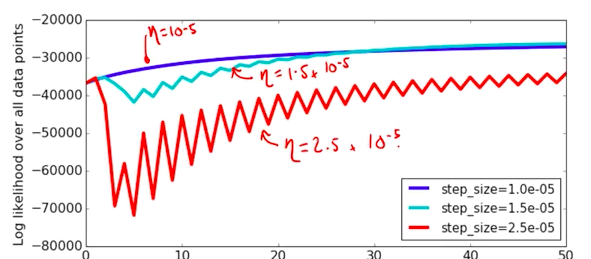
likelihood over all data points v.s. number of iterations

higher likelihood is better



for η=10^(-5), the curve still goes up after 50 iterations, thus requires more iterations still

for η=10^(-6), the curve goes slower, is worse than 10^(-5)



η=1.5 \* 10^(-5) is the best in the above plot

rule of thumb for choosing step size:

picking step size requires a lot of trial and error

try a several values, exponentially spaced

goal: plot learning curve to

find one η that is too small (smooth but moving too slowly)

find one η that is too large (oscillation or divergence)

try values in between to find best η

can also try step size that decreases with iterations, e.g.

(5) Evaluating a classifier

classifier error:

compare true label with the output of a classifier

count total correct and mistake cases to calculate accuracy

error = # mistakes/ # total data points

accuracy = 1 – error

(6) Overfitting in classification

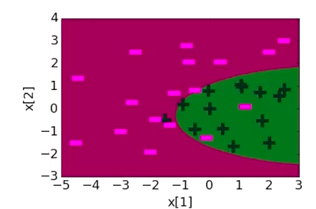
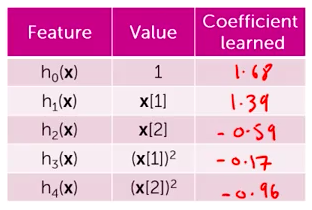
For regression, overfitting happens when there exists w\*:

training\_error(w\*) > training\_error(w^)

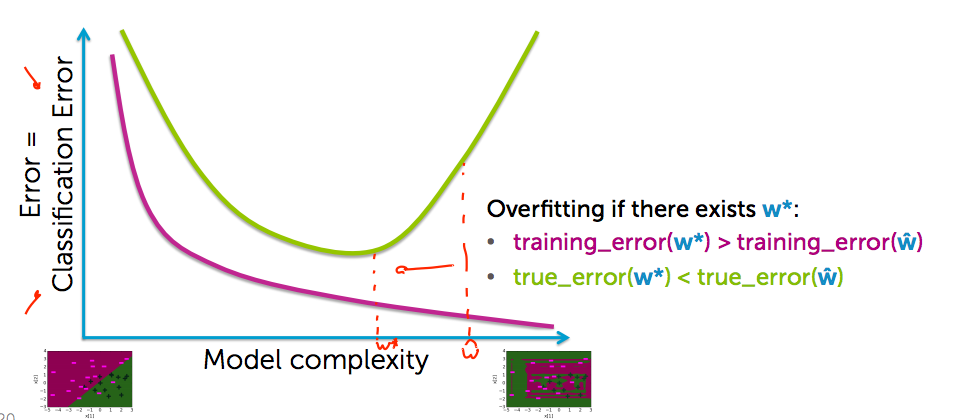
true\_error(w\*) < true\_error(w^)

For classification

e.g. higher degree fitting

even higher degree: coefficient learned gets quite larger

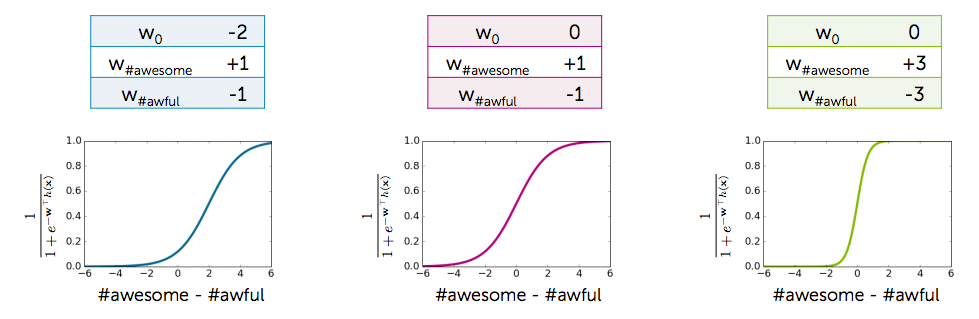


negative consequence of overfitting in logistic regression:

large coefficient values

score is very positive/negative, thus through the link function, sigmoid(score) goes to 1/0

model becomes extremely overconfident of predictions (for the same value, a model with higher coefficient will make a more a prediction with more confident probability)



(7) Avoid overfitting

penalizing large coefficients to mitigate overfitting

total quality = measure of fit – measure of magnitude of coefficients

measure of fit: data likelihood, the larger the better

typically, we use the log of likelihood function:

simplifies math and has better convergence properties

measure of magnitude of logistic regression coefficients

sum of squares (L2 norm)-

or sum of absolute value (L1 norm)-

(8) L2 regularized logistic regression

L2 regularized logistic regression

lambda is the tuning parameter = balance of fit and magnitude

if lambda = 0: just maximize likelihood- overfitting (standard MLE solution)

if lambda = infinity: only care about penalizing w, all weights become 0

pick lambda using:

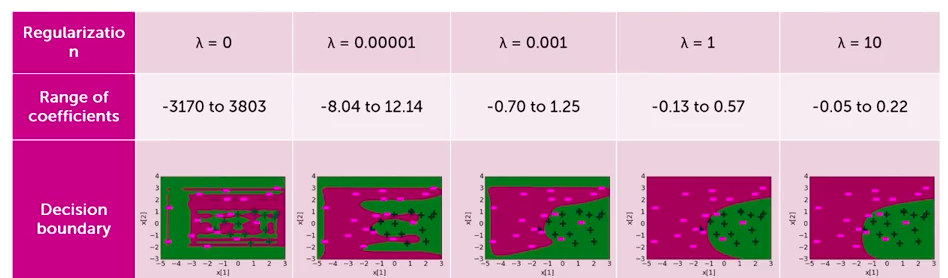
validation set (for large datasets)

cross-validation (for smaller datasets)

large lambda- high bias, low variance

small lambda- low bias, high variance

→ lambda controls model complexity

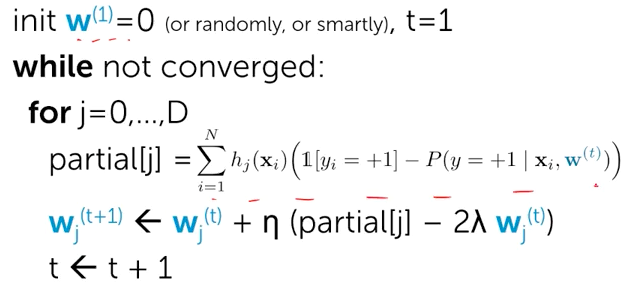


using gradient ascent to find best L2 regularized logistic regression:

derivative of log-likelihood:

derivative of L2 penalty:

→ total derivative:



(9) sparse logistic regression with L1 regularization

sparsity (many wj-hat=0) gives efficiency and interpretability

efficiency:

if size(w) = 100B, each prediction is expensive

if w-hat sparse, computation only depends on number of non-zeros

interpretability:

which features are relevant for prediction

total quality:

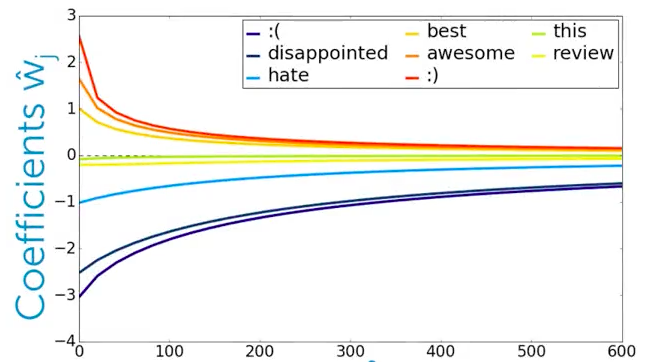
lambda is the tuning parameter = balance of fit and sparsity

if lambda = 0: just maximize likelihood- overfitting (standard MLE solution)

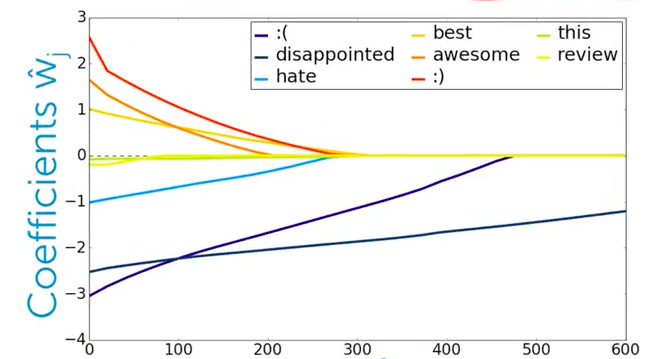
if lambda = infinity: only care about penalizing w, all weights become 0

if lambda in between: sparse solutions

Coefficients v.s. lambda (L2)



coefficients v.s. lambda(L1)



**Week 3 Decision Trees**

(1) Example - predicting potential loan defaults

makes a loan to buy a new house

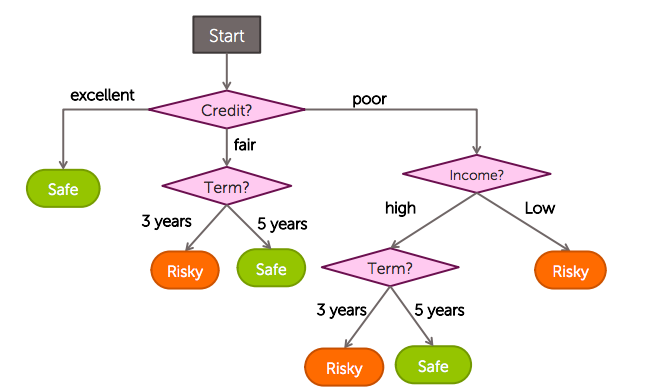
the bank will consider historical credit, income, term, and other personal info

→intelligent loan application review system

input: loan applications

output: rating of the loan (safe/risky)

decision tree model for loan:



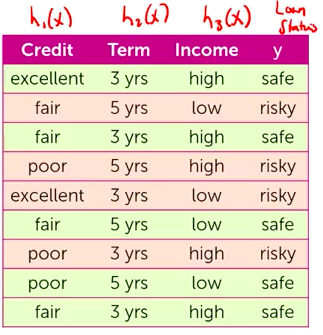
(2) Learning decision trees from data

N observations (**x**i, yi)

quality metric: classification error = # incorrect predictions / # examples

to minimize classification error is extremely hard: exponentially large number of possible trees makes decision tree learning hard (NP-hard problem)

→**simple(greedy) algorithm** could be helpful



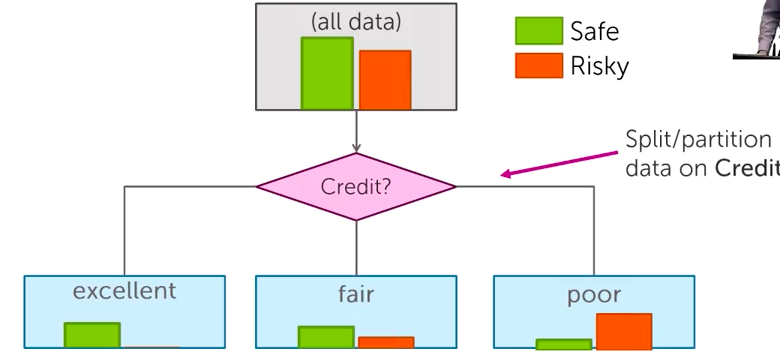
(3) Recursive greedy algorithm

Step 1: start with an empty tree

Step 2: select a feature to split data (feature split selection refer to (5))

e.g. split data on Credit feature

get the subset of data with credit = excellent/ fair/ poor



For each split of the tree:

Step 3: if nothing more to do, make predictions (stop condition refer to (6))

predict “safe” for every loan that has excellent credit

Step 4: otherwise, go to step2 and continue (recurse) on this split

build tree from subsets of “fair” and “poor”

(4) Learning a decision stump

feature split learning = decision stump learning

decision stump: one level on decision tree

steps:

start with all the data(root node) and do one split

we can make predictions with a decision stump (those intermediate nodes)

for “excellent” subset, the majority is “safe”; for “fair”, “safe”; for “poor”, “risky”

(5) Selecting best feature to split on

Split on “Credit” or on “Term”, what is a better split?

the better one should give lower classification error

→measuring mistakes for each decision stump

step 1: y-hat = class of majority of data in node

step 2: calculate classification error of predicting y-hat for this data

e.g.

for the root node, there are 40 data points, 22 are rated as “safe”, 18 as “risky”

thus y-hat = “safe”, error = 22/40 = 0.45

if split on “credit”: error = (0+4+4)/40 = 0.20

if split on “term”: error = (4+6)/40 = 0.25

thus split on “credit” is a better choice

Feature split selection algorithm:

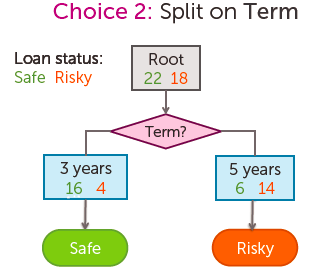
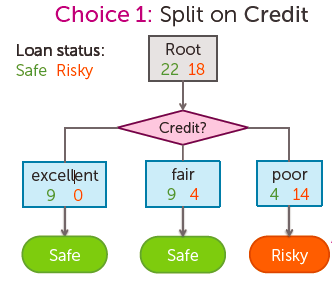
given a subset of data M (a node in a tree)

for each feature hi(x):

1. split data of M according to feature hi(x)

2. compute classification error split

choose feature h\*(x) with lowest classification error



(6) When to stop recursing

when we have learned a decision stump

for those subsets that can be made a prediction directly: create a leaf node of “safe” or “risky”

for the others: recurse (make second level)

→need to decide when to stop

**Stopping condition 1**: all data agrees on y

e.g. all loans with excellent credit are safe

**Stopping condition 2**: already split on all features

no more features can be split on

(7) Making predictions with decision trees

for any particular input,

e.g. **x**i = (Credit=poor, Income=high, Term=5years)

traverse a decision tree

→Decision Tree Predicting Algorithm:

predict(tree\_node, input)

if current tree\_node is a leaf:

return majority class of data points in leaf

else:

next\_node = child node of tree\_node whose feature value agrees with input

return predict(next\_node, input)

(8) Multiclass classification with decision trees

e.g. output for loan review system could be “safe”, “risky” and “danger”

we can predict probability of a certain classification as regard to each subset

(9) Threshold splits for continuous inputs

we could have real valued features (continuous inputs)

e.g. “income” could be any real value

solution 1: split on each numeric value

then when we split on “income”, each node may only contain one data point

very prone to overfitting

solution 2: **threshold split**

split on the feature “income” to “<60k” and “>= 60k”

to pick the best threshold:

for two data points VA and VB, if there are no points in between, or say same classification error for any threshold split between VA and VB

so we consider the middle point between any of the two neighboring data points that we have

Threshold split selection algorithm:

step1: sort the values of a feature hj(**x**):

let {v1, v2, ..., vN} denote sorted values

step2:

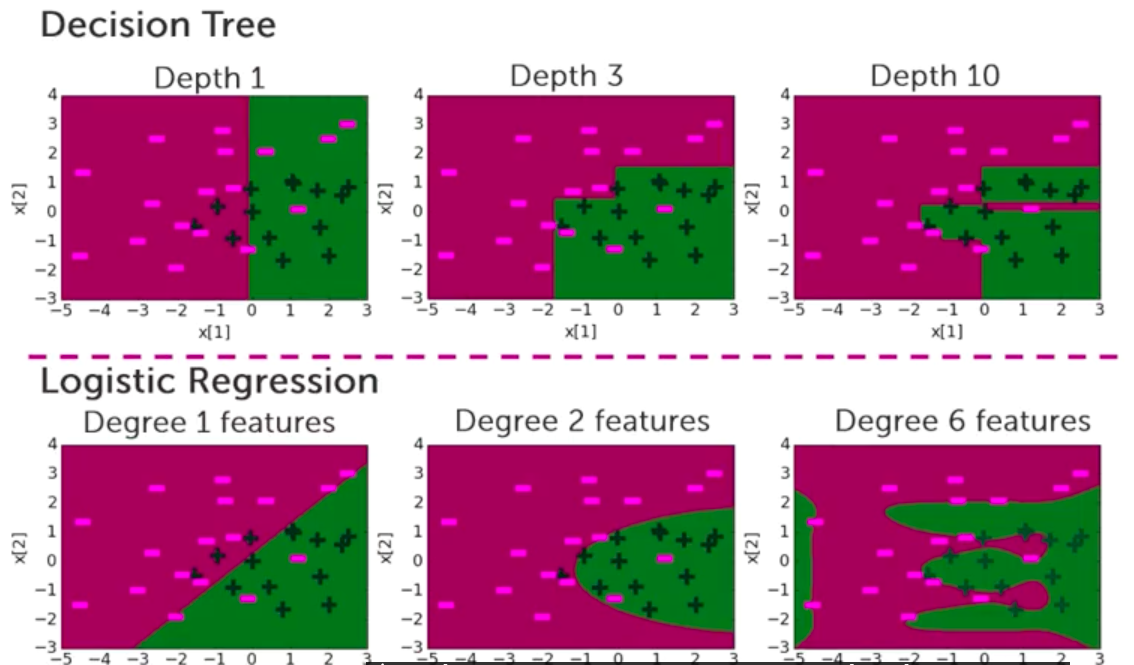
for i = 1, 2, …, N-1

consider split ti = (vi + vi+1)/ 2

compute classification error for threshold split hj(**x**)>= ti

choose the t\* with the lowest classification error

(10) Visualizing decision boundaries



**Week 4 Decision Trees Overfitting**

(1) Overfitting

Overfitting if there exist w\*:

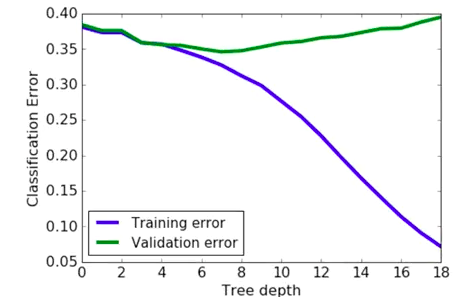
training\_error(w\*) > training\_error (w^)

true\_error(w\*) < true\_error(w^)

in decision trees:

decision stump can be visualized as a vertical line or horizontal line

when we increase the depth, training error reduces and decision boundary becomes more complex



→Principle of Occam’s razor: Simpler trees are better

“Amongh competing hypotheses, the one with fewest assumptions should be selected”, William of Occam, 13th century

when two trees have similar classification error on the validation set, pick the simpler one

to pick a simpler trees:

early stopping: stop learning algorithm before tree become too complex (2)

pruning: simplify tree after learning algorithm terminates (3)

(2) early stopping

①early stopping condition 1:

**limit the depth of a tree**

to pick value for max\_depth: validation set or cross-validation

②early stopping condition 2:

**use classification error to limit depth of tree**

if no split improves classification error, then stop

typically, add magic parameter ε –stop if error doesn’t decreases by more than ε

③early stopping condition 3:

**stop if number of data points contained in a node is too small**

stop when data points in a node <= Nmin

Nmin is typically between 10 and 100, but should consider the whole data size

modified decision tree learning algorithm:

Step 1: start with an empty tree

Step 2: select a feature to split data

For each split of the tree:

Step 3: meet stopping conditions 1&2 or early stopping conditions 1,2&3

Step 4: Otherwise, go to Step 2 and continue (recurse) on this split

(3) Pruning decision trees

grow a tree much bigger than you need, then you chop off parts that turn out to be less important

Pitfalls of early stopping conditions:

condition 1: hard to know exactly when to stop

condition 2: classification error may not smoothly go down, but jump down at some points, e.g. XOR operation for two features

→Tree pruning:

train a complex tree, simplify later

measure of complexity of trees: L(**T**)=# of leaf nodes

balance simplicity and predictive power:

total cost = measure of fit + measure of complexity

measure of fit: classification error, bad fit to traning data, Error(**T**)

measure of complexity: number of leaf nodes, L(**T**)

Total cost C(**T**) = Error(**T**) + λL(**T**)

if λ=0: overfitting

if λ=infinity: infinite penalty, learn a tree has no decisions in it (just the root)

y-hat is the majority class

if λ in between:

**Tree pruning algorithm:**

Start at bottom of tree T and traverse up, apply prune\_split to each decision node M

Prune\_split(T, M):

1. compute total cost of tree T using C(**T**) = Error(**T**) +λL(**T**)

2. let **T**smaller be tree after pruning subtree below M

3. compute total cost complexity of **T**smaller

C(**T**smaller) = Error(**T**smaller) +λL(**T**smaller)

4. if C(**T**smaller) < C(**T**), prune to **T**smaller

(4) Handling missing data

missing values impact training and predictions:

training data: contains “unknown” values

predictions: input at prediction time contains “unknown” values

①Strategy 1: purification by skipping

idea 1: skip data points with missing values

may lead to too much missing data

make sure only a few data points are skipped

idea 2: skip features with missing values

applied to those datasets, in which all the missing values occur in certain feature

need to make sure only a few features are skipped

pros

easy to understand and implement

can be applied to any model(decision trees, logistic regression, linear regreesion, …)

cons

removing data points and features may remove important information from data

unclear when it's better to remove data points versus features

doesn’t help if data is missing at prediction time

②Strategy 2: prificaton by imputing

get a best guess of what the missing value might be – fill in missing values

idea 1: categorical features use model

replace missing value with most common value of non-missing **x**i

idea 2: numetical features use average or median

average or median value of non-missing **x**­i

there are also some more advanced methods exist

pros:

easy to understand and implement

can be appleid to any model

can be used at prediction time: use same imputation rules

cons:

may result in systematic errors

e.g. feature “age” missing in all blanks in Washington by state law, this method will make everyone in Washington are thought to have the same age

③Strategy 3: adapt learning algorithm to be robust to missing values (most reliable)

associate missing values with a branch

e.g. when the income is unknown, then choose the branch “low”

for every decision node, includes choice of response to missing values

Feature split selection with missing data:

in the step 2 of greedy decision tree learning algorithm- “select a feature to split data”, also select a branch for missing values

choose the branch that leads to lowest classification error

put unknowns into each brand once, and calculate classification error each time

pros:

addresses traning and prediction time

more accurate predictions

cons:

requires modification of learning algorithm

→**Feature split selection algorithm with missing value handling**

given a subset of data M (a node in a tree)

for each feature hi(x):

1. split data points of M where hi(x) is not “unknown” according to feature hi(x)

2. consider assigning data points with “unknown” value for hi(x) to each branch

A. compute classfication error split and branch assignment of “unknown” values

choose feature h\*(x) and branch assginment of “unknown” with lowest classification error

**Week 5 Boosting**

(1) Boosting

apply to any classifier

make classifiers better and better by combining multiple classifiers

start from a weak classifier (e.g. a simple logistic regressor)

- low variance and learning is fast, but have high bias

make it stronger:

option1: add more features or depth

option2: boosting

(2) Ensemble classifiers

each classifier “votes” on prediction

for a certain input **xi** compute f1(**xi**), f2(**xi**), f3(**xi**), … for each simple classifier

ensemble model F(**x**i) = sign(w1 f1(**xi** + w2f2(**xi**) + w3f3(**xi**) + …)

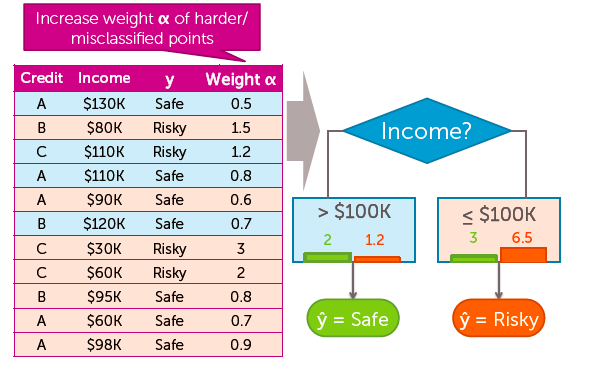
w1, w2, … is the learned coefficients

learning simple classifiers on weighted data:

more weight on “hard” or more important points

weighted dataset: each **x**i, yi weighted by αi, more improtant point = higher weight αi

during learning, data points j counts as αi data points



(3) AdaBoost

learning ensemble:

start same weight for all points: αi = 1/N

for t = 1, …, T

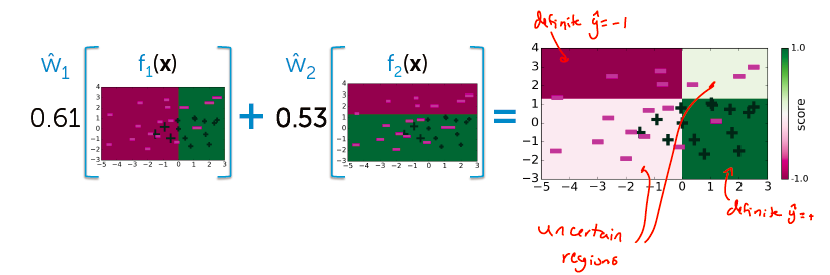
learn ft(**x**) with data weights αi

compute coeeficient wt-hat ( Problem 1: how much do I trust ft, refer to (4))

recompute weights αi (Problem 2: weight mistakes, refer to (5))

normalize weights αi (refer to (6))

final model predicts by:



(4) compute coefficient wt-hat

computing coefficient wt-hat of classifier ft(x)

is ft(x) good?

yes(ft has low training error)- wt-hat is large

no- wt-hat is small

to justify whether ft(x) is good:

measuring error in weighted data (weighted number of misclassified points)

to calculate weighted classification error = total weight of mistakes / total weight of all data points

when weighted error is 0.01: w-hat is 2.3

when weighted error is 0.5: w-hat is 0 (ignore)

when weighted error is 0.99: w-hat is -2.3 (do exactly opposite)

(5) reweighing data to focus on mistakes

updating weights αi based on where classifier ft(x) makes mistakes

does ft get **x**i right?

yes- decrease αi

no- increase αi

(6) Normalizing weights αi

If **x**i often mistake, weight αi gets very large

If **x**i often correntc, weight αi gets very small

but this can cause numetical instability after many iterations

→normalize weights to add up to 1 after iteration:

(7) Learning boosted decision stumps with AdaBoost

finding best next decision stump:

try splitting on each feature and measure the weighted error

choose the one with lowest weighted error

(8) The bossting theorem

after some iterations, training error of boosting goes to 0

**AdaBoost Theorem:**

under some technical conditions, training error of boosted classifier goes to 0 as T goes to infinity

condition = at every t, can find a weak learner with weighted error(ft) < 0.5

not always possible

(9) Overfitting in boosting

For decision trees, as tree depth increases, test error decreases and then increases

For boosting, as trees increse, test error tends to stay at a constant

→boosting tends to be robust to overfitting

but boosting will eventually overfit, so must choose max number of components T

choosing T:

use a validation set if you have a lot of data

cross-validation if the dataset is relatively small

(10) variants of boosting and related algorithms

other boosting algorithms:

gradient boosting- like AdaBoost, but useful beyond basic classification

random forests-

bagging: pick random subsets of the data

learn a tree in each subset

average predictions

simpler than boosting and easier to parallelize

typically higher error than boosting for same number of trees

impact of boosting

among most useful ML methods ever created

extremely useful in computer vision

used by most winners of ML competitions

most deployed ML systems use model ensembles

**Week 6 Precision & Recall**

(1) Accuracy

accuracy may not be the best metric for classification in many occasions

what is good accuracy?-

better than binary classification (classfication error 0.5)

is 90% accuracy good?- it depends (if 90% is negative and only those are found, then it does not make sense for finding positive reviews)

→Precision and recall

Precision: Did I (mistakenly) show a negative sentence

Recall: Did I not show a (great) positive sentence

(2) Precision

**Fraction of positive predictions that are actually positive**

for all sentences predicted to be positive (y-hat = +1),

find how much of them are actually positive (false positive)

precision = # true positive / (# true positive + # false positive)

best possible value: 1.0

worst possible value: 0.0

(3) Recall

**Fraction of positive data predicted to be positive**

for all the reviews, find how much of the positive reviews

are predicted to be positive (true positive)

recall = # true positive / (#true positive + # false negative)

best possible value: 1.0

worst possible value: 0.0

(4) Precision-recall extremes

①Optimistic model

high recall, low precision

predict almost everything as positive

②Pessimistic model

high precision, low recall

prediction positive only when very sure

(5) Trade off between precision and recall

how confident is your prediction- probability predicted

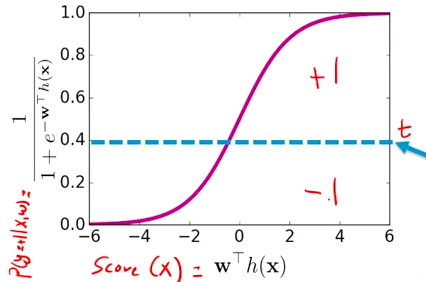
probability can be used to trade off between precision an recall

change the threshold of classification

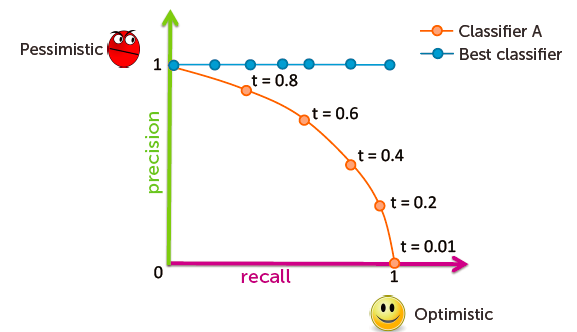
original: if P>0.5: +1, else: -1

e.g. if P > 0.999: +1, else: -1 – pessimism

e.g. if P > 0.001: +1, else: -1 – optimism



Preicsion-recall curve:



to compare algorithms:

often, reduce precision-recall to single number to compare algorithms

F1 measure, area-under-the-curve(AUC), …

→Preicision at k

if I want to show k=5 positive reviews on website

want my precision for the five top sentences be be as good as possible

**Week 7 Scaling to Huge Datasets & Online Learning**

(1) Scaling to huge datasets

drawbacks of gradient ascent:

gradient ascent is slow, since every update requires a full pass over data

but there are more and more data these days

→scalable machine learning & stochastic gradient

Stochastic gradient ascent

take your massive dataset and your current parameters **w­**(t)

when you’re computing the gradient, instead of looking at all the data,

just looks at a small subset of the data, and update the coefficient **w­**(t),

then look at a little bit more data, and update the coefficient **w­**(t), …

therefore: many updates for each pass over data

(2) Stochastic gradient ascent

standard gradient ascent:

need to sum over data points

Stochastic gradient ascent:

just use one data point, not all

every time we’re going to do an update, we use a different data point

Stochastic gradient ascent algorithm for logistic regression:

init **w**(1) = 0, t = 1

until converged

for i = 1, …, N

for j = 0, …, D

partial[j] =

**w**j(t+1)= **w**j(t) +η partial[j]

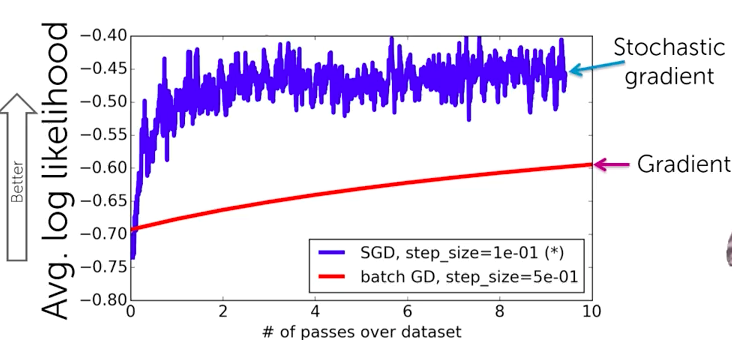
t = t + 1

operation time for each weight update, gradient ascent is O(N2), while for stochastic is O(N)

but more updates are needed

besides, stochastic gradient is much more sensitive to the choice of parameters like the choice of step size

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time per iteration** | **Total time in theory** | **Total time in practice** | **Sensitivity to parameters** |
| Gradient | Slow for large data | slower | Often slower | Modeate |
| Stochastic  gradient | Always fast | faster | Often faster | Very high |



Stochastic gradient achieves higher likelihood sooner, but it’s noiser

since each iteration, the gradient goes in one direction from 1 data point

most ierations increases likelihood, but sometimes decrease it

(gradient ascent finds direction of steeps ascent by suming over all data points)

**Order of data can introduce bias**

if all the negative points come first, then it will push the weights towards negative all along, and will introduce some bad behaviors to the algorithm

→we need to shuffle data before running stochastic gradient

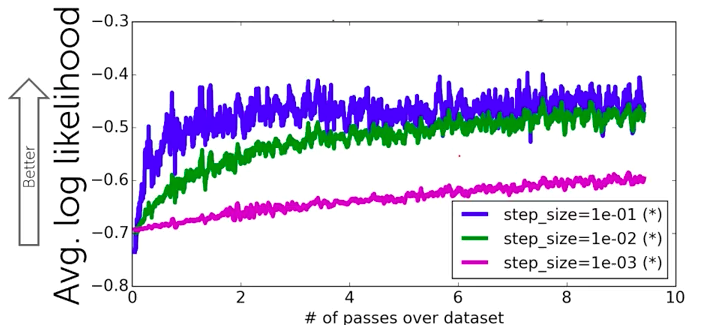
(3) Choosing stepsize η

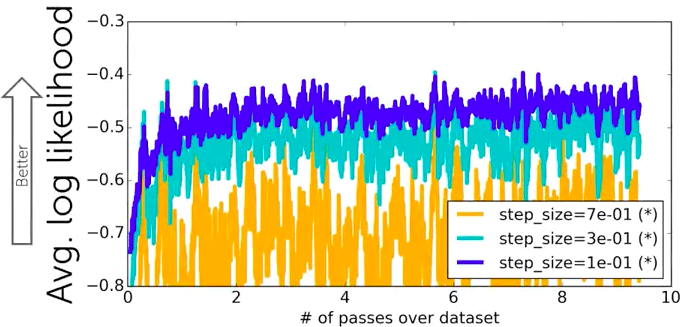
picking step size for stochastic gradient is very similar to picking step size for gradient, but it is more unstable

the blue curve below is the best choice

if step size is too small, stochastic gradient slow to converge

if step size is too large, stochastic gradient oscillates





simple rule of thumb for picking step size η similar to gradient

picking step size requires a lot a lot of trial and error, much worst than gradient

try a several values, exponentially spaced

goal: plot learning curves to

find one η that is too small

find one η that is too large

advanced tip: step size that decreases with iterations is very important for stochastic gradient

(4) Determine the coefficients

since Stochastic gradient never fully “converges”, but eventually oscillate around a solution, the last coefficients may be really good or really bad

→Stochastic gradient returns average coefficients:

minimize noise: don’t return last learned coefficients

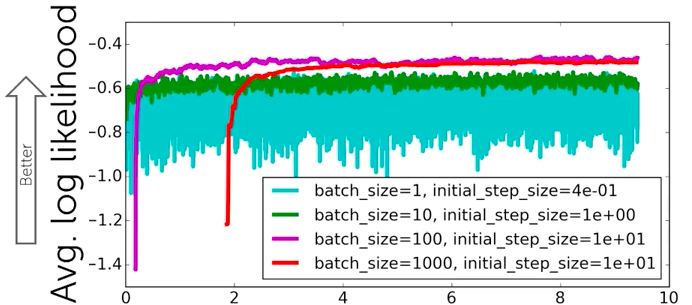
instead, output average:

use the average of all the coeffiicients that have computed

(5) Learning from batches of data

since learning from a single data point will make the solution noisy, you could learn from batches of fata (mini-batch)

less noisy and oscillates better in final stage



for the batch size:

too large: slow convergence

too small: noisy, may not converge

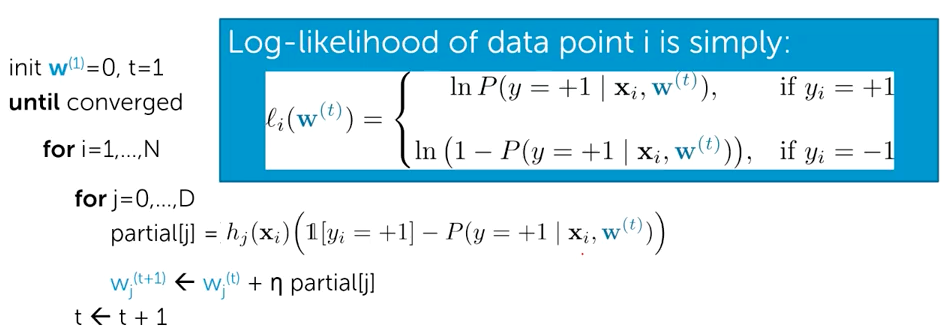
(6) Measuring convergence

making the average log-likelihood v.s. passes over dataset plot is really slow, since we need to compute l(**w**) for each iteration (including product over all data points)

but we could compute log-likelihood during run of stochastic gradient ascent:

what we need to compute partial[j] is exactly what is needed to compute log-likelihood of single data points

so we can estimate log-likelihood with calculating the moving average of the last k values of log-likelihood of single data points



(7) Adding regularization

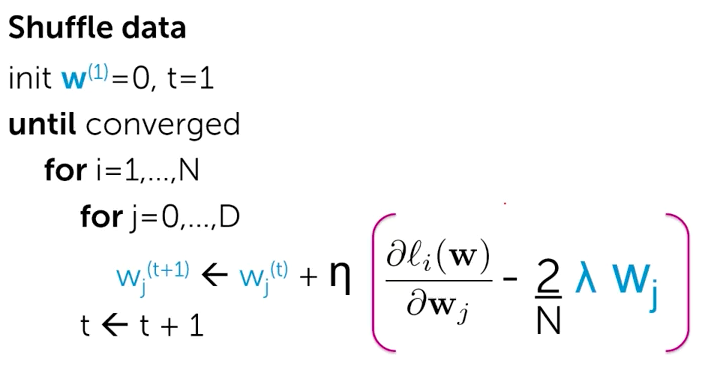
total quality = measure of fit – measure of magnitude of coefficient

measure of fit: l(w)

measure of magnitude of coefficient: lambda\*||w||2 (L2 penalty)

totalderivative = (for Stochastic gradient ascent)

Stochastic gradient ascent with regularization:



(8) Online learning

fitting models from streaming data

batch learning v.s. online learning

batch learning: all data is available at start of traning time

online learning: data arrives (streams in) over time, must train model as data arrives

example for online learning:

e.g. ad targeting- once you click a website, it immediately returns personalized ads, and when you click a certain ad, the true output is feed into the model

online learning

data arrives over time

must make a prediction every time new data point arrives

observe true class after prediction made

want to update parameters immedately

Stochastic gradient ascent can be used for online learning

pros:

model always up to date →often more accurante

lower computational cost

don’t need to store all data, but often do anyway

cons:

overall system is much more complex (bad real-world cost to build and maintain)