iPython notebook

Python + wikipages

when editing .ipynb files:

you can write codes in cells and run them immediately

you can write markdowns by changing the type to ‘Markdown’ and then run it

using graphlab

e.g.

import graphlab

sf = graphlab.SFrame('people-example.csv')

documentation:

https://dato.com/products/create/docs/generated/graphlab.SFrame.html

Regression- predicting house prices

(1) Regression process

①look at recent sales in neighborhood

plot recent house sales-

y- prices

x- square feet

normally no houses with exactly the same square feet

②linear regression modeling

modeling the relationship between square feet and prices

→fit a line through the data f(x) = w0 + w1x

residual sum of square (RSS)

look at how far each observation away from our line

then sum the square of each disparity

then minimize it to get best estimation of m0 and m1

then predit the house price at my square feet

③quadratic function modeling

f(x) = w0 + w1x + w2x2

then minimize RSS

④why not higher exponents? – adding higher order effects

still try to minimize RSS

buy only minimizing RSS may not seem reasonable in real sense- overfitting

(2) Evaluating regression models

①evaluating overfitting via training/test split

overfitting- minimize RSS but bad predictions

→how to choose order/complexity (quadratic or higher?)

**simulate predictions**

remove some houses

fit models on remaining

predict heldout houses and compare with the actual observed values

**training set:** the houses used to fit the model

**test set:** using as proxy for prediction

**training error**(w) =∑(actual value of training unit – predicted value of training unit)2

try to minimize w

decrease with model complexity

**test error**() =∑(actual value of test unit – predicted value of test unit)2

first decrease then increase with model complexity

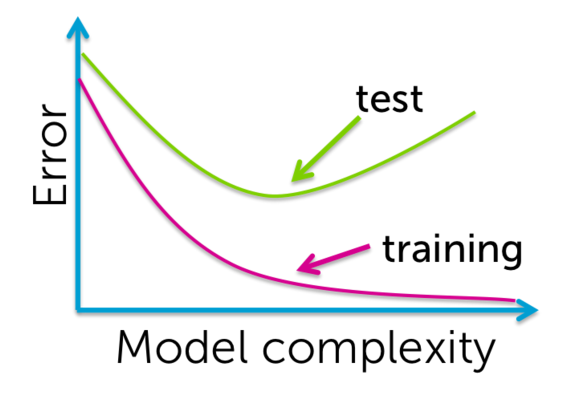
②adding other features

whether there are other influential features- e.g. number of bathrooms

→add number of bathrooms as the second feature, and plot in three-dimension

f(x) = w0 + w1x + w2#bath find a slice in the space

→how many features to use?



(3) predicting house prices using graphlab

import dataset

import graphlab

dataset = graphlab.SFrame(‘file\_route’)

visualize dataset

sales.show(view=’plot type’, x=’factor’, y=’factor’)

split training dataset and test dataset

train\_data, test\_data = sales.random\_split(percentage of training set,seed=0)

calculate mean of one feature of test dataset

print test\_data[‘factor’].mean()

build a linear regression model

model = graphlab.linear\_regression.create(traindata, target=’y-factor’, features=[list of x-factors])

show coefficients

print model.get(‘coefficients’)

evaluate a model(calculate RMSE)

print model.evaluate(test\_data)

(RMSE = (RSS/N)1/2 : to avoid RSS increase by number of factors)

Classification- Analyzing sentiment

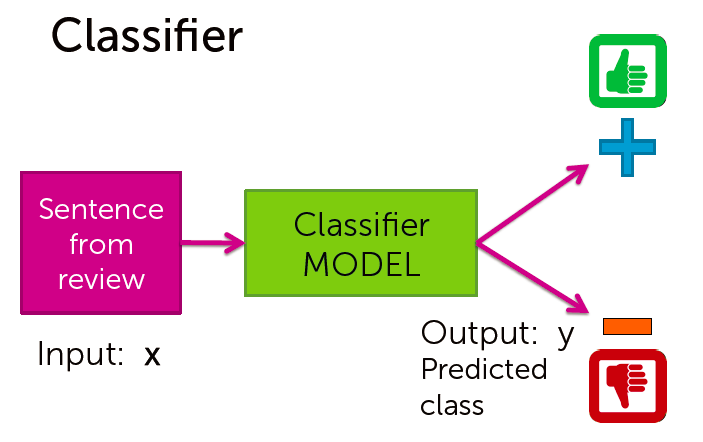
(1) Classification process

read all reviews, and pick all sentences about sushi,

and then put them into sentence sentiment classifier to get an average prediction

→need a **classifier application**

input x (sentence from review) into a classifier model to get an output y(predicted class)

examples:

webpages classification; spam filtering; image classification; personalized medical diagnosis

①simple threshold classifier

count positive& negative words in sentence

if number of positive words > number of negative words: positive prediction

limitation:

how do we get list of positive/negative words

words have different degrees of sentiment

single words are not enough (not good is negative)

the first two problems are addressed by learning a classifier

the last one is addressed by more elaborate features­

②simple linear classifier

use training data to learn a weight for each word (solve the above two limitations)

and compute the score, if >0: positive

esp. decision boundaries-

when 2 weights are non-zero: the line defines what’s positive and what’s negative(the line score = 0)

when 3 weights are non-zero: plane

more: hyperplane

(2) Evaluating classification models

①training a classifier- learning the weights

learn classifier using training set

then test how much of the test set are given the wrong prediction

(e.g. put(Sushi was great) into the learned classifier and see whether predicted positive)

error measures fraction of mistakes:

**error = # of mistakes/ total # of test sentences**

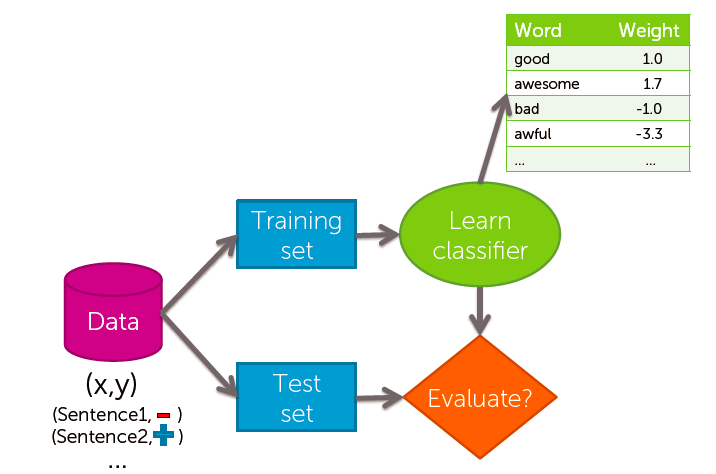
best possible value is 0.0

accuracy measures fraction of correct prediction:

**accuracy = # of correct / total # of test sentences**

best possible value is 1.0

error + accuracy = 1



good accuracy:

compare against random guessing- what if you ignore the sentence and just guess

for k classes, accuracy = 1/k

should beat the random guess

is a classifier with 90% accuracy good?- it depends.

e.g. 90% of the emails sent are spam, just guess all are spam get 90% accuracy

amazing performance when there is class imbalance

asking hard questions about reported accuracies

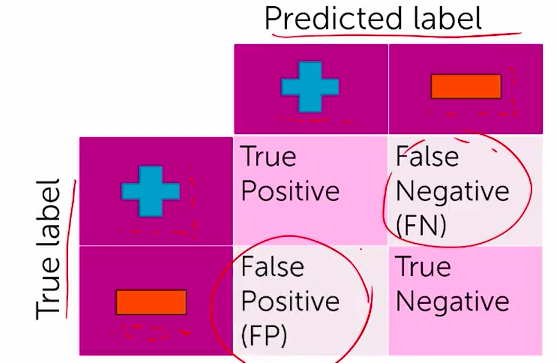
is there class imbalance

how does it compare to a simple baseline approach (e.g. random guessing, majority class)

most importantly- what accuracy does my application need (what is good enough for my user’s experience? what is the impact of the mistake we make?)

②types of mistakes

**confusion matrix:**



mistakes

False Positive- should be false but predicted as true

False Negative- should be true but predicted as false

cost of different types of mistakes can be different in some applications

e.g. Spam filtering- FP has higher cost for email lost

e.g. medical diagnosis- FN means disease not treated, FP means wasteful treatment

③learning curves

how much data does a model need to learn?

the more the merrier- but data quality is most important factor

theoretical techniques sometimes can bound how much data is needed

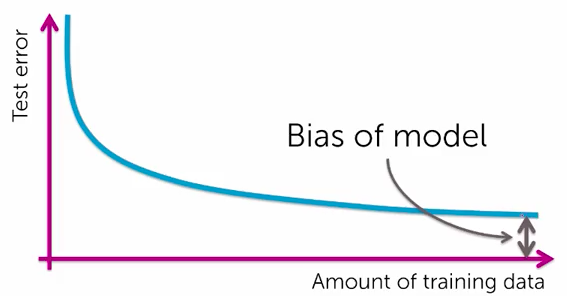
typically too loose for practical application, but provide guidance

in practice

more complex models require more data

empirical analysis can provide guidance

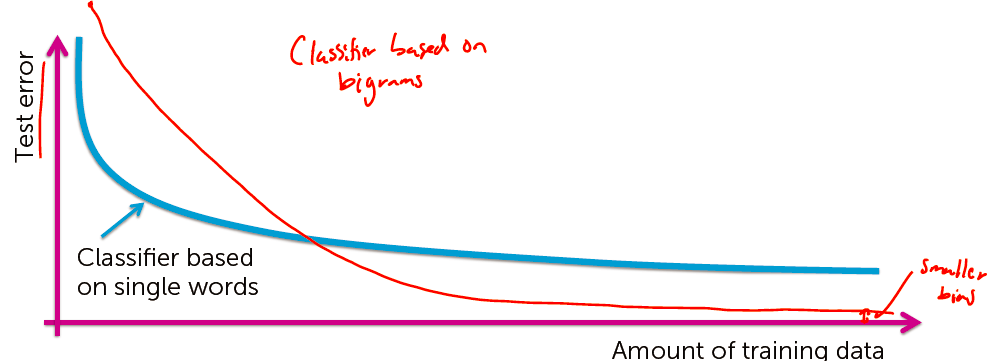
→learning curve (relationship between test error and amount of training data)



the bias of model can not be reduced to 0 even with infinite data

only consider single words (so mistreat ‘not good’)

→classify based on pair of words (bigrams) can be better



④class probabilities

many classifiers provide a confidence level: P(y|x)

y- output label, x-input sentence

e.g. P(y=+|x) = 0.99 means the probability that the sentence x is positive is 99%

(4) analyzing using graphlab

import graphlab

dataset = graphlab.SFrame(file\_route)

add a column ‘word\_count’ counting words (in dictionary type)

dataset['word\_count'] = graphlab.text\_analytics.count\_words(dataset[text\_column])

show the frequency of each rating

dataset['rating'].show(view='Categorical')

ignore all records rated as 3(neither positive nor negative)

dataset = datset[dataset['rating'] != 3]

add a column ’sentiment’ whose value is 0-1 for positive/negative rating

dataset[‘sentiment’] = dataset[‘rating’] >= 4

build sentiment model

train\_data,test\_data = dataset.random\_split(.8, seed=0)

sentiment\_model = graphlab.logistic\_classifier.create(train\_data, target=target\_column, features=list\_of\_features, validation\_set=test\_data)

see the evaluation of the model and the accuracy

sentiment\_model.evaluate(test\_data)

see the graph of relationship between FP and FN

sentiment\_model.show(view='Evaluation')

see the weight(co-efficiency) on each feature

print sentiment\_model['coefficients']

extract a subset of data for one single group

subset = dataset[dateset[‘name’] == group\_name]

see the rows of this subset(number of records)

print len(subset)

predict sentiment on all of the records in the subset

subset['predicted\_sentiment'] = sentiment\_model.predict(subset, output\_type='probability')

sorting the subset descending by predicted sentiment

subset = subset.sort('predicted\_sentiment', ascending=False)

Clustering and similarity- Retrieving documents

(1) Similarity process

how do we measure similarity?

how do we search over articles?

①word count representation for measuring similarity

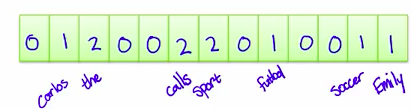
bag of words model

ignore order of words present in the article

count number of instances of each word in vocabulary

define a vector over the vocabulary presenting frequency of each word

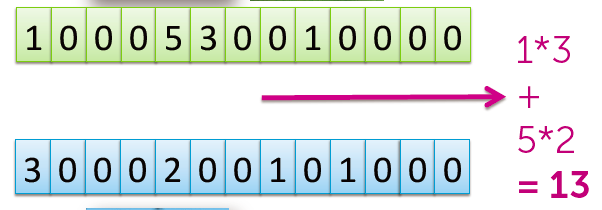
→word-count vector



②measuring similarity

look at an element-wise product over the vector

for each element on the same position, multiply the number



③issues with word counts-

(a)document length

the longer, the more likely to show higher product

solution→ **normalize**

normalize word count vector (compute the norm of the vector)

(b)rare words

common words like ‘the’ appears frequently and they dominate the similarity of the metric

therefore, we need to increase the importance of key rare words

solution→ **prioritizing important words (with tf-idf)**

what characterizes a rare word-

appear infrequently in the corpus

emphasize words appearing in few documents, or discount word weight based on the number of documents containing that word in corpus

do we want only rare words to dominate?-

we want to emphasize on important words of those rare words

what characterize an important word?-

appear frequently in document**(common locally)**

appear rarely in corpus **(rare globally)**

trade off between local frequency and global rarity

→**Term frequency - inverse document frequency (tf-idf)**

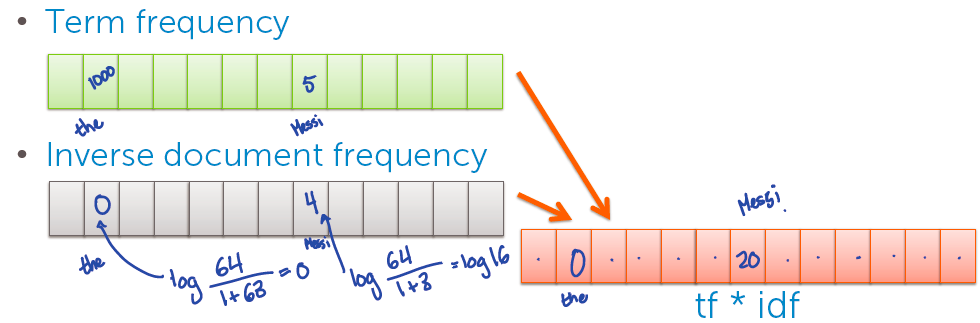
term frequency- count the words locally

inverse document frequency- downward factor based on inverse document frequency, look at all documents, and calculate

**log(number of documents/documents containing the word)**

then multiply every element in term frequency by that in inverse document frequency

thus common words get downward, and important rare words get upward

④nearest neighbor search

have a query article and a corpus (database to search)

specify a distance metric

(a) we need to output a set of similar article

→nearest neighbor search

search over each article in corpus

compute the similarity between the query and each article (using above algorithm)

if the similarity is better than the best similarity that we found so far, then we keep this article as the best article we found, and loop to find the best article

(b) we need to output list of k similar articles (top k most relevant articles)

→k-nearest neighbor

keep a priority queue of the top k articles found so far

(2) Clustering process

①**supervised learning problem**

when we have bunches of labeled documents

training set of labeled documents

then when we have another article, it turned out to be a multiclass classification problem- suit it into one of the classifications we learned

**②unsupervised learning problem**

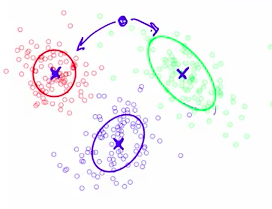
no labels provided and we want to uncover cluster structure

input each document as vectors and output cluster labels- **clustering process**

cluster defined by **center** and **shape/spread**

we assign observation to cluster by score it to each cluster

sometimes, we just use the cluster centers to measure



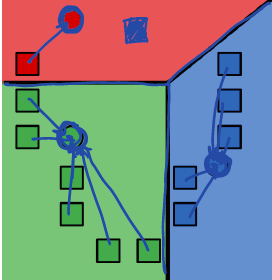
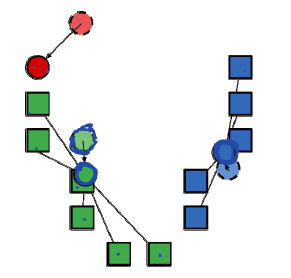
**K-means clustering algorithm**

initialize cluster center

assign every observations to closest cluster center

revise cluster centers as mean of assigned observations

repeat above two steps until convergence



(3) Clustering using graphlab

import graphlab

dataset = graphlab.SFrame(‘file\_route’)

explore a document in the dataset

document = dataset[dataset[‘column\_name’] == ‘value’]

get the word counts for the document

document[‘word\_count’] = graphlab.taxt\_analytics.count\_words(document[‘text’])

sort the word counts for the document (turning a dictionary into a table)

new\_table = document[[‘word\_count’]].stack(‘word\_count’, new\_column\_name = [‘word’, ‘count’])

new\_table.sort(‘count’, ascending = False)

compute TF-IDFs for the corpus

dataset[‘word\_count’] = graphlab.taxt\_analytics.count\_words(dataset[‘text’])

tfidf = graphlab.text\_analytics.tf\_idf(dataset[‘word\_count’])

dataset[‘tfidf’] = tfidf

examine the tdidf for certain document

document = dataset[dataset[‘column\_name’] == ‘value’]

document[‘tfidf’].stack(‘tfidf’, new\_column\_name = [‘word’, ‘tfidf’]).sort(‘tfidf’, ascending = False)

computing distances between a few documents

document2 = dataset[dataset[‘column\_name’] == ‘value2’]

document3 = dataset[dataset[‘column\_name’] == ‘value3’]

use cosine method- the lower the closer (maximum to 1)

graphlab.distance.cosine(document2[‘tfidf’][0], document3[‘efidf’][0])

build a nearest neighbor model

knn\_model = graphlab.nearest\_neighbors.create(dataset, features=[‘efidf’], label=’name’)

apply the nearest-neighbors model for retrieval- which is closest to document1

knn\_model.query(document1)

Recommender systems- Recommending products

(1) Recommendation process

In modern day, recommendation should be up-to-date, coherent and diverse.

to build a recommender system:

①solution 0: **popularity**

recommend most-bought products

but lack personalization

②solution 1: **classification model**

use user info, purchase history and product info to classify whether a person like/dislike a product

pros: personalized; features can capture context- time of the day, what I just saw etc.; even handle limited user history

cons: features may not be available, often doesn’t perform as well as collaborative filtering methods

③solution 2: **collaborative filtering**

people who bought this also bought…

using **co-occurrence matrix** to make recommendations

matrix C: store number of users who bought both items i & j, a square matrix of number of items- a symmetric matrix

when user purchased diapers, we look at diapers row of matrix and recommend other items with large counts

attention: **Co-occurrence matrix must be normalized**

e.g. diapers can be a popular baby item that for any item, it will count largely. everybody buys diapers does not personalize one’s needs.

results of no normalizing- drowns out other effects and recommend based on popularity

solution- normalize co-occurrence using **Jaccard similarity**

normalizes by popularity: (who purchased I and j) divide by (who purchased i or j)

limitations- only current page matters, no history (recommend similar items based on the one you just bought)

→improve by **(weighted) average of purchased items**

this is the solution to the limitation of collaborative filtering- to get history involved

look at the history i, and compute user-specific score for each item j in inventory by calculating the average of similarities between j and each item i

sort scores for each item j and find the one with highest similarity

limitations:

does not utilize context(e.g. time of day), user features, product features etc.

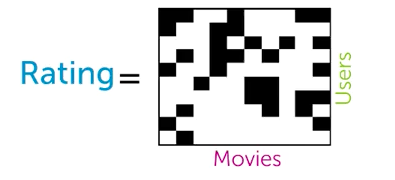
cold star problem- what if a new user or product arrives

④solution 3: **matrix factorization model**

e.g. user watch movies and rate them

we make a users \* movies matrix, data in the matrix is the user score of the movie, and if the rating is known, then it is shown below and black cells, otherwise white.

we wan to fill missing data.



process:

describe movie v with topics Rv, such as action, romance, drama to [0.3, 0.01, 1.5]

describe user u with topics Lu, such as [2.6, 0, 0.8]

rating(u, v)-hat is the sum of element wise- product of the two vectors Rv and Lu

just the multiply of single vector L and R

then we look at the difference of rating(u, v) and rating(u, v)-hat

RSS(L, R) = ∑(actual rating – rating-hat)2

(sum for every (u,v) pair where rating is available)

find the user vector and movie vector that make the minimum RSS

limitation of matrix factorization model

cold-start problem- the model still cannot handle a new user or movie

→bring it all together- **featurized matrix factorization**

features capture context like time of day, user info, past purchase

discovered topics from matrix factorization capture groups of users who behave similarly

so we can combine both to mitigate cold-start problem-

rating for a new user from features only

as more information about user is discovered, matrix factorization topics become relevant

(2) evaluate performance for different recommender systems

why not use classification accuracy-

we try to figure out whether user like or dislike one product

but normally, what we want to know is only a small subset of products that a certain user likes

we may even get the result that the user dislike any of them

to evaluate a recommender systems-

highlight those products been recommended

then ask how many of the items user likes has been recommended

**recall = number of products liked & shown / number of products liked**

**precision = number of products liked & shown / number of products shown**

to maximize recall: recommend everything, but this causes very small precision

optimal recommender: recall = precision = 1

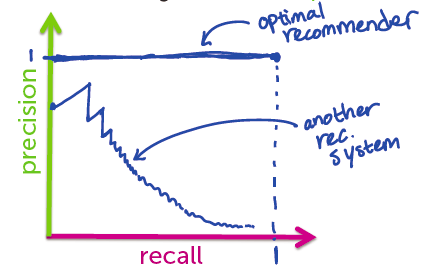
→precision-recall curve

input a specific recommender system

output algorithm-specific precision-recall curve

to draw curve, vary threshold on number of items recommended

for each setting, calculate the precision and recall



for a given precision, we want recall as large as possible,

therefore, we want largest area under the curve (AUC)

or, we can set desired recall and maximize precision

(3) Recommending using graphlab

import graphlab

dataset = graphlab.SFrame('file\_route')

count number of unique users in the dataset

users = dataset[‘user\_id’].unique()

print len(users)

creating & evaluating a popularity-based recommender

train\_data,test\_data = song\_data.random\_split(.8,seed=0)

popularity\_model = graphlab.popularity\_recommender.create(train\_data, user\_id='user\_id', item\_id='product')

recommend products for a certain user

popularity\_model.recommend(users=[users[0]])

creating & evaluating a personalized recommender

personalized\_model = graphlab.item\_similarity\_recommender.create(train\_data, user\_id='user\_id', item\_id='product')

recommend products for a certain user

personalized\_model.recommend(users=[users[0]])

find similar products of a certain product

personalized\_model.get\_similar\_items(['product1'])

using precision-recall to compare recommender models

%matplotlib inline

model\_performance = graphlab.recommender.util.compare\_models(test\_data, [popularity\_model, personalized\_model], user\_sample=.05)

calculate the sum of listen times for each artist

artist\_count = song\_data.groupby(key\_columns='artist', operations={'total\_count': graphlab.aggregate.SUM('listen\_count')})

calculate times to be recommended for each song

artist\_count = recommended\_songs.groupby(key\_columns='song', operations={'count': graphlab.aggregate.COUNT()})

Deep learning- Searching for images

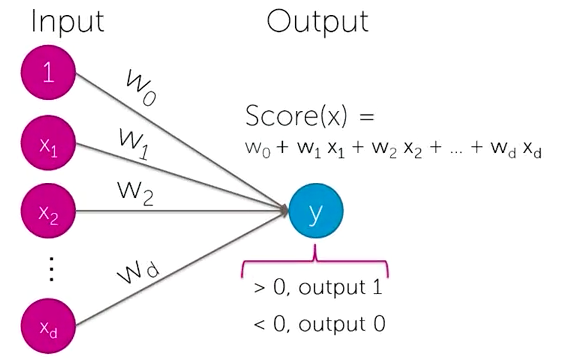
(1) non-linear classifier (neural network)

sometimes, just key in key words, does not help a lot

→image similarity can be helpful

we need to learn very non-linear features (neural network)

linear classifier:



for parameter X1 and X2, y = w1x1 + w2x2 + w3

to express X1 or X2 (if True, y >0), we can set three w to 1, 1, -0.5

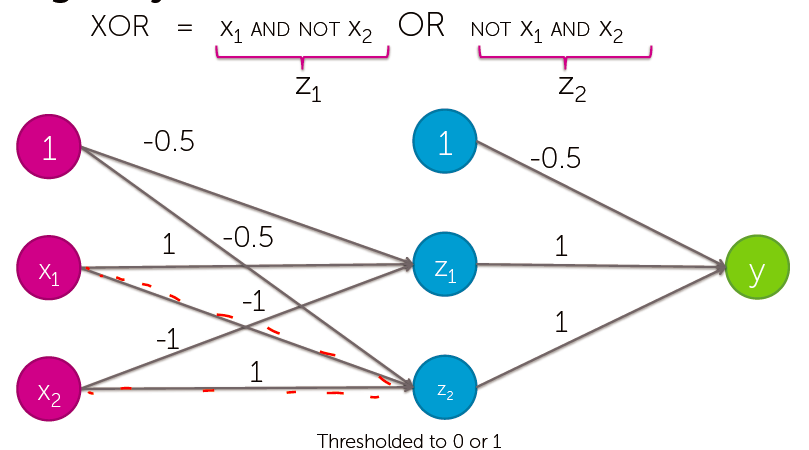
to express X1 and X2 (if True, y >0), we can set three w to 1, 1, -1.5

However, linear classifier cannot express XOR (only one of X is true)

to predict it, we need to first generate other two parameters:

Z1 = X1 and not X2, Z2 = not X1 and X2

then sue two Zs to get XOR



neural network can be transformed by layers and layers of linear models

(2) deep learning process

①hand-created features

features = local detectors, combined to make prediction

e.g. to identify whether a image is a face image, you can detect nose, eyes and mouth, and then use a neural system to determine whether it is a face

image features are collections of locally interesting points which are distinctive

a typical example of hand-created features is SIFT feature

standard image classification approach

input an image

extract features using hand-created features like SIFT

create a vector describe the image (whether each features found)

feed it into a linear classifier to make a predicition

however, hand-created features are quite complex

②→**use neural networks** to discover and learn those features automatically

input an image

run it through a multiple-layer neural network to make a prediction

deep learning process

split training set and validation set from lots of labeled data

learn deep neural network using training set

validate the network using validation set

adjust parameters, network architectures to get better network

pros of deep learning

enables learning of features rather than hand tuning

impressive performance gains

computer vision, speech recognition, some text analysis

potential for more impact

cons of deep learning

requires a lot of data for high accuracy

computationally really expensive

extremely hard to tune

choice of architecture

parameter types

hyperparameters

learning algorithm

when you don’t have a lot of data

③→**deep features**

deep features = deep learning + transfer learning

transfer learning: use data from one task to help learn on another

e.g. cats vs. dogs to more categories

there is a neural network with good accuracy to distinguish cats and dogs

we can use the features we learned to combine some simple classifier to classify new categories

the last few layers for the cat vs. dog task concentrate specifically on distinguish cats and dogs, so they should be ignored for other tasks

then we only learn end part of neural network to distinguish other categories

(3) deep features using graphlab

import graphlab

image\_train = graphlab.SFrame('file\_route')

train a classifier on the raw image pixels

raw\_pixel\_model = graphlab.logistic\_classifier.create(image\_train, target = ‘label’,

features=[‘image\_array’])

make a prediction with the simple model

image\_test[0:3][‘image’].show() show first three pictures

image\_test[0:3][‘lable’].show() show the actual label

raw\_pixel\_model.predict(image\_test[0:3]) show the predicted label

evaluate the model on test dataset

raw\_pixel\_model.evaluate(image\_test) it show the accuracy of the model

improving the model using deep features

load pre-trained model

deep\_learning\_model = graphlab.load\_model('http://s3.amazonaws.com/GraphLab-Datasets/deeplearning/imagenet\_model\_iter45')

image\_train['deep\_features'] = deep\_learning\_model.extract\_features(image\_train)

given the deep features, we can train a classifier

deep\_features\_model = graphlab.logistic\_classifier.create(image\_train, features=['deep\_features'], target='label')

again test and evaluate

deep\_features\_model.predict(image\_test[0:3])

deep\_features\_model.evaluate(image\_test)

deep features for image retrieval (similar to document retrieval)

train a nearest-neighbors model for retrieving features

knn\_model = graphlab.nearest\_neighbors.create(image\_train,features=['deep\_features'], label='id')

using image-retrieval model with deep features to find similar images

cat = image\_train[18:19] get a cat image

cat['image'].show()

knn\_model.query(cat) find similar pictures

show all similar pictures

def get\_images\_from\_ids(query\_result):

return image\_train.filter\_by(query\_result['reference\_label'],'id')

cat\_neighbors = get\_images\_from\_ids(knn\_model.query(cat))

cat\_neighbors['image'].show()

show the most similar picture

query\_cat = cat\_model.query(proxy)

highest\_query = query\_cat[query\_cat['rank'] == 1]

similar\_cat = cat\_train.filter\_by(highest\_query['reference\_label'],'id')

similar\_cat['image'].show()

calculate nearest neighbors accuracy manually

dog\_auto\_neighbors = auto\_model.query(image\_test\_dog, k=1)

dog\_dog\_neighbors = dog\_model.query(image\_test\_dog, k=1)

dog\_cat\_neighbors = cat\_model.query(image\_test\_dog, k=1)

dog\_bird\_neighbors = bird\_model.query(image\_test\_dog, k=1)

dog\_distances = graphlab.SFrame({'dog\_auto': dog\_auto\_neighbors['distance'],

'dog\_dog': dog\_dog\_neighbors['distance'],

'dog\_cat': dog\_cat\_neighbors['distance'],

'dog\_bird': dog\_bird\_neighbors['distance']})

dog\_distances.head()

def is\_dog\_correct(row):

if row['dog\_dog'] < row['dog\_auto'] and row['dog\_dog'] < row['dog\_cat'] and row['dog\_dog'] < row['dog\_bird']:

return 1

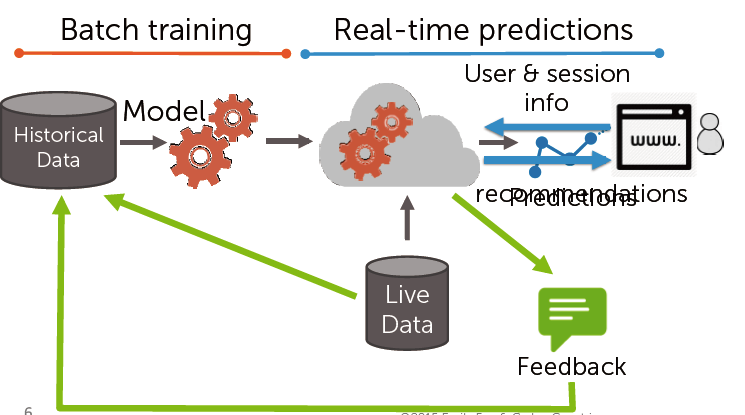
else:

return 0

dog\_distances.apply(is\_dog\_correct).sum()

Deployment

deployment system



after deployed, we evaluate, manage, and monitor it.

evaluate and track metric over time

react to feedback from deployed models

key questions

when to update a model

how to choose between existing models

→we need continuous evaluation and testing

why update

trends and user tastes change over time

model performance drops

when to update

track statistics of data over time

monitor both offline & online metric

update when offline metric diverges from online metrics or achieving desired targets

open challenges:

model selection

feature engineering/ representation

growing scaling