Web Science:
kNN and Summary

(Part 1 - Intro to kNN)

CS 432/532

Old Dominion University

Permission has been granted to use these slides from Frank McCown, Michael L. Nelson, Alexander Nwala, Michele C. Weigle

Main reference:

Ch 8 from *Programming Collective Intelligence* by Toby Segaran

(abbreviated as PCI)

code available at Ch 8 GitHub repo

Some Evaluations are Simple…

vs.

- Which is a classic car and which is a tiny car?

Some Evaluations are More Complex

vs.

- How much should a particular bottle of wine cost?
  - what is its quality?
  - how old is it?

Some Evaluations are More Complex

vs.

- Chapter 8 price model for wines: price = f(rating,age)
  - wines have a peak age
    - far into the future for good wines (high rating)
    - nearly immediate for bad wines (low rating)
  - wines can gain 5X original value at peak age
  - wines go bad 5 years after peak age

Some Evaluations are More Complex

- price = f(rating,age)
  - wines have a peak age
    - far into the future for good wines (high rating)
    - nearly immediate for bad wines (low rating)
  - wines can gain 5X original value at peak age
  - wines go bad 5 years after peak age

```
def wineprice(rating,age):
      peak_age=rating-50

      # Calculate price based on rating
      price=rating/2

      if age>peak_age:
        # Past its peak, goes bad in 5 yrs
        price=price*(5-(age-peak_age))

      else:
        # Increases to 5x original value
        # as it approaches its peak
        price=price*(5*((age+1)/peak_age))

      if price<0: price=0

      return price
```

numpredict.py

Generate Some Wine Prices

`wineprice(rating,age)`

```
wineprice(95.0,3.0)
21.11111111111114

wineprice(95.0,8.0)
47.5

wineprice(99.0,1.0)
10.102040816326529
```

good wine, but not peak age = low price

```
wineprice(20.0,1.0)
0

wineprice(30.0,1.0)
0
```

skunk water

```
wineprice(50.0,1.0)
112.5

wineprice(50.0,2.0)
100.0

wineprice(50.0,3.0)
87.5
```

middling wine, but peak age = high price

Build the Wine Price Dataset

```
def wineset1():
  rows=[]
  for i in range(300):
    # Create a random age and rating
    rating=random()*50+50
    age=random()*50

    # Get reference price
    price=wineprice(rating,age)

    # Add some noise
    #price*=(random()*0.2+0.9)
    price*=(random()*0.4+0.8)

    # Add to the dataset
    rows.append({'input': (rating,age),
                 'result': price})
  return rows
```

```
wineset1()

data[0]
{'input': (89.232627562980568, 23.392312984476838),
'result': 157.65615979190267}

data[1]
{'input': (59.87004163297604, 2.6353185389295875),
'result': 50.624575737257267}

data[2]
{'input': (95.750031143736848, 29.800709868119231),
'result': 184.99939310081996}

data[3]
{'input': (63.816032861417639, 6.9857271772707783),
'result': 104.89398176429833}

data[4]
{'input': (79.085632724279833, 36.304704141161352),
'result': 53.794171791411422}
```

[numpredict.py]()

Predict How Much is *This* Bottle Worth

- Use k-nearest neighbors algorithm
- Find the *k* “nearest neighbors” to the item in question and average their prices. Your bottle is probably worth what the others are worth.

- Questions:
  - how big should *k* be?
  - what dimensions should be used to judge “nearness”

k=1, Too Small

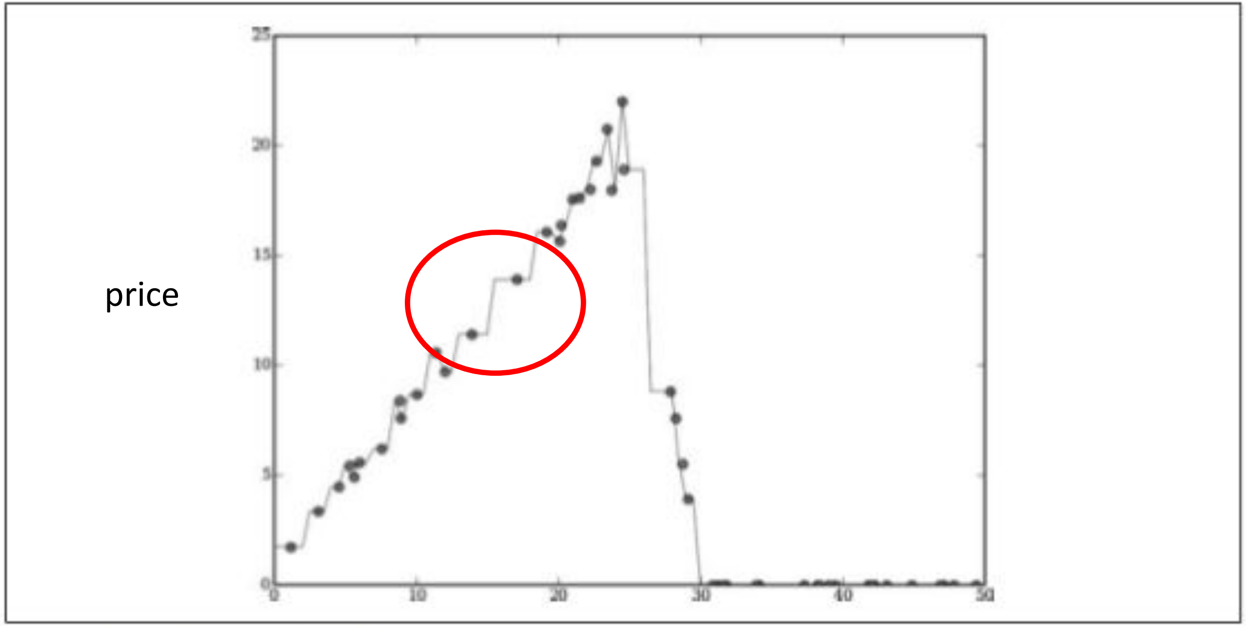


Figure 8-1. kNN using too few neighbors

From *PCI*

k=20, Too Big

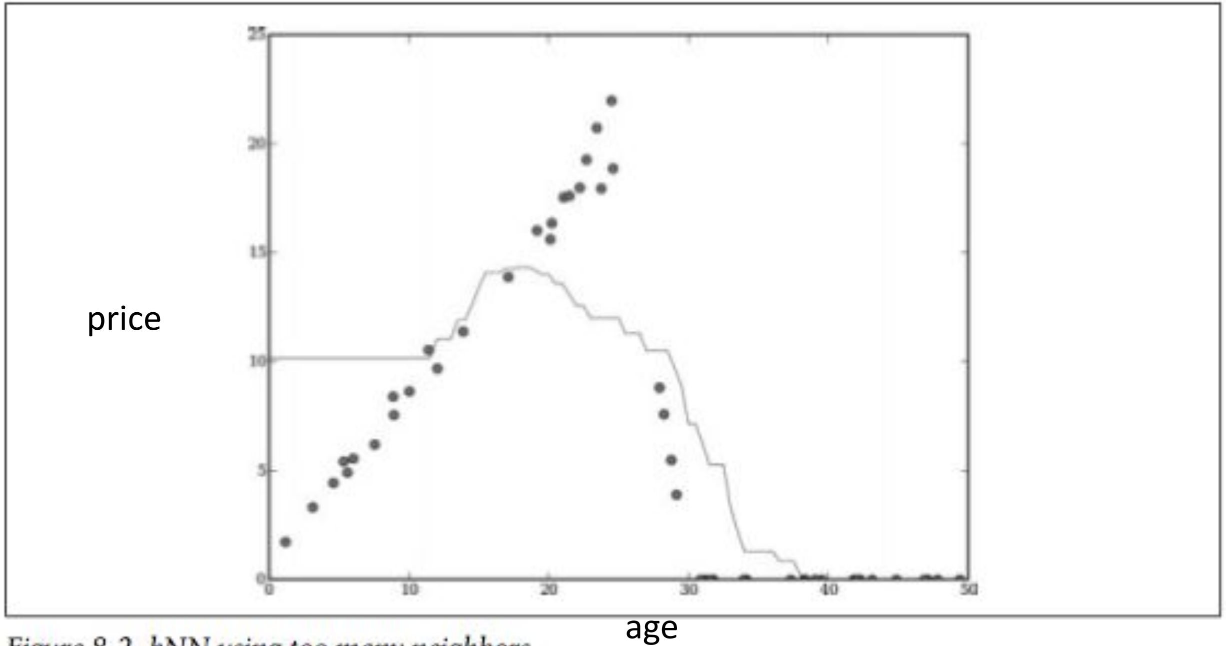


Figure 8-2. kNN using too many neighbors

From *PCI*

Define “Nearness” as f(rating,age)

```
data[0]: {'input': (89.232627562980568, 23.392312984476838), 'result': 157.65615979190267}
data[1]: {'input': (59.87004163297604, 2.6353185389295875), 'result': 50.624575737257267}
data[2]: {'input': (95.750031143736848, 29.800709868119231), 'result': 184.99939310081996}
data[3]: {'input': (63.816032861417639, 6.9857271772707783), 'result': 104.89398176429833}
data[4]: {'input': (79.085632724279833, 36.304704141161352), 'result': 53.794171791411422}
```

```
def euclidean(v1,v2):
 d=0.0
 for i in range(len(v1)):
   d+=(v1[i]-v2[i])**2
 return math.sqrt(d)
```

```
euclidean(data[0]['input'],data[1]['input'])
35.958507629062964
euclidean(data[0]['input'],data[2]['input'])
9.1402461702479503
euclidean(data[0]['input'],data[3]['input'])
30.251931245339232
euclidean(data[0]['input'],data[4]['input'])
16.422282108155486

euclidean(data[1]['input'],data[2]['input'])
45.003690219362205
euclidean(data[1]['input'],data[3]['input'])
5.8734063451707224
euclidean(data[1]['input'],data[4]['input'])
38.766821739987471
```

numpredict.py

Calculate Distance to Every Item

```
def getdistances(data,vec1):
 distancelist=[]
  # Loop over every item in the dataset
 for i in range(len(data)):
   vec2=data[i]['input']

   # Add the distance and the index
   distancelist.append((euclidean(vec1,vec2),i))
  # Sort by distance
 distancelist.sort()
 return distancelist
```

numpredict.py

kNN Estimator

```
def knnestimate(data,vec1,k=5):
  # Get sorted distances
  dlist=getdistances(data,vec1)
  avg=0.0

  # Average of the top k results
  for i in range(k):
    idx=dlist[i][1]
    avg+=data[idx]['result']
  avg=avg/k
  return avg
```

```
knnestimate(data,(95.0,3.0))
21.635620163824875
wineprice(95.0,3.0)
21.111111111111114

knnestimate(data,(95.0,15.0))
74.744108153418324
knnestimate(data,(95.0,25.0))
145.13311902177989
knnestimate(data,(99.0,3.0))
19.653661909493177
knnestimate(data,(99.0,15.0))
84.143397370311604
knnestimate(data,(99.0,25.0))
133.34279965424111

knnestimate(data,(99.0,3.0),k=1)
22.935771290035785
knnestimate(data,(99.0,3.0),k=10)
29.727161237156785
knnestimate(data,(99.0,15.0),k=1)
58.151852659938086
knnestimate(data,(99.0,15.0),k=10)
92.413908926458447
```

mo neighbors, mo problems

[numpredict.py]()

Should All Neighbors Count Equally?

- `getdistances()` sorts the neighbors by distances, but those distances could be:

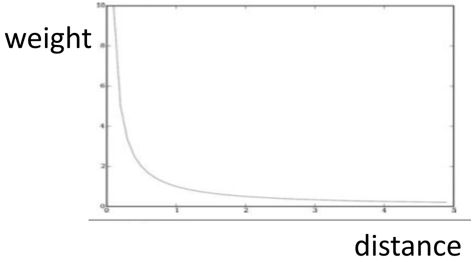
```
1, 2, 5, 11, 12348, 23458, 456599
```

- We'll notice a big change going from k=4 to k=5
- How can we weight the 7 neighbors above accordingly?

Weight Functions

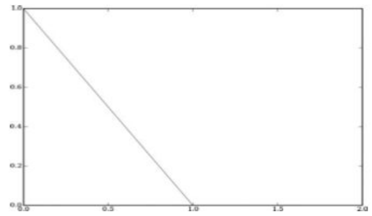
Figures 8-3, 8-4, 8-5 from *PCI*

```
def inverseweight(dist,num=1.0,const=0.1):
 return num/(dist+const)
```



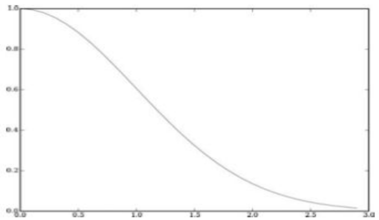
Falls off too quickly

```
def subtractweight(dist,const=1.0):
 if dist>const:
   return 0
 else:
   return const-dist
```



Goes to Zero

```
def gaussian(dist,sigma=5.0):
 return math.e**(-dist**2/(2*sigma**2))
```



Falls Slowly, Doesn't Hit Zero

numpredict.py

Weighted kNN Estimator

```python
def knnestimate(data,vec1,k=5):
 # Get sorted distances
 dlist=getdistances(data,vec1)
 avg=0.0

 # Average of the top k results
 for i in range(k):
   idx=dlist[i][1]
   avg+=data[idx]['result']
 avg=avg/k
 return avg
```

```python
def weightedknn(data,vec1,k=5,weightf=gaussian):
 # Get distances
 dlist=getdistances(data,vec1)
 avg=0.0
 totalweight=0.0

 # Get weighted average
 for i in range(k):
   dist=dlist[i][0]
   idx=dlist[i][1]
   weight=weightf(dist)
   avg+=weight*data[idx]['result']
   totalweight+=weight
 if totalweight==0: return 0
 avg=avg/totalweight
 return avg
```

[numpredict.py]()

Weighted vs. Non-Weighted

```
wineprice(95.0,3.0)
21.111111111111114

knnestimate(data,(95.0,3.0))
21.635620163824875
weightedknn(data,(95.0,3.0))
21.648741297049899

wineprice(95.0,15.0)
84.444444444444457

knnestimate(data,(95.0,15.0))
74.744108153418324
weightedknn(data,(95.0,15.0))
74.949258534489346
```

```
wineprice(95.0,25.0)
137.2222222222222

knnestimate(data,(95.0,25.0))
145.13311902177989
weightedknn(data,(95.0,25.0))
145.21679590393029

knnestimate(data,(95.0,25.0),k=10)
137.90620608492134
weightedknn(data,(95.0,25.0),k=10)
138.85154438288421
```

Web Science:
kNN and Summary
(Part 2 - Validating and Optimizing kNN)

CS 432/532

Old Dominion University

Permission has been granted to use these slides from Michele C. Weigle

Main reference:

Ch 8 from *Programming Collective Intelligence* by Toby Segaran

(abbreviated as PCI)

code available at Ch 8 GitHub repo

Cross-Validation

- Testing all the combinations would be tiresome…

- We cross validate our data to see how our method is performing:
  - divide our 300 bottles into training data and test data (typically something like a (0.95,0.05) split)
  - train the system with our training data, then see if we can correctly predict the results in the test data (where we already know the answer) and record the errors
  - repeat *n* times with different training/test partitions

Cross-Validation Functions

`dividedata(data, test=0.05)` - divides `data` into `trainset` and `testset`, `test` is the portion of the data that should be used for the test set

`testalgorithm(algf, trainset, testset)` - tests the function `algf` using the `trainset` and a query from `testset`, runs for all items in testset

`crossvalidate(algf, data, trials=100, test=0.05)` - divides `data` and runs `testalgorithm()` `trials` number of times and reports the average error (totalerror / trials)

numpredict.py

Preparing for Cross-Validation

```
def knn3(d,v): return knnestimate(d,v,k=3)

def knn1(d,v): return knnestimate(d,v,k=1)

def wknn3(d,v): return weightedknn(d,v,k=3)

def wknn1(d,v): return weightedknn(d,v,k=1)

def wknn5inverse(d,v): return weightedknn(d,v,weightf=inverseweight)
```

Cross-Validating kNN and WkNN

```
crossvalidate(knnestimate,data) # k=5
357.7541491964171 9
```

```
crossvalidate(knn3,data)
374.27654623186737
```

```
crossvalidate(knn1,data)
486.38836851997144
```

```
crossvalidate(weightedknn,data) # k=5
342.80320831062471
```

```
crossvalidate(wknn3,data)
362.6781643445813 2
```

```
crossvalidate(wknn1,data)
524.82845502785574
```

```
crossvalidate(wknn5inverse,data)
342.68187472350417
```

Heterogeneous Data

- Suppose that in addition to rating & age, we collected:
  - bottle size (in ml)
    - 375, 750, 1500, 3000
    - (book code goes to 3000, github code to 1500)
    - [Wine bottle sizes](#) (Wikipedia)
  - the # of aisle where the wine was bought (aisle 2, aisle 9, etc.)

```
def wineset2():
  rows=[]
  for i in range(300):
    rating=random()*50+50
    age=random()*50
    aisle=float(randint(1,20))

bottlesize=[375.0,750.0,1500.0][randint(0,2)]
    price=wineprice(rating,age)
    price*=(bottlesize/750)
    price*=(random()*0.2+0.9)
    rows.append({'input':
        (rating,age,aisle,bottlesize),
        'result':price})
  return rows
```

[numpredict.py](#)

Vintage #2

```
data2 = wineset2()
data2[0]
{'input': (54.165108104770141, 34.539865790286861, 19.0, 1500.0), 'result': 0.0}
data2[1]
{'input': (85.368451290310119, 20.581943831329454, 7.0, 750.0), 'result': 138.67018277159647}
data2[2]
{'input': (70.883447179046527, 17.510910062083763, 8.0, 375.0), 'result': 83.519907955896613}
data2[3]
{'input': (63.236220974521459, 15.66074713248673, 9.0, 1500.0), 'result': 256.55497402767531}
data2[4]
{'input': (51.634428621301851, 6.5094854514893496, 6.0, 1500.0), 'result': 120.00849381080788}

crossvalidate(knn3,data) # from wineset1()
374.27654623186737

crossvalidate(knn3,data2)
1197.0287329391431

crossvalidate(weightedknn,data) # from wineset1()
342.80320831062471

crossvalidate(weightedknn,data2)
1001.3998202008664
```

We have more data -- why are the errors larger?

Differing Data Scales

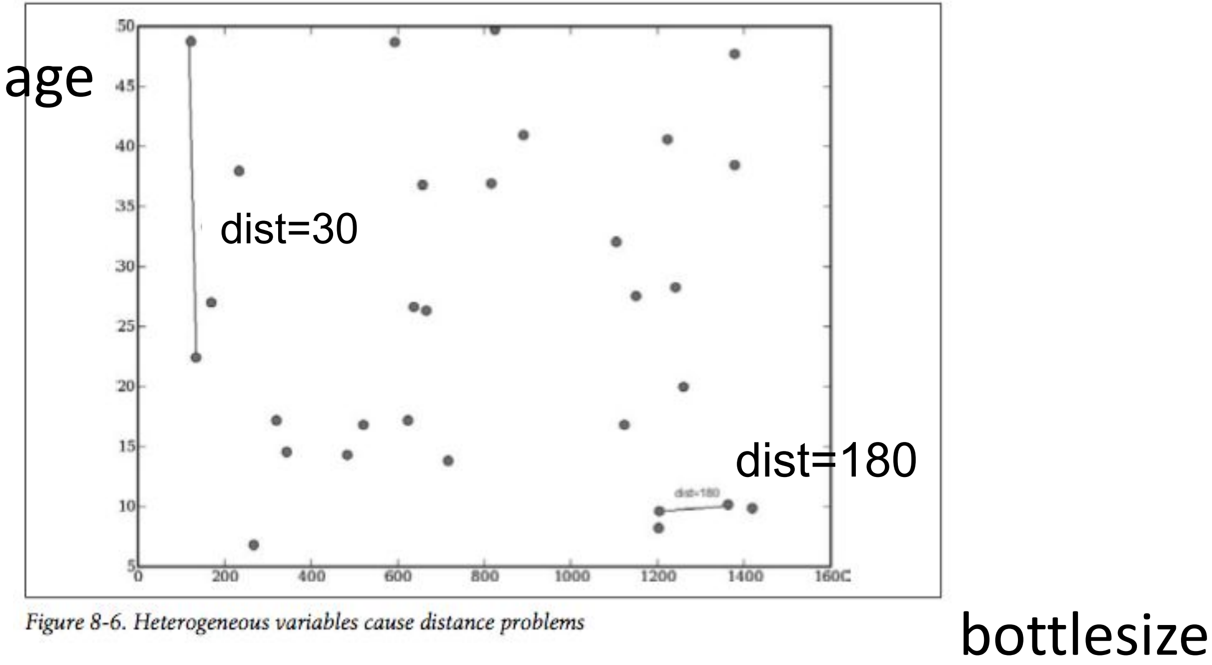


Figure 8-6. Heterogeneous variables cause distance problems

From *PCI*

Rescaling The Data

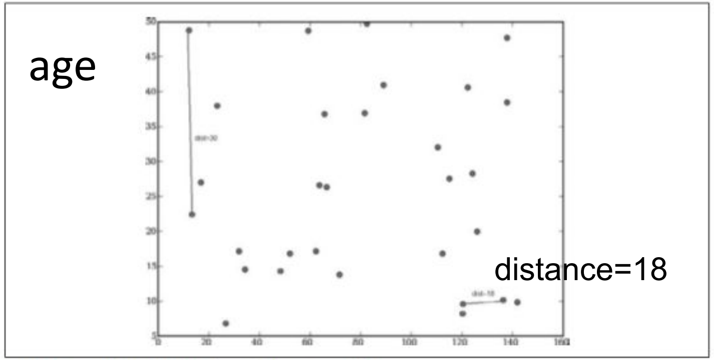


Figure 8-7. Scaling the axes fixes the distance problem

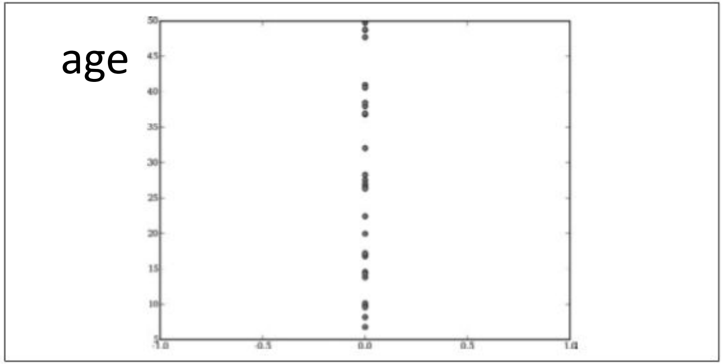


Figure 8-8. Unimportant axes are scaled to 0

Rescale ml by 0.1

Rescale aisle by 0.0

From *PCI*

Cross-Validating Our Scaled Data

```
def rescale(data,scale):
  scaleddata=[]
  for row in data:
     scaled=[scale[i]*row['input'][i] for i in range(len(scale))]
     scaleddata.append({'input':scaled,'result':row['result']})
  return scaleddata
```

```
crossvalidate(knn3,data2)
1197.0287329391431
crossvalidate(weightedknn,data2)
1001.3998202008664

sdata=rescale(data2,[10,10,0,0.5])
crossvalidate(knn3,sdata)
874.34929987724752
crossvalidate(weightedknn,sdata)
1137.6927754808073
```

```
sdata=rescale(data2,[15,10,0,0.1])
crossvalidate(knn3,sdata)
1110.7189445981378
crossvalidate(weightedknn,sdata)
1313.7981751958403

sdata=rescale(data2,[10,15,0,0.6])
crossvalidate(knn3,sdata)
948.16033679019574
crossvalidate(weightedknn,sdata)
1206.6428136396851
```

2nd and 3rd guesses are worse than the initial guess -- but how can we tell if the initial guess is "good"?

[numpredict.py]()

Preparing for Optimization

```
def createcostfunction(algf,data):
  def costf(scale):
    sdata=rescale(data,scale)
    return crossvalidate(algf,sdata,trials=20)
  return costf
```

```
weightdomain=[(0,20)]*4   # book version, code has [(0,10)]*4
```

numpredict.py

Optimizing The Scales

```
import optimization # using the ch 5 version, not the ch 8 version!

costf=createcostfunction(knnestimate,data)

optimization.annealingoptimize(weightdomain,costf,step=2)
[4, 8.0, 2, 4.0]

optimization.annealingoptimize(weightdomain,costf,step=2)
[4, 8, 4, 4]

optimization.annealingoptimize(weightdomain,costf,step=2)
[6, 10, 2, 4.0]
```

the book got [11,18,0,6] --
the last solution is close, but we are hoping to see 0 for aisle…

Optimizing - Genetic Algorithm

```
optimization.geneticoptimize(weightdomain,costf,popsize=5)
1363.57544567
1509.85520291
1614.40150619
1336.71234577
1439.86478765
1255.61496037
1263.86499276
1447.64124381
```

[lots of lines deleted]

```
1138.43826351
1215.48698063
1201.70022455
1421.82902056
1387.99619684
1112.24992339
1135.47820954
[5, 6, 1, 9]
```

the book got [20,18,0,12] on this one

Web Science:
kNN and Summary
(Part 3 - Algorithm Summary)

CS 432/532

Old Dominion University

Permission has been granted to use these slides from Michele C. Weigle

Main reference:

Ch 12 from *[Programming Collective Intelligence]* by Toby Segaran

(abbreviated as PCI)

PCI Algorithms

- Clustering
  - Hierarchical Clustering - Ch 3
  - K-Means Clustering - Ch 3
- Multidimensional Scaling - Ch 3
- Classification
  - Bayesian Classifier - Ch 6
  - Decision Tree Classifier - Ch 7 - *not covered*
  - Neural Networks - Ch 4 - *not covered*
  - Support-Vector Machines (SVM) - Ch 9 - *not covered*
- Prediction
  - k-Nearest Neighbors - Ch 8
- Optimization - Ch 5 - *not covered, but used*

Clustering

- Cluster a set of data into similar groups
- Unsupervised learning
  - no training data required
  - no predictions made
- Hierarchical clustering
- K-Means clustering

Hierarchical Clustering

- Works when there are numerical properties
- Start with all items in their own clusters
  - progressively merge similar clusters
- Result can be shown as a dendrogram

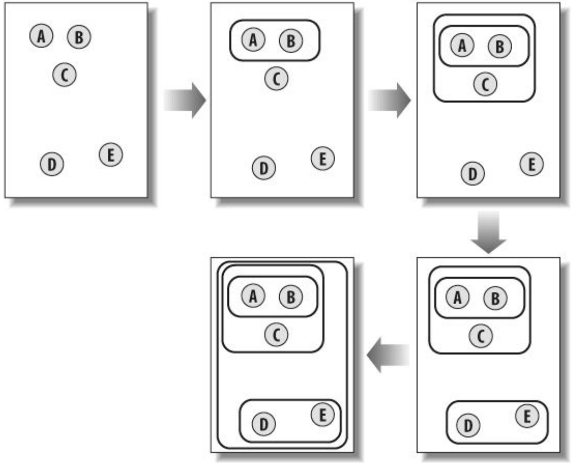
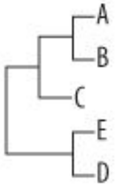


Figure 12-12. Process of hierarchical clustering



From *PCI*

K-Means Clustering

- K-Means separates the data into groups
- Must decide how many groups (*K*) before the process begins
- Division is performed by measuring the distance between items and *centroids* (located at the mean position of the items in the group)

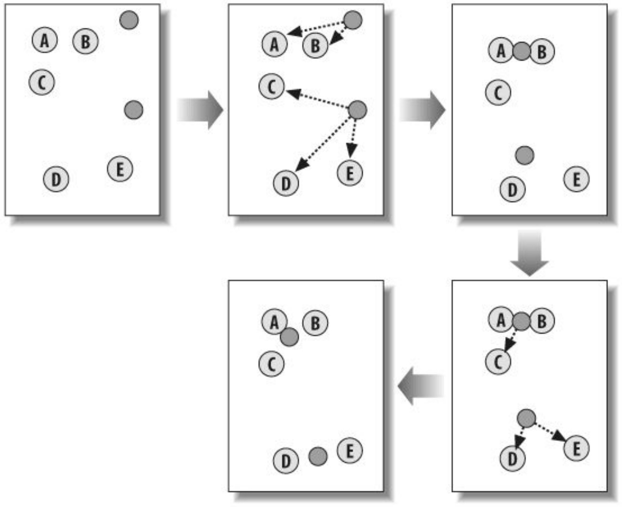
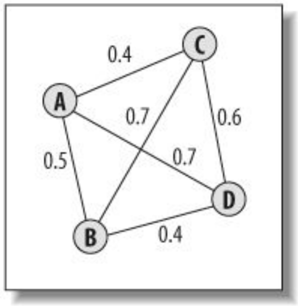


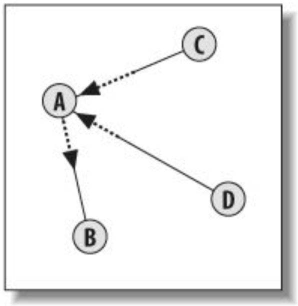
Figure 12-14. Process of K-means clustering

From *PCI*

Multidimensional Scaling (MDS)

- Allows to convert multiple dimensions to 2D for plotting
- Distance in 2D should be similar to distance calculated in all dimensions
- Nodes moved according to combination of forces from other nodes





From *PCI*

Classification

- Classify data into known categories

- Supervised learning
  - training data consists of data and correct classification
  - testing data is what is being classified

- Bayesian Classifier
- Decision Tree Classifier
- Neural Networks
- Support Vector Machines

Bayesian Classifier

- Data is divided up into *features* that can be marked as present or absent
- Uses Bayes' Rule to compute probabilities for each item (based on the probability of the features)
- Trained classifier - list of features and their probabilities
  - no need to store the original data once trained

Strengths/Weaknesses of Bayesian Classifier

Strengths

- Fast
- Supports incremental training
- Easy to interpret the trained model

Weaknesses

- Cannot deal with combinations of features

Decision Tree Classifier

- Once model is trained, classifying is following the tree

- Training is based on dividing up data to maximize information gain

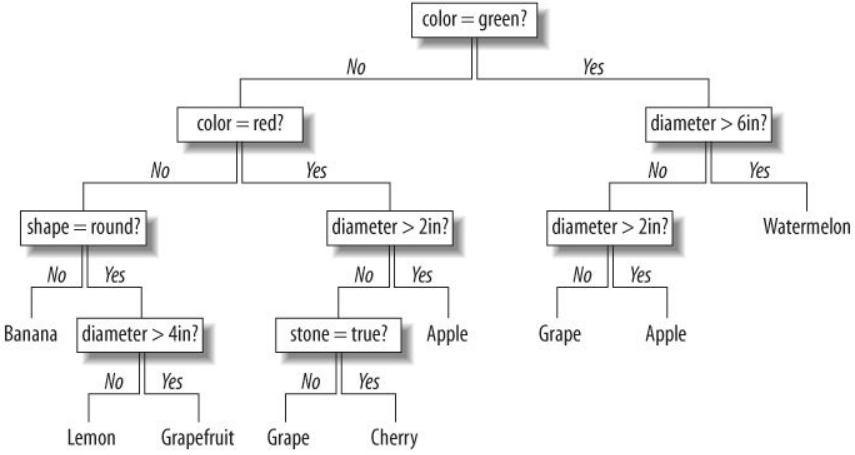


Figure 12-1. Example decision tree

From *PCI*

Strengths/Weaknesses of Decision Trees

Strengths

- Easy to interpret the trained model
- Brings important factors to the top of the tree
- Can deal with combinations of variables

Weaknesses

- Does not support incremental training

Neural Networks

- Layer of input neurons that feed into one or more layers of hidden neurons
- Outputs of one set of neurons are fed to the next layer
- Can learn combinations that are important
- Can be applied to classification and numerical prediction

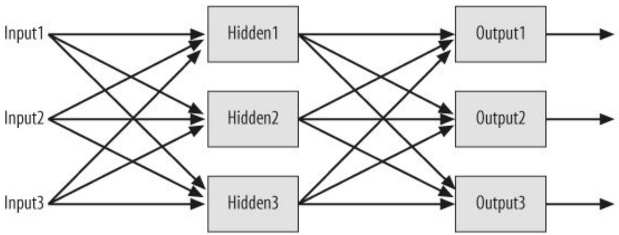


Figure 12-3. Basic neural network structure

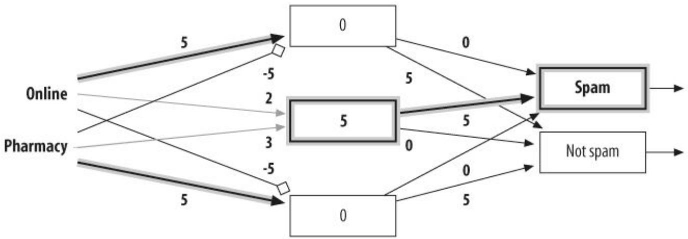


Figure 12-6. Neural network response to "online pharmacy"

From *PCI*

Strengths/Weaknesses of Neural Networks

Strengths

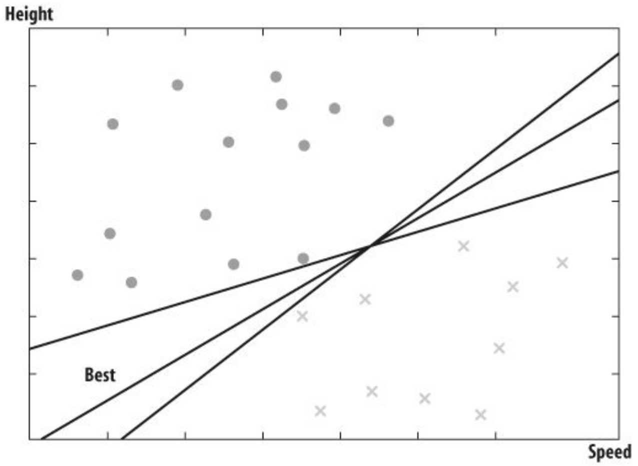
- Can handle complex nonlinear functions
- Can discover dependencies between different inputs
- Allows for incremental training

Weaknesses

- Black box method - impossible to determine how the network came up with the answer
- No definitive rules for choosing the training rate and network size for a given problem

Support Vector Machines (SVM)

- Take datasets with numerical inputs
- Tries to predict categories
- Builds a predictive model by finding the dividing line that separates the data most cleanly
- Support vectors - points closest to the line



From *PCI*

Strengths/Weaknesses of SVMs

Strengths

- Very powerful classifier
- After training, is very fast to classify
- Can work with a mixture of categorical and numerical data

Weaknesses

- Best parameters will be different for every dataset
- Black box technique

k-Nearest Neighbors (kNN)

- Numerical prediction method

- Take the new item and compare it to a set of known items

- Averages the values of the *k* known items most similar for the prediction

From *PCI*

Strengths/Weaknesses of kNN

Strengths

- Can make numerical predictions in complex functions
- Easy to interpret the results
- *Online technique* - new data can be added at any time

Weaknesses

- Requires all training data to be present
- Finding correct scaling factors can be tedious

Optimization

- Attempts to select values that minimize the output of a cost function
- Cost function - returns a higher value for worse solutions and lower value for better solutions
- Optimization functions use the cost function to test solutions and search possible solutions
- Optimization algorithms (covered in PCI)
  - Simulated annealing
  - Genetic algorithms

PCI Algorithms

- Clustering
  - Hierarchical Clustering - Ch 3
  - K-Means Clustering - Ch 3
- Multidimensional Scaling - Ch 3
- Classification
  - Bayesian Classifier - Ch 6
  - Decision Tree Classifier - Ch 7 - *not covered*
  - Neural Networks - Ch 4 - *not covered*
  - Support-Vector Machines (SVM) - Ch 9 - *not covered*
- Prediction
  - k-Nearest Neighbors - Ch 8
- Optimization - Ch 5 - *not covered, but used*

Objectives

- Describe the main idea behind the k-nearest neighbor algorithm.
- Describe the tradeoffs in setting the value of k (not too low, not too high).
- Explain the benefits of weighted kNN.
- Explain the purpose of cross-validation in evaluating prediction algorithms.
- Explain the purpose of the cost function in optimization.