**CS486 - Homework Assignment 3 Chen Lang 20528863**

1. Decision Tree Learning

1.1 Code

**articles.py**

class Attributes(object):

# class variables

# @ self.cnt : total count of attributes (words)

# @ self.name\_list : all attributes

def \_\_init\_\_(self):

file = open("words.txt")

self.cnt = 0

# count

while 1:

line = file.readline()

if not line:

break

self.cnt += 1

file.close()

# read all attributes

file = open("words.txt")

self.name\_list = [None for n in range(self.cnt)]

for i in xrange(0, self.cnt):

self.name\_list[i] = file.readline()

file.close()

def get\_name(self, idx):

return self.name\_list[idx]

def get\_cnt(self):

return self.cnt

# Used as Examples

class Article(object):

# class variables

# @ self.attr\_vals : array of all attributes values

# @ self.classification : class

def \_\_init\_\_(self, attr\_cnt):

# pre-allocate list

self.attr\_vals = [False for n in range(attr\_cnt)]

def set\_attr(self, idx):

if self.attr\_vals[idx] == True:

raise DoubleSetException()

# Set corresponding attributes to True

self.attr\_vals[idx] = True

def get\_attr(self, idx):

return self.attr\_vals[idx]

def set\_class(self, cls):

# 1 = False

# 2 = True

self.classification = cls == 2

def get\_class(self):

return self.classification

class ArticleCollection(object):

# class variables

# @ self.arts : array of all articles

def \_\_init\_\_(self, art\_cnt, data\_file, label\_file, attr\_cnt):

# pre-allocate list

self.arts = [Article(attr\_cnt) for n in range(art\_cnt)]

# read data

file = open(data\_file)

while 1:

line = file.readline()

if not line:

break

# get article index and attribute index

tokens = line.split("\t")

art\_idx = int(tokens[0]) - 1

attr\_idx = int(tokens[1]) - 1

# set attribute

self.arts[art\_idx].set\_attr(attr\_idx)

file.close()

# read lable

file = open(label\_file)

art\_idx = 0

while 1:

line = file.readline()

if not line:

break

# get classification

classification = int(line)

# set attribute

self.arts[art\_idx].set\_class(classification)

# move to next article

art\_idx += 1

file.close()

def get\_art\_cls(self, art\_idx):

return self.arts[art\_idx].get\_class()

def get\_art\_attr(self, art\_idx, attr\_idx):

return self.arts[art\_idx].get\_attr(attr\_idx)

def get\_cnt(self):

return len(self.arts)

**dtl.py**

import math

from articles import Attributes, Article, ArticleCollection

# NOTE: convention for classification

# class 1 -> negative

# class 2 -> positive

class DTNode(object):

# class variables

# @ self.pos : branch of positive value

# @ self.neg : branch of negative value

# @ self.attr\_idx : attribute index (only valid on internal node)

# @ self.ig : information gain

# @ self.cls : classification (only valid on leaf node)

# @ self.depth : the depth of this node

def \_\_init\_\_(self, depth):

self.pos = None

self.neg = None

self.cls = False

self.attr\_idx = -1

self.ig = -1.0

self.depth = depth

class DTL(object):

# class variables

# @ self.zero\_val : zero value threshold

# @ self.attr : attribute collection

# @ self.att\_cnt : count of all attributes

# @ self.art\_col : the collection of all training articles

# @ self.root : root of decision tree

def \_\_init\_\_(self):

# initialize all data

self.zero\_val = 0.0000000000000001

self.attr = Attributes()

self.att\_cnt = self.attr.get\_cnt()

self.art\_col = ArticleCollection(1061, "trainData.txt", "trainLabel.txt", self.att\_cnt)

# split list of articles to 2 lists of articles based on an attribute

# @ idx\_list : list of all examples' index

# @ attr\_idx : attribute index

# RETURN : tup

def split(self, idx\_list, attr\_idx):

pos\_len = 0

neg\_len = 0

# pre-scan

for art\_idx in idx\_list:

if True == self.art\_col.get\_art\_attr(art\_idx, attr\_idx):

pos\_len += 1

else:

neg\_len += 1

# pre-allocate memory

pos\_list = [-1 for n in range(pos\_len)]

neg\_list = [-1 for n in range(neg\_len)]

# scan again to split

pos\_pos = 0

neg\_pos = 0

for art\_idx in idx\_list:

if True == self.art\_col.get\_art\_attr(art\_idx, attr\_idx):

pos\_list[pos\_pos] = art\_idx

pos\_pos += 1

else:

neg\_list[neg\_pos] = art\_idx

neg\_pos += 1

return (pos\_list, neg\_list, pos\_len, neg\_len)

# calculate entropy of a set of examples

# @ idx\_list : list of all examples' index

def entropy(self, idx\_list):

# initialize count

cnt\_pos = 0

cnt\_neg = 0

cnt\_tol = 0

# iterate

for art\_idx in idx\_list:

if True == self.art\_col.get\_art\_cls(art\_idx):

cnt\_pos += 1

else:

cnt\_neg += 1

cnt\_tol += 1

# calculate

p\_pos = float(cnt\_pos) / float(cnt\_tol)

p\_neg = float(cnt\_neg) / float(cnt\_tol)

if p\_pos < self.zero\_val:

# consider possibility of positive articles is 0

epy = 0.0

elif p\_neg < self.zero\_val:

# consider possibility of negative articles is 0

epy = 0.0

else:

# nothing has possibility of 0

epy = float(-1) \* p\_pos \* math.log(p\_pos, 2.0) \

- p\_neg \* math.log(p\_neg, 2.0)

return epy

# calculate information gain based on an attribute

# @ idx\_list : list of all examples' index

# @ attr\_idx : attribute index

def ig(self, idx\_list, attr\_idx):

# split and calculate ig

(pos\_list, neg\_list, pos\_len, neg\_len) = self.split(idx\_list, attr\_idx)

tol\_len = pos\_len + neg\_len

assert(tol\_len == len(idx\_list))

# calculate ig

if 0 == pos\_len or 0 == neg\_len:

# any sub list = 0, IG = 0

ig = 0.0

else:

epy\_base = self.entropy(idx\_list)

# calculate remainder

epy\_pos = self.entropy(pos\_list)

epy\_neg = self.entropy(neg\_list)

nor\_epy\_pos = float(pos\_len) / float(tol\_len) \* float(epy\_pos)

nor\_epy\_neg = float(neg\_len) / float(tol\_len) \* float(epy\_neg)

remainder = nor\_epy\_pos + nor\_epy\_neg

ig = epy\_base - remainder

return ig

# calculate mode classification

# @ idx\_list : list of all examples' index

def mode(self, idx\_list):

pos\_len = 0

neg\_len = 0

for art\_idx in idx\_list:

if True == self.art\_col.get\_art\_cls(art\_idx):

pos\_len += 1

else:

neg\_len += 1

return pos\_len >= neg\_len

# determine is the classification same among all examples

def is\_same\_cls(self, idx\_list):

pos\_len = 0

neg\_len = 0

for art\_idx in idx\_list:

if True == self.art\_col.get\_art\_cls(art\_idx):

pos\_len += 1

else:

neg\_len += 1

return pos\_len == 0 or neg\_len == 0

# choose best attribute based on IG

# @ idx\_list : list of all examples' index

# @ attr\_list : list of all attribute index

def choose\_attr(self, idx\_list, attr\_list):

# determine the attr\_list is empty

if len(attr\_list) == 0:

return -1

# init.

best\_attr\_ig = -1.0 # IG of corresponding attribute (NOTE: larger is better)

best\_attr\_idx = -1 # attribute index

best\_attr\_idx\_in\_list = -1 # index of attrbute index in attr\_list

# calculat best

for i in xrange(0, len(attr\_list)):

attr\_idx = attr\_list[i]

# calculate IG

attr\_ig = self.ig(idx\_list, attr\_idx)

# determine is this attribute is better?

if attr\_ig > best\_attr\_ig:

best\_attr\_ig = attr\_ig

best\_attr\_idx = attr\_idx

best\_attr\_idx\_in\_list = i

# remove that attribute

del attr\_list[best\_attr\_idx\_in\_list]

return best\_attr\_idx, best\_attr\_ig

# DTL recurse function

# @ cur\_depth : current depth of decision tree

# @ idx\_list : list of all current examples' index

# @ attr\_list : list of all current attribute index

# @ default\_cls : default classification

def learn\_recurse(self, max\_depth, cur\_depth, idx\_list, attr\_list, default\_cls):

self.node\_cnt += 1

# print "current node count: ", self.node\_cnt, "/", self.att\_cnt

if cur\_depth == max\_depth:

# reach max\_depth

node = DTNode(cur\_depth)

node.cls = self.mode(idx\_list)

return node

elif len(idx\_list) == 0:

# empty example list

node = DTNode(cur\_depth)

node.cls = default\_cls

return node

elif self.is\_same\_cls(idx\_list):

# all examples have same classification

node = DTNode(cur\_depth)

node.cls = self.mode(idx\_list)

return node

elif len(attr\_list) == 0:

# empty attribute list

node = DTNode(cur\_depth)

node.cls = self.mode(idx\_list)

return node

else:

# duplicate attr\_list

dup\_attr\_list = list(attr\_list)

# calculate best attribute

(best\_attr, best\_ig) = self.choose\_attr(idx\_list, dup\_attr\_list)

# split

(pos\_list, neg\_list, pos\_len, neg\_len) = self.split(idx\_list, best\_attr)

# recurse procedure start here

new\_default\_cls = self.mode(idx\_list)

new\_depth = cur\_depth + 1

# print "current depth: ", cur\_depth

node = DTNode(cur\_depth)

# best\_attr => True

node.pos = self.learn\_recurse(max\_depth, new\_depth, pos\_list, \

dup\_attr\_list, new\_default\_cls)

# best\_attr => False

node.neg = self.learn\_recurse(max\_depth, new\_depth, neg\_list, \

dup\_attr\_list, new\_default\_cls)

# add branch label

node.attr\_idx = best\_attr

node.ig = best\_ig

return node

# perfrom a DTL

# @ max\_depth : maximum depth of decision tree

def learn(self, max\_depth):

# init.

idx\_list = range(0, self.art\_col.get\_cnt())

attr\_list = range(0, self.att\_cnt)

default\_cls = self.mode(idx\_list) # get default cls by mode

self.node\_cnt = 0

# start to learn

self.root = self.learn\_recurse(max\_depth, 0, idx\_list, attr\_list, default\_cls)

# print decision recurse procedure

def print\_tree\_recurse(self, dt\_node):

if dt\_node.pos == None and dt\_node.neg == None:

print "Class",

if dt\_node.cls == False:

print "1"

else:

print "2"

else:

print "Label", dt\_node.attr\_idx + 1, "(", dt\_node.ig, ")",

print "-", self.attr.get\_name(dt\_node.attr\_idx),

# negative branch

for n in xrange(0, dt\_node.depth):

print " ",

print "False:",

self.print\_tree\_recurse(dt\_node.neg)

# positive branch

for n in xrange(0, dt\_node.depth):

print " ",

print "True:",

self.print\_tree\_recurse(dt\_node.pos)

# print decision tree

def print\_tree(self):

self.print\_tree\_recurse(self.root)

# test recurse procedure

# @ test\_art\_col : test article collection

# @ test\_art\_idx : test article index

# @ dt : decision tree node

def test\_recurse(self, test\_art\_col, test\_art\_idx, dt\_node):

assert(dt\_node != None)

if dt\_node.pos == None and dt\_node.neg == None:

# reach a leaf node

return dt\_node.cls

else:

# need to determine which node

art\_attr\_val = test\_art\_col.get\_art\_attr(test\_art\_idx, dt\_node.attr\_idx)

# print "article: ", test\_art\_idx, ", attribute index: ", dt\_node.attr\_idx

# go to corresponding node

if True == art\_attr\_val:

return self.test\_recurse(test\_art\_col, test\_art\_idx, dt\_node.pos)

else:

return self.test\_recurse(test\_art\_col, test\_art\_idx, dt\_node.neg)

# perform a test

# @ art\_cnt : count of all articles

# @ data\_file : test data file name

# @ label\_file : test label file name

def test(self, art\_cnt, data\_file, label\_file):

# read all test data

test\_art\_col = ArticleCollection(art\_cnt, data\_file, label\_file, self.att\_cnt)

# pre-allocate memory for result

result = [False for n in range(test\_art\_col.get\_cnt())]

# init. other variables

pass\_cnt = 0

fail\_cnt = 0

# run test for all

for test\_art\_idx in xrange(0, test\_art\_col.get\_cnt()):

test\_result = self.test\_recurse(test\_art\_col, test\_art\_idx, self.root)

if test\_result == test\_art\_col.get\_art\_cls(test\_art\_idx):

pass\_cnt += 1

else:

fail\_cnt += 1

print "Test result:"

print "Pass / Fail : ", pass\_cnt, "/", fail\_cnt

# calculate accuracy

tol\_cnt = test\_art\_col.get\_cnt()

accuracy = float(pass\_cnt) / float(tol\_cnt) \* 100.0

print "Accuracy: ", accuracy

print "Loading..."

dtl = DTL()

# 1. Calculate the accuracy under each max\_depth

for max\_depth in xrange(0, 26):

print "Max Depth:", max\_depth

print "Learning..."

dtl.learn(max\_depth)

print "Test trainData: "

dtl.test(1061, "trainData.txt", "trainLabel.txt")

print "Test testData: "

dtl.test(707, "testData.txt", "testLabel.txt")

# 2. Output tree when maximum accuracy is reached

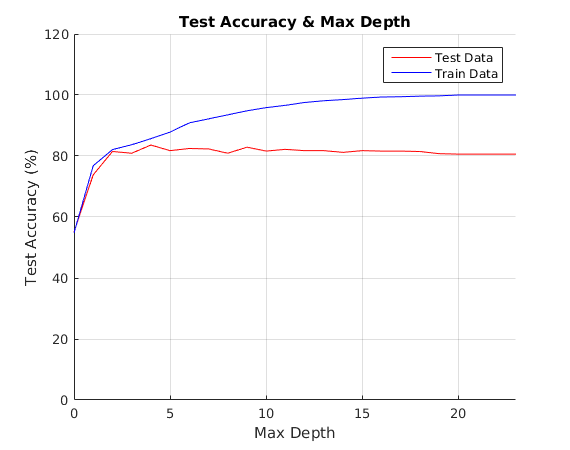
dtl.learn(4)

dtl.print\_tree()

dtl.test(707, "testData.txt", "testLabel.txt")

1.2 Graph

Note: Tree’s depth starts from 0. “0 depth” means only 1 node in the tree.



1.3 Overfitting

Yes, the overfitting occurs. After the maximum depth of 4, overfitting does occur since the accuracy starts to decrease and fluctuate after maximum depth of 4.

1.4 Tree Structure with the Highest Test Accuracy: 83.5926449788 %

Label 485 (0.214992437297) - writes

False: Label 212 (0.109215568062) - god

False: Label 153 (0.0778348840957) - that

False: Label 74 (0.0475424422994) - bible

False: Class 2 - comp.graphics

True: Class 1 - alt.atheism

True: Label 188 (0.097186247382) - wrote

False: Class 2 - comp.graphics

True: Class 1 - alt.atheism

True: Label 184 (0.212290066617) - use

False: Class 1 - alt.atheism

True: Label 1 (1.0) - archive

False: Class 2 - comp.graphics

True: Class 1 - alt.atheism

True: Label 3143 (0.118888826771) - graphics

False: Label 2109 (0.085767875642) - image

False: Label 153 (0.0864686295328) - that

False: Class 1 - alt.atheism

True: Class 1 - alt.atheism

True: Class 2 - comp.graphics

True: Class 2 - comp.graphics

The numerical value in the parenthesis represents the Information Gain (IG) when choose the word after the dash line.

“True” means that word (attribute) is existed for a certain article, and “False” means that word (attribute) is not existed for a certain article.

“Class 1” means the classification of “alt.atheism”, and “Class 2” means the classification of “comp.graphics”.

The integer after the word “Label” represents the word index.

1.5 Brief Discussion

By looking at the tree structure in question 1.4, most word features make sense, but some of them are somewhat vague so these words actually do not mean anything.

Considering the words that makes sense, the articles are classified as “alt.atheism” when they contain the word “god” or “bible”, which is an expected result based on the definition of atheism. Additionally, the articles are classified as “comp.graphics” when they contain the word “graphics” or “image”, which is also an expected result based on the definition of graphics.

However, there are also some vague words that we cannot use to differentiate the class of the article, such as “writes” and “that”. These words are quite common in almost all articles, so they are not expected to appear in the decision tree.

Furthermore, another phenomenon is also expected that some discriminative words are existed in the decision tree to examine the article after the vague words “writes” and “that”. This makes sense since vague words can tell very limited information about the class of an article, so more discriminative words are required for classification.

2. Naïve Bayes Model

2.1 Code

**articles.py**

class Attributes(object):

# class variables

# @ self.cnt : total count of attributes (words)

# @ self.name\_list : all attributes

def \_\_init\_\_(self):

file = open("words.txt")

self.cnt = 0

# count

while 1:

line = file.readline()

if not line:

break

self.cnt += 1

file.close()

# read all attributes

file = open("words.txt")

self.name\_list = [None for n in range(self.cnt)]

for i in xrange(0, self.cnt):

self.name\_list[i] = file.readline()

file.close()

def get\_name(self, idx):

return self.name\_list[idx]

def get\_cnt(self):

return self.cnt

# Used as Examples

class Article(object):

# class variables

# @ self.attr\_vals : array of all attributes values

# @ self.classification : class

def \_\_init\_\_(self, attr\_cnt):

# pre-allocate list

self.attr\_vals = [False for n in range(attr\_cnt)]

def set\_attr(self, idx):

if self.attr\_vals[idx] == True:

raise DoubleSetException()

# Set corresponding attributes to True

self.attr\_vals[idx] = True

def get\_attr(self, idx):

return self.attr\_vals[idx]

def set\_class(self, cls):

# 1 = False

# 2 = True

self.classification = cls == 2

def get\_class(self):

return self.classification

class ArticleCollection(object):

# class variables

# @ self.arts : array of all articles

def \_\_init\_\_(self, art\_cnt, data\_file, label\_file, attr\_cnt):

# pre-allocate list

self.arts = [Article(attr\_cnt) for n in range(art\_cnt)]

# read data

file = open(data\_file)

while 1:

line = file.readline()

if not line:

break

# get article index and attribute index

tokens = line.split("\t")

art\_idx = int(tokens[0]) - 1

attr\_idx = int(tokens[1]) - 1

# set attribute

self.arts[art\_idx].set\_attr(attr\_idx)

file.close()

# read lable

file = open(label\_file)

art\_idx = 0

while 1:

line = file.readline()

if not line:

break

# get classification

classification = int(line)

# set attribute

self.arts[art\_idx].set\_class(classification)

# move to next article

art\_idx += 1

file.close()

def get\_art\_cls(self, art\_idx):

return self.arts[art\_idx].get\_class()

def get\_art\_attr(self, art\_idx, attr\_idx):

return self.arts[art\_idx].get\_attr(attr\_idx)

def get\_cnt(self):

return len(self.arts)

**nbm.py**

import math

from articles import Attributes, Article, ArticleCollection

# NOTE: convention for classification

# class 1 -> negative

# class 2 -> positive

class factor(object):

# class variables

# @ self.att\_idx : attribute index

# @ self.classification : classification

# NOTE: p(att | classification)

def \_\_init\_\_(self):

self.att\_idx = -1

self.classification = False

self.p\_pos = 0.0

# set parameter of this factor

# @ att\_idx : represented attribute index

# @ classification : represented classification

# @ p\_pos : possibility of when attribute is True

def set\_param(self, att\_idx, classification, p\_pos):

self.att\_idx = att\_idx

self.classification = classification

self.p\_pos = p\_pos

# get possibility of when attribute is True

def get\_p\_pos(self):

return self.p\_pos

# get possibility of when attribute is False

def get\_p\_neg(self):

return 1.0 - self.p\_pos

class NBM(object):

# class variables

# @ self.attr : attribute collection

# @ self.att\_cnt : count of all attributes

# @ self.art\_col : the collection of all training articles

# @ self.factors\_neg : the list of all factors of classification 1

# @ self.factors\_pos : the list of all factors of classification 2

# @ self.prior\_neg : prior possibility of classification 1

# @ self.prior\_pos : prior possibility of classification 2

def \_\_init\_\_(self):

# initialize all data

self.attr = Attributes()

self.att\_cnt = self.attr.get\_cnt()

self.art\_col = ArticleCollection(1061, "trainData.txt", "trainLabel.txt", self.att\_cnt)

self.factors\_neg = None

self.factors\_pos = None

self.prior\_neg = -1.0

self.prior\_pos = -1.0

# train the model by using trainData

def learn(self):

# init.

idx\_list = range(0, self.art\_col.get\_cnt())

# split articles by classification

(pos\_list, neg\_list) = self.split\_cls(idx\_list)

# calculate prior possibility

self.prior\_pos = float(len(pos\_list) + 1) / float(len(idx\_list) + 2)

self.prior\_neg = float(len(neg\_list) + 1) / float(len(idx\_list) + 2)

# print "Prior (neg, pos): ", self.prior\_neg, self.prior\_pos

# calculate all factors

self.factors\_neg = [factor() for n in range(self.att\_cnt)]

self.factors\_pos = [factor() for n in range(self.att\_cnt)]

for i in xrange(0, self.att\_cnt):

# calculate and assign

p\_pos = self.cal\_pos\_attr(pos\_list, i)

p\_neg = self.cal\_pos\_attr(neg\_list, i)

self.factors\_neg[i].set\_param(i, False, p\_neg)

self.factors\_pos[i].set\_param(i, True, p\_pos)

# print "P -", i, "(neg, pos): ", p\_neg, p\_pos

# test an article by using trained model

# @ test\_art\_col : collection of all articles

# @ art\_idx : article index

def test\_art(self, test\_art\_col, art\_idx):

# calculate posterior possibility of classification 1

sum\_neg = math.log(self.prior\_neg)

for i in xrange(0, self.att\_cnt):

if test\_art\_col.get\_art\_attr(art\_idx, i) == True:

# attribute == True

sum\_neg += math.log(self.factors\_neg[i].get\_p\_pos())

else:

# attribute == False

sum\_neg += math.log(self.factors\_neg[i].get\_p\_neg())

# calculate posterior possibility of classification 2

sum\_pos = math.log(self.prior\_pos)

for i in xrange(0, self.att\_cnt):

if test\_art\_col.get\_art\_attr(art\_idx, i) == True:

# attribute == True

sum\_pos += math.log(self.factors\_pos[i].get\_p\_pos())

else:

# attribute == False

sum\_pos += math.log(self.factors\_pos[i].get\_p\_neg())

# return result

# print "pos, neg: ", sum\_pos, ",", sum\_neg

if sum\_pos > sum\_neg:

return True

else:

return False

# perform a test

# @ art\_cnt : count of all articles

# @ data\_file : test data file name

# @ label\_file : test label file name

def test(self, art\_cnt, data\_file, label\_file):

# read all test data

test\_art\_col = ArticleCollection(art\_cnt, data\_file, label\_file, self.att\_cnt)

# pre-allocate memory for result

result = [False for n in range(test\_art\_col.get\_cnt())]

# init. other variables

pass\_cnt = 0

fail\_cnt = 0

# run test for all

for test\_art\_idx in xrange(0, test\_art\_col.get\_cnt()):

test\_result = self.test\_art(test\_art\_col, test\_art\_idx)

if test\_result == test\_art\_col.get\_art\_cls(test\_art\_idx):

pass\_cnt += 1

else:

fail\_cnt += 1

print "Test result:"

print "Pass / Fail : ", pass\_cnt, "/", fail\_cnt

# calculate accuracy

tol\_cnt = test\_art\_col.get\_cnt()

accuracy = float(pass\_cnt) / float(tol\_cnt) \* 100.0

print "Accuracy: ", accuracy

# calculate the possibility of a True attribute within a list of articles

# @ idx\_list : list of articles' index

# @ attr\_idx : attribute index

def cal\_pos\_attr(self, idx\_list, attr\_idx):

pos\_cnt = 0

for art\_idx in idx\_list:

if self.art\_col.get\_art\_attr(art\_idx, attr\_idx) == True:

pos\_cnt += 1

return float(pos\_cnt + 1) / float(len(idx\_list) + 2)

# split list of articles into articles of class 2 and class 1

# @ idx\_list : list of articles' index

def split\_cls(self, idx\_list):

pos\_cnt = 0

neg\_cnt = 0

# calculate pos\_cnt & neg\_cnt

for art\_idx in idx\_list:

if self.art\_col.get\_art\_cls(art\_idx) == True:

pos\_cnt += 1

else:

neg\_cnt += 1

# pre-allocate memory

pos\_list = [-1 for n in range(pos\_cnt)]

neg\_list = [-1 for n in range(neg\_cnt)]

# split

pos\_pos = 0

neg\_pos = 0

for art\_idx in idx\_list:

if self.art\_col.get\_art\_cls(art\_idx) == True:

pos\_list[pos\_pos] = art\_idx

pos\_pos += 1

else:

neg\_list[neg\_pos] = art\_idx

neg\_pos += 1

return (pos\_list, neg\_list)

# sort index and value based on value

def sort\_idxval(self, idx, vals):

for i in xrange(0, len(vals) - 1):

for j in xrange(0, len(vals) - 1 - i):

if (vals[j] <= vals[j + 1]):

# swap vals

(vals[j], vals[j + 1]) = (vals[j + 1], vals[j])

# swap index

(idx[j], idx[j + 1]) = (idx[j + 1], idx[j])

# list top 10 discriminative words

def list\_top10(self):

# pre-allocate memory

result\_idx = range(0, self.att\_cnt)

result\_val = [-1.0 for n in range(self.att\_cnt)]

# calculate

for attr\_idx in result\_idx:

result\_val[attr\_idx] = abs(\

math.log(self.factors\_pos[attr\_idx].get\_p\_pos(), 2) -\

math.log(self.factors\_neg[attr\_idx].get\_p\_pos(), 2))

# sort

self.sort\_idxval(result\_idx, result\_val)

# print

for i in xrange(0, 10):

print "No.", i, ":", result\_idx[i] + 1, ",", result\_val[i],

print ",", self.attr.get\_name(result\_idx[i]),

# run

nbm = NBM()

nbm.learn()

nbm.list\_top10()

nbm.test(707, "testData.txt", "testLabel.txt")

nbm.test(1061, "trainData.txt", "trainLabel.txt")

2.2 List of 10 most discriminative word features

|  |  |  |  |
| --- | --- | --- | --- |
| **Order** | **Word Index** | **Measured Value** | **Word** |
| 1 | 3143 | 6.38374874578 | graphics |
| 2 | 3 | 5.73389435561 | atheism |
| 3 | 17 | 5.66678015975 | religion |
| 4 | 426 | 5.55986495583 | moral |
| 5 | 768 | 5.55986495583 | evidence |
| 6 | 571 | 5.55986495583 | keith |
| 7 | 563 | 5.52239025041 | atheists |
| 8 | 212 | 5.45768456103 | god |
| 9 | 74 | 5.40374575392 | bible |
| 10 | 272 | 5.36192557822 | christian |

Note: “Measure Value” is computed by

These words have good features.

Based on the understanding about “atheism” and “graphics”, these words are quite discriminative.

1. “graphics” represents the feature of the articles with class “comp.graphics”;
2. “atheism”, “religion”, “moral” and other words represents the feature of the articles with class “alt.atheism”.

2.3 Accuracy

For test data set, the accuracy is 88.9674681754%

For training data set, the accuracy is 92.8369462771%

2.4 Assumption

This is not a reasonable assumption. When we write a sentence, some words in this sentence are related. So, there exists some word features that are dependent on each other. For example, “bible”, “god”, “christian” are related to each other since the probability of word “bible” will increase if words “god” and “christian” appear in an article.

2.5 Extension

A solution could be adding k edges to a node of word feature, and these edges connect to other nodes of word features in the Naïve Bayes Model Graph.

Edges between nodes of word features represent the dependency of these words. In order to reduce the complexity of the entire graph, only kth top dependency will be considered during training and testing. This can reduce the negative effects brought by the dependency among nodes of word features.

2.6 Better Approach

Naïve Bayes model performs best.

According to the best accuracy achieved by each approach, Naïve Bayes model has the accuracy of 88.97%, but Decision Tree learning only has the accuracy of 83.59%. Also, Decision Tree has the defeat of overfitting based on the graph in the answer 1.2, and this leads to a decline of accuracy. So, Naïve Bayes model performs best under this evaluation criterion.

Furthermore, nodes in decision tree with the highest accuracy still contain vague word features that are used to classify articles, such as “writes”, “that” and “use”. However, the 10 most discriminative word features in Naïve Bayes model are all pretty discriminative. So, Naïve Bayes model has better structure and characteristic under this criterion.

Above all, Naïve Bayes model performs best.