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]
```

```python
    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
```

```python
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

```python
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
```

```python
    # @ 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):
```

```python
                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
```

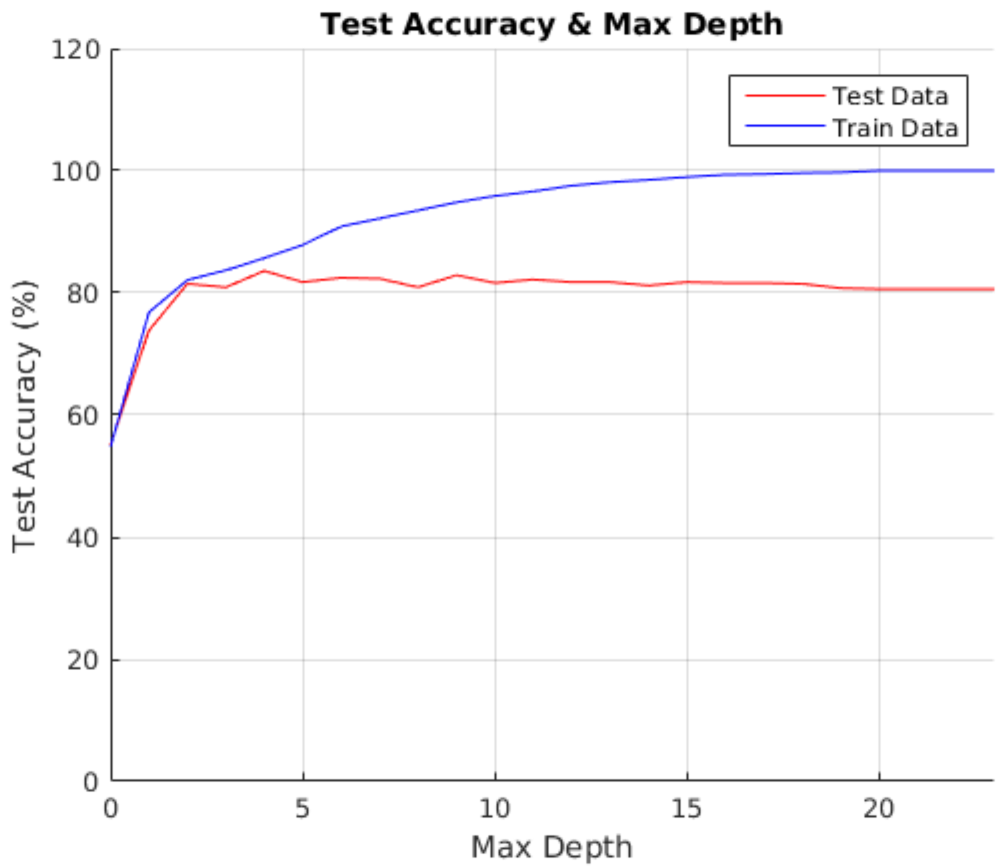
```
        for test_art_idx in xrange(0, test_art_col.get_cnt()):
            test_result = self.test_recurse(test_art_col, test_art_idx,
self.root)
            if test_result == test_art_col.get_art_cls(test_art_idx):
                pass_cnt += 1
            else:
                fail_cnt += 1

        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):
```

```python
        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
```

```python
    # @ 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
```

```python
            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

```python
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
```

```python
    # 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
```

```python
        (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()) -\
            math.log(self.factors_neg[attr_idx].get_p_pos()))
        # 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	4.42487744454	graphics
2	3	3.97443270622	atheism
3	17	3.92791269058	religion
4	426	3.85380471843	moral
5	768	3.85380471843	evidence
6	571	3.85380471843	keith
7	563	3.82782923203	atheists
8	212	3.78297866586	god
9	74	3.74559113379	bible
10	272	3.71660359692	christian

Note: "Measure Value" is computed by

$$| \log_e \Pr(\text{word} \mid \text{label 1}) - \log_e \Pr(\text{word} \mid \text{label 2})|$$

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.