Machine Learning

Program assignment #2

0410001

Hong-Shuo, Chen

1. Abstract

   Implement Kd-Tree and use K-NN classifier to analyze a data set.

2. Problem
   - Construct K-NN classifier with Kd-Tree and use Kd-Tree to find the designated nearest instance
   - Distance metric: Euclidean distance
3. Environment
   - Ubuntu 16.04.3 LTS
4. Using library and language
   - Library: numpy, math, sys, copy
   - Language: Python 3.5.2

5. Results

```
KNN accuracy: 1.0
[ 0.]
[ 1.]
[ 2.]

KNN accuracy: 0.8888888888888888
[   0.  163.   23.  126.  193.]
[   1.  118.   63.   50.   56.]
[   2.    4.    1.  118.   50.]

KNN accuracy: 0.8333333333333334
[   0.  163.   23.  126.  193.  171.  206.   98.  293.   30.]
[   1.  118.   63.   50.   56.    2.    4.  166.  233.  186.]
[   2.    4.    1.  118.   50.   56.   63.  166.  170.  186.]

KNN accuracy: 0.8333333333333334
[   0.  163.   23.  126.  193.  171.  206.   98.  293.   30.   15.  141.
  198.  211.  252.  120.  233.   87.  246.  279.   39.  150.   63.  153.
   85.  225.  263.  231.  111.   29.  286.    2.  281.   60.  118.  296.
  184.  189.  176.  207.  119.  213.   94.  197.  244.  222.  115.  238.
  185.  194.  234.  216.  143.   43.  229.   37.  170.  259.  135.   27.
    7.   25.    1.  284.   51.   66.   57.  243.   75.  290.   77.   93.
   33.   17.  223.  205.   80.  258.  112.  102.  155.   52.   18.  291.
   64.   65.   14.   12.  105.  109.  251.  100.   92.  237.  108.  138.
  161.  157.  181.  280.]
[   1.  118.   63.   50.   56.    2.    4.  166.  233.  186.  170.  163.
  241.  231.  207.  222.  169.  288.   66.    7.  193.   24.  297.   25.
  153.  269.  225.   51.   43.  299.   33.  155.  260.  184.  171.  296.
  141.  189.  285.  290.  292.  234.  206.  181.   85.  111.   67.  161.
  293.  247.   30.  216.  224.  251.  238.  284.   23.   18.  182.  142.
  135.   17.  143.   77.  147.   53.   32.  279.  243.  257.  158.   99.
  112.   26.  295.  262.   39.  140.  281.   87.   19.  198.  286.  289.
   16.  202.   65.  252.  211.   92.   72.  100.   78.  235.  265.  138.
  149.  157.  139.  226.]
[   2.    4.    1.  118.   50.   56.   63.  166.  170.  186.  233.  163.
  241.  231.  222.  193.  207.  169.   66.    7.   25.   24.  288.  153.
  269.  225.  299.   51.   43.  260.  171.  181.  292.  184.  141.  296.
  111.  189.  297.  234.   33.  206.  155.   85.  247.  158.  161.  251.
  293.   53.   30.  216.  142.  224.  182.   87.  238.   18.  135.  143.
   23.   17.   67.  140.   77.  284.  285.  279.  243.  257.   19.  295.
  198.   26.   99.  289.  286.  290.  252.   16.  226.  112.   32.  236.
   72.  211.   65.   36.  202.   92.  262.  100.  265.  102.   78.  138.
  147.  157.  139.  281.]

K = 5, KNN_PCA accuracy: 0.868421052631579
```

6. Algorithm

   I define the tree node as below as a container for my KD-tree. There are four elements in each node, which respectively are parent, left, right, and data. We store our information and the attribute in the data.

   For example,

   root = Tree( ),

   root.data = [data[0], attribute],

   while data[0] is our information and the attribute is the attribute we use to classify the left node and the right node.

```
class Tree(object):
    def __init__(self):
        self.parent = None
        self.left = None
        self.right = None
        self.data = None
```

   There are three important parts, respectively which are creating the KD-tree, finding the nearest node, and KNN. I will explain them in detail below.

   - Create KD-tree

     We need two parameters in this function, which respectively are data and attribute. I build the tree with recursive method.

     There are three conditions.

     When the dataset is empty, we return None.

     While the size of the dataset is 1, we build a tree node which the data is [data[0], attribute] and return it.

     In the last condition, the size of the data is bigger than 1, and we need to split the data with the attribute we get from the parameter. We find the middle of the dataset, and build the tree, which the data is [best, attribute].

     We split the remaining dataset into two parts. The first is smaller than the middle node, and the second part is bigger than the middle node. Finally, we call this

function again, set the left of the root is create_kdtree(data0, attribute) and set the right of the root is create_kdtree(data1, attribute), and return the root.

```
def create_kdtree(data, attribute):
    if data.size == 0:
        return None
    elif data.size == 1:
        root = Tree()
        root.data = [data[0], attribute]
        return root
    else:
        best = choose_middle(data,attribute)
        root = Tree()
        root.data = [best,attribute]
        data0 = data1 = np.array([],dtype = data.dtype)
        for i in range(data.size):
            if data[i] != best:
                if data[i][attribute] < best[attribute]:
                    data0 = np.append(data0,data[i])
                else:
                    data1 = np.append(data1,data[i])
        attribute = get_next_attribute(attribute)
        subtree = create_kdtree(data0, attribute)
        root.left = subtree
        if subtree != None:
            subtree.parent = root
        subtree = create_kdtree(data1, attribute)
        root.right = subtree
        if subtree != None:
            subtree.parent = root
    return root
```

- Find the nearest node

  In this function, we get the query and the root node of the tree. t is our target node, while d_t is the distance between the query and the target node. While x is not None, which means the x is not the parent of the root of the tree, so we need to keep finding. If the distance between the q and the x is less than d_t, we need to update t and d_t. If x is not the leaf node and the boundaryDist(q,x) is less than d_t, we search the subtree,

so we go down the subtree to the leaf node, else we go to our parent and prune the subtree. In the end, it will return the nearest node.

```
def find_the_nearest(q, r):
    t = None
    d_t = 2147483647
    x = descendTree(r,q)
    while x != None:
        if d(q,x) < d_t:
            t = x
            d_t = d(q,x)
        if x.left != None and x.right != None and boundaryDist(q,x) < d_t:
            x = descendTree(x,q)
        else:
            x = parent(x)
    return t
```

- KNN

  In this function, there are four parameters, which are training data, testing data, k, and the list. We could set the k as we want, such as 1, 5, 10, 100...and so on. The list is used to keep tack to the k nearest data.

  Trace is used to know which class appears most. I create the tree and find the first nearest data, and then I delete that data from the dataset, and create the tree and find the nearest data again until we find the k nearest data. Finally, we return we return the predicted result and the list.

```
def KNN(trainning_data, testing_data, k, list):
    tmp_data = trainning_data
    trace = [0,0,0,0,0,0,0,0]
    for i in range(k):
        tree = create_kdtree(tmp_data, '1')
        x = find_the_nearest(testing_data,tree)
        list = np.append(list, x.data[0]['index'])
        trace[check_result(x.data[0]['10'])] = trace[check_result(x.data[0]['10'])] + 1
        for j in range(tmp_data.size):
            if tmp_data[j] == x.data[0]:
                tmp_data = np.delete(tmp_data,j)
                break
    max = 0
    max_id = 0
    for i in range(len(trace)):
        if trace[i] > max:
            max = trace[i]
            max_id = i
    return [max_id, list]
```

In order to complete the three primary function above, I define the following function below:

- get_next_attribute(attribute)

  get the next attribute

  Parameters:

  - attribute: str

  return: attribute: str

```
def get_next_attribute(attribute):
    if attribute == '9':
        return '1'
    else:
        return str(int(attribute) + 1)
```

- choose_middle(data, attribute)

  choose the middle data In the dataset in the order of the attribute

  Parameters:

  - data: numpy.ndarray
  - attribute: str

  return: data[mid]: numpy.ndarray

```
def choose_middle(data, attribute):
    data = np.sort(data, order = attribute)
    if data.size % 2 == 1:
        mid = (data.size + 1)/2
    else:
        mid = (data.size)/2
    mid = int(mid - 1)
    return data[mid]
```

- d(q, x)

  calculate the distance between the query and the given node

Parameters:

- q: numpy.ndarray.dtype
- x: tree node

return: distance: float64

```
def d(q, x):
    sum = 0
    names = np.asarray(x.data[0].dtype.names)
    for i in range(names.size):
        if i != 0 and i != 1 and i != names.size-1:
            sum = sum + (x.data[0][names[i]] - q[names[i]]) * (x.data[0][names[i]] - q
    return sqrt(sum)
```

- descendTree(x, q)

  go to the leaf node

  Parameters:

  - q: numpy.ndarray.dtype
  - x: tree node

  return: leaf node: tree node

```
def descendTree(x, q):
    if x == None:
        return None
    if x.left == None and x.right == None:
        return x
    elif q[x.data[1]] >= x.data[0][x.data[1]]:
        if x.right != None:
            return descendTree(x.right, q)
        else:
            return descendTree(x.left, q)
    else:
        if x.left != None:
            return descendTree(x.left, q)
        else:
            return descendTree(x.right, q)
```

- parent(x)

  return the parent of x and prune the tree

  Parameters:

  - x: tree node

return: parent: tree node

```
def parent(x):
    parent = x.parent
    if parent == None:
        return None
    if parent.left != None and x == parent.left:
        parent.left = None
    elif parent.right != None and x == parent.right:
        parent.right = None
    return parent
```

- boundaryDist(q, x)

  return distance between q and boundary

  Parameters:

  - q: numpy.ndarray.dtype
  - x: tree node

  return: distance: float64

```
def boundaryDist(q, x):
    return q[x.data[1]] - x.data[0][x.data[1]]
```

- check_result(x)
  return the id of the result

  Parameters:

  - x: str

  return: id: int

```
def check_result(x):
    if x == "cp":
        return 0
    elif x == "im":
        return 1
    elif x == "pp":
        return 2
    elif x == "imU":
        return 3
    elif x == "om":
        return 4
    elif x == "omL":
        return 5
    elif x == "imL":
        return 6
    elif x == "imS":
        return 7
```

- main(k)
  do all the things

  Parameters:

  - k: int

  return:

7. Bonus: PCA

   In the function, we do the following things. Firstly, we get the A matrix which we need to calculate the average of the data and subtract the original data from the average. We use the package of the numpy to calculate the eigenvalue(w) and the eigenvector(v) of the transpose of A dot A. I set the threshold to the 0.98, so I need to select the 7 biggest eigenvalue, and the dimension of the data will decrease from 9 to 7. After get the Matrix Q, we need to process our testing data as well. Finally, we put the data which is processed into our function defined above, and it will get the similar result. My accuracy rate is about 0.868, it is a little worse than the accuracy without using PCA which is 0.888. However, due to the dimension is lower, I think this result is predictable. We need less time to process the data, but the trade-off is the accuracy rate.

```
def PCA():
    D = np.loadtxt('tr.csv', delimiter = ',', dtype={'names':('index','0
    T = np.loadtxt('ts.csv', delimiter = ',', dtype={'names':('index','0
    At = np.array([])
    At = np.append(At,D['1'])
    At = np.append(At,D['2'])
```

```
    At = np.append(At,D['9'])
    At = np.reshape(At,(9,300))
    x_avg = np.array([])
    for i in range(9):
        avg = np.average(At[i])
        At[i] = At[i] - avg
        x_avg = np.append(x_avg, avg)
    A = np.transpose(At)
    dot = np.dot(At,A)
    w, v = LA.eig(dot)
    eig = np.sum(w)
    ts = 0.98
    for i in range(w.size - 1):
        for j in range(w.size - i - 1):
            if w[j] < w[j+1]:
                swap = w[j]
                w[j] = w[j+1]
                w[j+1] = swap
                tmp = deepcopy(v[j])
                v[j] = v[j+1]
                v[j+1] = tmp
    sum = 0
    cnt = 0
    Q = np.array([])
    for i in range(w.size):
        if sum/eig > ts:
            break
        else:
            Q = np.append(Q, v[i])
            sum = sum + w[i]
            cnt = cnt + 1
    Q = np.reshape(Q,(cnt,9))
    Q = np.transpose(Q)
```

```
    PCA_data = np.dot(A,Q)
    print('K = 5, KNN_PCA accuracy:', end = " ")
    test = np.array([])
    test = np.append(test,T['1'])
    test = np.append(test,T['2'])
    test = np.append(test,T['3'])
    test = np.append(test,T['4'])
    test = np.append(test,T['5'])
    test = np.append(test,T['6'])
    test = np.append(test,T['7'])
    test = np.append(test,T['8'])
    test = np.append(test,T['9'])
    test = np.reshape(test,(9,36))
    test = np.transpose(test)
    x_avg = np.reshape(x_avg,(1,9))
    test = np.subtract(test,x_avg)
    PCA_test = np.dot(test,Q)
    list = np.array([])
    PCA_T = np.array([],dtype={'names':('index','1','2','3','4','5','6'
    for i in range(36):
        PCA_T = np.append(PCA_T, np.array([(i,PCA_test[i][0],PCA_test[i
    PCA_D = np.array([], dtype=PCA_T.dtype)
    for i in range(300):
        PCA_D = np.append(PCA_D, np.array([(i,PCA_data[i][0],PCA_data[i
    t = f = 1
    for i in range(36):
        if check_result(PCA_T[i]['10']) == KNN_PCA(PCA_D, PCA_T[i], 5,
            t = t + 1
        else:
            f = f + 1
    print(t/(t+f))
    return
```