A Recursive Branch and Bound Algorithm for Feature Selection

Xifeng Tong
Northeast Petroleum University
Daqing, P.R. China
Email: csxftong [AT] 163.com

Shizhong Ma
Northeast Petroleum University
Daqing, P.R. China

Abstract—**Feature selection is a basic problem in pattern recognition application. The branch and bound (BB) algorithm is the only feature selection algorithm that provides global optimal results. All BB algorithms in literature are currently non-recursive algorithms, and their corresponding program code is highly complex. In this study, a recursive BB algorithm, which is easy to program and debug, is designed and implemented. Given that all nodes of a BB tree are built in RAM, the algorithm is unsuitable for high-dimensional features. An improved recursive algorithm that saves each tree node to a hard disk file is also designed and implemented to address the said shortcoming. The number of tree nodes has no limitation, which is suitable for application with high-dimension features. The program code of this algorithm is included in this paper.**

Keywords-feature selection; branch and bound algorithm; recursive algorithm

I. INTRODUCTION

The feature selection procedure in a pattern recognition application involves selecting a subset of *m* features from a given set of *D* features with maximum results derived from a criterion function [1]. Feature selection can decrease feature dimensions with comparatively good classification results, which is a necessity in many pattern recognition applications. Different search strategies indicate that feature selection algorithms can be classified into three major categories [2], including (1) feature selection with global optimal results, such as exhaustive search and branch and bound (BB) algorithms [3][4]; (2) feature selection with a random search strategy; and (3) feature selection with a heuristic search strategy. The global optimal solution in an exhaustive search algorithm is determined by searching all possible feature combinations, which requires high computational time cost. BB algorithms usually require lower computational time cost than the exhaustive search algorithm because they discard several branches under a certain condition. In general, the feature selection with a random search strategy is combined with simulated annealing, genetic algorithm, tabu search, etc. These algorithms can typically obtain suboptimal feature combinations. However, they can decrease the search space that lowers computational time cost. The feature selection with a heuristic search strategy includes sequential forward selection, sequential backward selection, and floating sequential search methods [5][6]. These algorithms involve low computational time cost, but only suboptimal combinations of features can be obtained.

Although the BB algorithm can obtain optimal results, most algorithms in previous literature are non-recursive [1][3][7], and the program codes are excessively long for programming and debugging. This paper presents a recursive BB algorithm for feature selection with a shorter code length and is easy to debug.

II. BRANCH AND BOUND SEARCH

The criterion function *Cfunc*() in a BB algorithm satisfies the monotonicity condition. For two feature subsets S_1 and S_2, if S_1 is a subset of S_2, then $Cfunc(S_1) < Cfunc(S_2)$. Figure 1 shows an example that explains the monotonicity condition: a BB example selects two features from five features. *X* denotes all five features, and the tag of each node denotes the feature to be discarded.

Theoretically, fetching *d* features from *D* features can be conducted in C_d^D ways. The corresponding BB tree has C_d^D leaf nodes. Given the monotonicity of the criterion function in practice, if the criterion function value of a non-leaf node is less than that of another leaf node, all node descendants of the non-leaf node will be discarded. For example, if the criterion function value of node *B* is less than that of node *A* in Figure 1, then all node *B* descendants are discarded from the search space. Therefore, the search space is less than C_d^D.

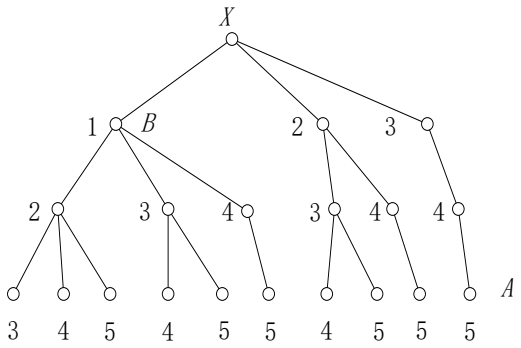

Figure 1. BB search diagram.

III. Proposed Recursive Branch and Bound Algorithm

Figure 1 shows the existing relationships between number of children and sequence numbers for the tree nodes. Assume that *p* is a node and *q* is the father of *p*, then variable *cld_s* of the *p* node is used to denote the count of node *q*'s children. Assume that variable *cld_c* of *p* is the sequence number of *p* in all children nodes of *q*, then the count of node *p*'s children is *cld_s*+1- *cld_c*. We also assume that the level sequence number of the root node is 1. If the level sequence number of a node is *lv*, then the level sequence numbers of children nodes of *p* are all *lv*+1. Assume that *D* features exist and the sequence number of node *p* in *D* features is *m*, then the sequence number in the feature set of each child of node *p* are *m*+1, *m*+2, *m*+3, ..., *m*+*cld_s*+1-*cld_c*. Assume that the search sequence is from top to bottom and from right to left, then a recursive BB algorithm is described as follows. *Stack*[1000] refers to a stack for saving node information, whereas *stack_pt* is the top pointer of the stack in the following algorithm.

```
#define M 7  //number of features in total
#define N 2  //number of features remained
typedef struct TNode
{
        int data;
        double v;
        struct TNode * childs[N+1];
} TNode, * Tree;

int stack[1000], stack_pt;
void CTree(Tree &T, int lv, int fea_c, int cld_s, int cld_c)
{
   int i;
        if(lv>M-N+1)
        {
                T=NULL;
                return;
        }
        T=(Tree)malloc(sizeof(TNode));
        T->data=fea_c;
   for(i=1;i<=cld_s+1-cld_c;i++)
        CTree(T->childs[i-1],lv+1,fea_c+i,cld_s+1-cld_c,i);
        for(i=cld_s+2-cld_c;i<=N+1;i++)
                T->childs[i-1]=NULL;
}
```

```
void OutResult3(Tree T)
{
        if(T==NULL)
                return;
   stack[stack_pt++]=T->data;
        T->v=Cfunc(stack);
        if(T->v <m_bound && max_flag==1)
                return;
        if(T->childs[0]==NULL)
        {
                if(T->v >m_bound)
                {
                        m_bound=T->v;
                        max_flag=1;
                }
                for(int j=1;j<stack_pt;j++)
                        printf("%d ",stack[j]);

                printf("%f  \n",T->v);
        }
        for(int i=N;i>=0;i--)
                OutResult3(T->childs[i]);
        stack_pt--;
}
```

IV. Improved Recursive Branch and Bound Algorithm

In the previous algorithm, all tree nodes are saved in the RAM, making it unsuitable for high-dimensional features. Hence, an improved algorithm where all tree nodes are saved to hard disk files is designed to save RAM space. Each tree node is saved to a hard disk file in the improved algorithm. The feature dimensions has nearly no limitations because the file number in a disk folder also has almost no limitations. The complete source code can be found in the appendix of this paper. *TNodeFile* is used to define a tree node. *data* in *TNodeFile* is used to record the sequence number of the tree node in the feature sets, whereas *v* in *TNodeFile* is used to record the criterion function value. The array *fname[N+1][50]* in *TNodeFile* is used to record each file name that corresponds to each child, *TNodeFile*, respectively. For example, the file name that corresponds to the leftmost child is saved in *fname[0]*. If no file name corresponds to *fname[i]*, then "999999999" is saved to *fname[i]* as a flag. The function *CreateTree()* is used to create a tree recursively. The function *OutResultFile()* is used to search the BB tree recursively and output each group of feature subsets to be deleted. The function *ReadWriteFile()* is used to read data from a disk file with the third parameter set as 0, or write data to a disk file with the third parameter set as 1. Function *Cfunc*() is used to calculate the criterion function value and should be modified according the actual criterion function. A file called "100000000" is created first in the function *main()*; this file only has one child called "100000001". The file called "100000001" is the root node.

V. Implementation of Two Improved Branch and Bound Algorithms in the Proposed Recursive Framework

Yu and Yuan's BB+ algorithm [8] and Chen's IBB algorithm [4] are more efficient than the original BB algorithm. The nodes with only one branch in Yu and Yuan's BB+ algorithm do not require calculation of the criterion function value, unless the node is at the last level. If the criterion function value of a path composed of nodes is less than the bound *B* in Chen's IBB algorithm, then the nodes in the path is saved to a set. All such sets form a set list. If several nodes of a path are equivalent to a set from the set list in the following search, then the path is cut, to enable it to become faster than the BB algorithm. The two algorithms can also be implanted in the proposed recursive framework. Therefore, the proposed recursive framework is generally suitable for the improved BB algorithm. The following algorithm shows the implementation of the BB+ and IBB algorithms in the proposed recursive framework. The function *inSet()* is used to determine whether a node set belongs to the set list, whereas function *addToSet()* is used to add a node set to the set list. The sentence "*if(tf.fname[0][0]!='9' && tf.fname[0][1]=='9')*" is used to judge whether a node only has one branch.

```
void OutResultFile(char filename[50])
{
    int i;
    TNodeFile tf;
    if(filename[0]=='9')
       return;
    ReadWriteFile(filename,tf,0);
    stack[stack_pt++]=tf.data;
    if(tf.fname[0][0]!='9' && tf.fname[0][1]=='9')
tf.v=0;
     else
    {
      if(inSet(stack,stack_pt))
          return;
      tf.v=Cfunc(stack,stack_pt);
      if(tf.v <max_j && max_flag==1)
      {
          addToSet(stack,stack_pt);
          return;
       }
    }
    if(tf.fname[0][0]=='9')
    {
       if(tf.v > max_j)
       {
         max_j=tf.v;
         max_flag=1;
       }
      for(int j=1;j<stack_pt;j++)
         printf("%d ",stack[j]);
      printf("%f  \n",tf.v);
      }
      for(i=N;i>=0;i--)
         OutResultFile(tf.fname[i]);
      stack_pt--;
}
```

Conclusions

A recursive BB algorithm is designed and implemented in this study. The algorithm is easy to program and debug. However, because all tree nodes are built in RAM, the algorithm is unsuitable for high-dimensional features. Hence, a recursive algorithm that saves each tree node to a hard disk file is also designed and implemented to address this shortcoming. No limitations are set on the number of tree nodes, making it suitable for high-dimensional features. Two improved BB algorithms from previous literature are also included in the proposed recursive framework, which suggests that the proposed recursive framework is compatible with improved BB algorithms.

Acknowledgment

This work was supported by Scientific Research Fund of Heilongjiang Provincial Education Department (NO: 12541078).

References

[1] Z. Q. Bian and X. G. Zhang (2000). Pattern recognition. 2nd ed. Beijing: Tsinghua University Publisher.

[2] X. Yao, X. D. Wang, Y. X. Zhang and W. Quan, "Summary of feature selection algorithms", Control and Decision, vol. 27, issue 2, 2012, pp. 161-166.

[3] P. M. Narendra and K. Fukunaga, "A branch and bound algorithm for feature subset selection", IEEE Transactions on Computers, vol. 26, issue 9, 1977, pp. 917-922.

[4] X. W. Chen, "An improved branch and bound algorithm for feature selection", Pattern Recognition Letters, vol. 24, issue 12, 2003, pp. 1925-1933.

[5] K. Fukunaga, "Introduction to Statistical Pattern Recognition", second ed. Academic Press Inc., New York, 1992.

[6] P. Pudil, J. Novovicova, and J. Kittler, "Floating search methods in feature selection", Pattern Recognition Letters, vol. 15, 1994, pp. 1119‑1125.

[7] K. Fukunaga, and P. M. Narendra, "A branch and bound algorithm for computing k-Nearest neighbors", IEEE Transactions on Computer, vol. 24, issue 7, 1975, pp. 750-753.

[8] B. Yu, and B. Yuan, "A more efficient branch and bound algorithm for feature selection", Pattern Recognition, vol. 26, 1993, pp. 883–889.

Appendix

The following is the program code of the recursive BB algorithm, where all tree nodes are saved to hard disk files.

```
#include "stdlib.h"
#include "string.h"
#define M 7  //defining number of features in total
#define N 4  //defining number of feature remained
typedef struct TNodeF
{
        int data;
        double v;
        char fname[N+1][50];
} TNodeFile;
int ct,stack[300],stack_pt;
double m_bound;
```

```
int max_flag;
double Cfunc(int stack[100])
{
	return 1.0;
}
void ReadWriteFile(char filename[100],TNodeFile & tf,int type)
{
	char filenameFull[100];
	strcpy(filenameFull,"e:\\temp_TreeNode\\");
	strcat(filenameFull,filename);
	FILE *fp;
	if(type==0)
	{
		fp=fopen(filenameFull,"rb");
		fread(&tf,sizeof(tf),1,fp);
	}
	else
	{
		fp=fopen(filenameFull,"wb");
		fwrite(&tf,sizeof(tf),1,fp);
	}
	fclose(fp);
}

void CreateTree(char filename[50], int fname_index, int lv, int fea_c,
int cld_s, int cld_c)
{
	double rval;
	int i;
	int k=0;
	TNodeFile tf;
	char filenameNew[50];
	if(lv>M-N+1)
	{
		ReadWriteFile(filename,tf,0);
		strcpy(tf.fname[fname_index],"999999999");
		ReadWriteFile(filename,tf,1);
		return;
	}
	tf.data=fea_c;
	for(i=1;i<=N+1;i++)
		strcpy(tf.fname[i-1],"999999999");
	itoa(ct,filenameNew,10);
	ReadWriteFile(filenameNew,tf,1);
	ct++;
	ReadWriteFile(filename,tf,0);
	strcpy(tf.fname[fname_index],filenameNew);
	ReadWriteFile(filename,tf,1);
	for(i=1;i<=cld_s+1-cld_c;i++)
	CreateTree(filenameNew,i-1,lv+1,fea_c+i,cld_s+1-cld_c,i);
}

void OutResultFile(char filename[50])
{
		int i;
	TNodeFile tf;
	if(filename[0]=='9')
		return;
	ReadWriteFile(filename,tf,0);
	stack[stack_pt++]=tf.data;
	tf.v=Cfunc(stack);
	if(tf.v <m_bound && max_flag==1)
		return;
	if(tf.fname[0][0]=='9')
	{
		if(tf.v > m_bound)
		{
			m_bound=tf.v;
			max_flag=1;
		}
		for(int j=1;j<stack_pt;j++)
			printf("%d ",stack[j]);
		printf("%f  \n",tf.v);
	}
	for(i=N;i>=0;i--)
		OutResultFile(tf.fname[i]);
	stack_pt--;
}
void main()
{
	ct=100000001;
	TNodeFile tf;
	ReadWriteFile("100000000",tf,1);
	CreateTree("100000000",0,1,0,N+1,1);
	stack_pt=0;
	m_bound=0;
	max_flag=0;
	OutResultFile("100000001");
}
```