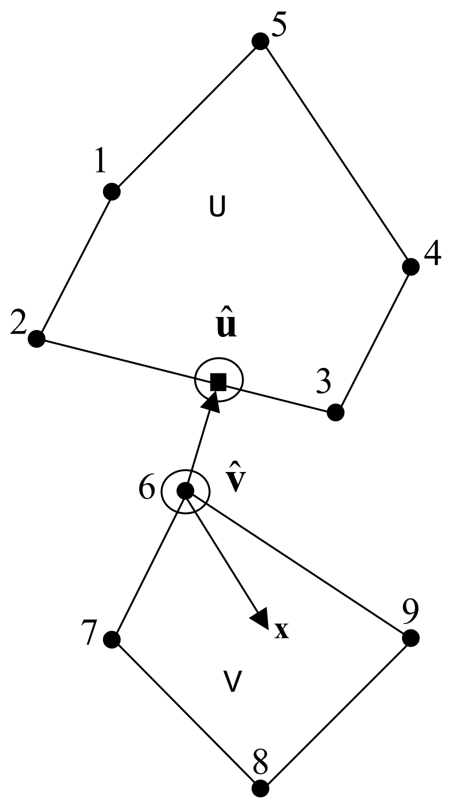
Multiple-Threaded Nearest Point Algorithm
Operating Systems (CSCI-4210) --- A Project Report

Jinbo Bi bij2@rpi.edu
Department of Mathematical Sciences
Rensselaer Polytechnic Institute

Basic Idea

The nearest point algorithm (NPA) is used to find the two closest points in two convex sets as shown in Figure 1. In this example, the convex sets denoted respectively as U and V are the convex hull of the points 1 to 5, and the convex hull of the points 6 to 9. The NPA problem is stated as finding a point from U and a point from V such that the two points are as close as possible. In Figure 1, the $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ are such two points. The core algorithm of NPA can be found in [1, 2]. In this project, the NPA is implemented by breaking it into three threads. Assume that NPA can run on a parallel computing system, this implementation is expected to run faster than does the usual way. The first two threads are each used to iterate through one of the convex sets. The third thread is used for event handling. When the program is running, every time the user types in a character whose function has been pre-specified in a particular set of characters, an event will occur. No GUI has been established for this implementation by this moment. However, the event handling is still carried out by an appropriate interface.



Introduction

As a graduate student in the department of mathematics, I'm interested in solving a family of interesting data mining problems that can be formulated as mathematical optimization problems. Particularly for this project, binary classification problems are investigated. Given a bunch of sample points with some of them belonging to a class (U) and all remaining points belonging to another class (V), the goal of classification is to find the best decision rule to separate the two classes of points. The best decision boundary can be constructed by finding the two closest points in the two convex hulls generated respectively using the two sets of points. Thus, it is actually a NPA problem. Usually solving this kind of problems, (if it has been formulated as a specific sort of optimization problems), needs to use matrix decomposition techniques as done in solving classic support vector machine approaches. It will be computationally very expensive and slow. One way to improve it is to break down the optimization problem into some iterative procedures in order to use fast iterative algorithms. Nearest point algorithm (NPA) is one of the iterative algorithms. Based on previous research, NPA can find a solution very fast. The question of whether or not it is scalable on large data sets is left for further research.

In this project, an effort has been made to implement the NPA in a parallel way with at least two threads running simultaneously although threading is not necessary for the implementation of a NPA. We are not aware of any attempts on speeding up a NPA in terms of its implementation as parallel computing based on the current literature. Hence it will be a cool idea if the algorithm can be efficiently broken down into several threads and each part can be parallel distributed. Although the current implementation may not produce such an efficient decomposition, it is oriented towards this direction. Part of the algorithm computation is not an issue related to OS, such as computing values of a kernel function, and thus is omitted in this report. See [1, 2] for more complete description.

Algorithm and Data Structure

The algorithm contains the following steps:

1. Ask for initial data and parameters as required. This step includes giving the data set used to run NPA, specifying the type of kernel function, providing the kernel parameter and so on. At this moment, only training data is needed.
2. Initialize the starting points **u** and **v** based on the given data set and parameters.
3. Thread 1 starts with the class U, i.e., check the optimality of **u** by iterating through points 1 to 9 (for example if using the file "toytrain.txt" which has been depicted in Figure 1), if the optimality is not satisfied, update **u** and **v** in turn.
   (A reader from the OS area does not have to know a lot about member functions such as "check_optimality", "min_sort" and "update". They are basically not related to OS, but they are the fundamental codes for a NPA algorithm. These functions will be called in the thread function named "Execute".)
4. Thread 2 starts with the class V, i.e., check the optimality of **v** by iterating through points 1 to 9 (as in Figure 1). Then perform similar operations to those in step 3.
5. Thread 3 is created to listen to any events if the user types in characters when both previous threads are running, in other words, when the program is running.
6. All three threads need to cooperate with each other to carry out the task of computing the closest points **u** and **v**. Specifications needed in the event thread include:
   a. If the character 'u' has been entered when the program is listening to events, the program will create a new process to output a log file called "npa1.log" and all intermediate results from Thread 1 are stored in the "npa1.log" file.
   b. If the character 'v' has been typed in when the program is listening to events, the program will create a new process to output a log file called "npa2.log" and all running results from Thread 2 are stored in the "npa2.log" file. Notice that both a. and b. may not be immediately shown up since the "notepad" command may take time to start up.
   c. If the user types in "q", the algorithm may stop running if both Thread 1 and Thread 2 finish running, otherwise abnormally quit from running.
7. If NPA program stops running in a regular way, which means it completes the two NPA threads. The model obtained by the NPA is saved in a file specified by

the user, (the default outcome file name is “toy.mod”). Providing a default name is just for convenience in debugging. In practical running, users can give the outcome model file any names based on the training file name. Besides the model file, the NPA program also produces two other log files, the “npa1.log” for Thread 1, the “npa2.log” for Thread 2.

Race condition can be avoided by defining a class and instanciating it with two objects. The “vector” containers are used to contain data points after reading them in, and store some intermediate matrices. Data points can be in a very high dimensional space. It depends on specific problems. For example, Figure 1 shows a problem in two-dimensional space. In practice, it is common to see thousands of variables for each point. We specify the data format as: the first variable of each sample point is the identification number. The second variable indicates the class which the point belongs to. We have two classes in NPA, so this variable takes values of either +1 or -1. All variables after these two variables are attributes or descriptors of each point. These attributes comprise the various problem dimension.

Specifications and Description

In this project, two VC++ workspaces have been generated: one contains the NPA main functioning program, which perform the NPA computation as described in [1, 2] after correctly reading in data; the other one is a program for testing the models obtained by the first program, which is auxiliary and is not necessarily included in this project. The two workspaces are:

1) npa.dsw
2) npatest.dsw

Open them respectively. Compile and build executable program called npa.exe and npatest.exe respectively. Several warnings may appear when compiling, and can be ignored.

The NPA algorithm and its test program are implemented and debugged using visual C++ 6.0 on a Windows 2000 platform.

1) npa.dsw includes the following codes:

header files:	getin.h
	kernel.h
	npa.h
source files:	getin.cpp
	kernel.cpp
	npa.cpp (NPA class)
	main.cpp

2) npatest.dsw includes the following codes:

header files:	external header files (getin.h and kernel.h)
source files:	getin.cpp
	kernel.cpp
	npatest.cpp

3) the class NPA is the crucial element of the entire program. Its fields and member functions are listed as follows:

```
private:
```

```
            vector<double> u;
            vector<double> v;
            vector< vector<double> > kernel;
            double intercept;
            int M, start_class;
```

member function `initialize --  initializes the uu, uv, and vv values in order to start running NPA.`

member function `comput_value --  computes the function value as specified in [1, 2] Algorithm 1` p_1 and q_1.

member function `value_sort -- saves` p_1 and q_1 `values in the "checker" vector, and then sorts those values and stores the indices in xhat in the ascending order.`

member function `check_optimality -- computes the formula` $\sum p_{i_l} - q_{i_l}$ `and checks if the optimality has been satisfied.`

member function `min_sort -- this function calls the functions comput_value, value_sort, and check_optimality together to finish optimality evaluation and identify a xhat, the point violating optimality conditions.`

member function `compute_norm -- computes the norm of a vector, no matter for` **u**, **v** `or xhat.`

member function `update -- if optimality is not met, it means the current points` **u** `and` **v** `are not optimal yet, so update u or v using the xhat found in min_sort.`

member function `compute_b – computes the intercept term b needed in the classification function because the function does not only need` **u** `and` **v** `but also an offset b.`

member function `Npa_Thread -- This is important for OS class, which repeatedly calls "min_sort" and "update" functions based on the specific situations to perform the NPA algorithm.  This function will be called when a thread is generated.`

member function `NPA(const NPA& copy);`
class operator `NPA& operator = (const NPA& rhs);`

```
      public:
            char* logFile;
```

class constructor `NPA()`

```
            class destructor ~NPA();
```

member function `HANDLE Execute -- this function creates a thread and calls the function Npa_Thread().`

member function `get_u -- interface to return u`
member function `get_v -- interface to return v`
member function `get_intercept -- interface to return b`

The program provides the following functions:

1. Read in data, save as a matrix, (for big data set, it has to do chunking, which will not be included in this project.)
2. Interact with users for information required for running NPA.
3. Main thread sets the mode. If mode =0, the program will only run a demo on a toy data; otherwise if mode =1, it need users to put in corresponding data such as the following:

```
Please input which data file you'd like to handle:
toytrain.txt
Please indicate how many iterations you'd like to run:
1
Please provide the percentage of points made up the nearest
points:
0.4
Please testify what kind of kernel you want to use:
2
Please supply kernel parameter value:
1.0
Please specify which file the result will be stored in:
toy.mod
```

4. Compute kernel matrix, and do whatever to prepare for running NPA.
5. Run NPA by starting two threads.
6. Create the third thread to handle special operations required by users, like "quit" abnormally.
7. Output resulting models obtained by the two threads regardless of which thread gets the solution, and output some information recorded during running, After users type in characters (tackled by the third thread), output the log files.
8. Upon the model generation, run npatest.exe program to test the model on another test data set. The npatest.exe needs several arguments, so the command line would look like:
   If only type in "npatest.exe" without any arguments, the program shows the help information like

```
Usage: npatest model_in training_in test_in outfile kernel_type
```

   An example is the following command line

```
npatest toy.mod toytrain.txt toytest.txt toy.out 2
```

   It tests the model "toy.mod" on the data stored in the "toytest.txt" file, and results will be saved in "toy.out". The program uses polynomial kernel (indicated by the number 2).
   Remember the kernel_type must be consistent to the kernel type used in the npa.exe program where the model is generated by training on data "toytrain.txt". Otherwise it produces a contradiction.

The sample results are illustrated in
toy.mod
npa1.log
npa2.log
which are included in the zip file.

This program is not really targeted towards CS people with basic CS knowledge, and it requires more machine learning knowledge to understand the content. Howver any people can use the program to solve a classification problem if they formulate their problem as a NPA problem and store data in the designated format. It is not hard to know how to use the program, whereas perhaps hard to understand how the algorithm works. The program will have sort of a simple user interface although it won't use GUI. After the program starts over, it will ask users for some information such as what is the seed to generate random numbers when initializing **u** and **v**, or some parameters to perform NPA, etc. After all configurations are settled down, it starts iterations and output resultant variable values for each iteration, in the meanwhile, listening to any action that user may take.

The zip file includes
report.pdf (this file, which serves as a README file)
Header files: getin.h, kernel.h, npa.h, npatest.h, Resource.h, StdAfx.h
Source files: getin.cpp, kernel.cpp, main.cpp, npa.cpp, npatest.cpp, StdAfx.cpp
Project files: npa.dsp, npatest.dsp
Workspaces: npa.dsw, npatest.dsw
Data files: toytrain.txt, toytest.txt
Sample output files: toy.mod, npa1.log, npa2.log

March 25, 2002

[1] J. Bi and K. P. Bennett, *A Generic Nearest Point Algorithm for Solving Support Vector Machines*, Unpublished manuscript (2002).
[2] J. Bi and K. P. Bennett, *A Geometric Approach to Support Vector Regression*, Neurocomputing, Special issue on Support Vector Machines (2003).