Final Exam: k-NN

Author: Gabriel Hofer

Course: CSC-410 Parallel Computing

Instructor: Dr. Karlsson

Due: December 7, 2020

Computer Science and Engineering
South Dakota School of Mines and Technology

Design

The data and task division

One processor is assigned to each record or plant in the data set. The root process (process with rank 0) manages communication, divides the data to different tasks, and prints the results. All of the other processors simply compute the manhattan distance between their assigned record from the training set and each query example.

Description of local and global communication

Let $nprocs$ be the number of processors that are started. First, process 0 opens the file containing the training data set, reads each record, and stores them into the array called $data$. Then, process 0 sends a record (4 attributes, class) to a unique processor. Similarily, process 0 read each record from the file containing the query examples and stores each record into an array called $query$. Process 0 sends the same query to every process with rank > 0. Then every process computes the manhattan distance between their assigned record from the training set and the query example. Each process sends the manhattan distance back to the root process.

The description of the larger combined tasks

The only larger combined task is the root process where the other processes send the results from calculating the manhattan distance.

The description of the mapping

The following function maps the index of the record in the $data$ array to the rank of a process tha the record is sent to:

$$f(data\_index) = rank + 1$$

Program Documentation

How to compile and run

Important specification: the number of processors $nprocs$ that are started must be equal to one more than the number of records in the data-file. In other words, $nprocs = 1 + (wc\ -l\ iris.data)$. For example, iris.data has 150 lines (or records) of data. So, I run the program with 151 processes. This is because the root isn't assigned a record in the data set. Run the program using the Makefile in the root of the submission.

```
$ make small_reg
$ make small_mode
```

This is the format for how command line parameters read by the program:

```
$ mpirun -np  <nprocs>  <data-file>  <query-file>  k  [r|m]
```

Implementation Description

Below are the steps in the k-NN algorithm (pseudo-code from the writeup). Code listings from the program are used to help explain how each step in the algorithm is implemented.

`a. Load the data`

The name of the data-file is read from argv[1]. records are read from the data-file using getline until EOF. Fields in the records are expected to be comma-separated. Therefore, we use sscanf for parsing the fields.

```
if(myproc==0){
  FILE *fp = fopen(argv[1],"r");
  while(getline(&buf, &leng, fp)>0){
    sscanf(buf, "%f %[,] %f %[,] %f %[,] %f %[,] %s", &a, &ch, &b, &ch, &c, &ch, &d, &ch,
        ↪ e);
    data[plant_idx].attr[0]=a;
    data[plant_idx].attr[1]=b;
    data[plant_idx].attr[2]=c;
    data[plant_idx].attr[3]=d;
    strcpy(data[plant_idx].class,e);
    plant_idx++;
  }
  data_size=plant_idx;
  fclose(fp);
}
```

`b. Initialize k to your chosen number of neighbors`

K is passed as a command-line argument.

```
int K=atoi(argv[3]);
```

`c. For each example in the data`
`i. Calculate the distance between the query example and the current example from the data.`

This is the manhattan distance function.

```
float dist(float d[], float q[]){
  return fabs(d[0]-q[0]) + fabs(d[1]-q[1]) + fabs(d[2]-q[2]) + fabs(d[3]-q[3]);
}
```

Each process > 0 calls $dist$ to calculate the manhattan distance.

```
float manhattan=-1;
if(myproc>0) { manhattan=dist(attr, qattr); }
MPI_Barrier(MPI_COMM_WORLD);
```

`ii. Add the distance and the index of the example to an ordered collection.`

Each process sends the distance back to the root process.

```
float arr[1000][2];
if(myproc==0){
  for(int i=1;i<nprocs;i++){ MPI_Irecv(&arr[i-1], 2, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &
      ↪ request[i]); }
} else { MPI_Isend(&man_proc, 2, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &request[0]); }
MPI_Barrier(MPI_COMM_WORLD);
```

d. Sort the ordered collection of distances and indices from smallest to largest (in ascending

qsort is called to sort pairs of data contained in arr. The first element in the pair is the manhattan distance. The second element is the index or label of the record in the data array.

```
int cmp(const void * a, const void * b){ return ((float *)a)[0] > ((float *)b)[0]; }
qsort(arr, data_size, 2*sizeof(float), cmp);
```

e. Pick the first k entries from the sorted collection
f. Get the labels of the selected K entries
g. If regression, return the mean of the k labels

This section of code iterates through the k entries with minimal manhattan distance to the query example and calculates the average for each of the 4 attributes.

```
if(0==strcmp(argv[4],"r")){
  /* regression */
  /* iterate through the closest K vectors and calculate the arith. mean of attributes */
  for(int i=0;i<K;i++){
    rattr[0]+=data[(int)arr[i][1]].attr[0]/(float)K;
    rattr[1]+=data[(int)arr[i][1]].attr[1]/(float)K;
    rattr[2]+=data[(int)arr[i][1]].attr[2]/(float)K;
    rattr[3]+=data[(int)arr[i][1]].attr[3]/(float)K;
  }
  printf("MEAN: %f %f %f %f\n", rattr[0], rattr[1], rattr[2], rattr[3]);
}
```

h. If classification, return the mode of the k labels

This section of code iterates through the k entries with the minimal manhattan distance to the query example. The frequency of each type of plant is counted. A plant with the highest frequency is the mode.

```
else if(0==strcmp(argv[4],"m")){
  /* iterate through the closest K vectors and count the frequency of each class of plant
      ↪  */
  int Iris_virginica=0;
  int Iris_versicolor=0;
  int Iris_setosa=0;
  for(int i=0;i<K;i++){
    if(strcmp(data[(int)arr[i][1]].class,"Iris-virginica")==0)
      Iris_virginica+=1;
    if(strcmp(data[(int)arr[i][1]].class,"Iris-versicolor")==0)
      Iris_versicolor+=1;
```

```
    if(strcmp(data[(int)arr[i][1]].class,"Iris-setosa")==0)
      Iris_setosa+=1;
  }
  /* print the counts for the three plant classes */
  printf("iris-virginica: %d\n", Iris_virginica);
  printf("iris-versicolor: %d\n", Iris_versicolor);
  printf("iris-setosa: %d\n", Iris_setosa);
  /* find the mode by finding the class with the max cnt */
  int mode_idx=-1;
  if(Iris_virginica>=Iris_versicolor && Iris_virginica>=Iris_setosa)
    mode_idx=0;
  else if(Iris_versicolor>=Iris_virginica && Iris_versicolor>=Iris_setosa)
    mode_idx=1;
  else if(Iris_setosa>=Iris_virginica && Iris_setosa>=Iris_versicolor)
    mode_idx=2;
  /* print the mode */
  if(mode_idx==0) printf("MODE: Iris-virginica\n");
  if(mode_idx==1) printf("MODE: Iris-versicolor\n");
  if(mode_idx==2) printf("MODE: Iris-setosa\n");
}
```

Expected format of the training set (input data).

The first command-line argument (argv[1]) passed to the program is the name of the file containing the training set. The iris.data file from
`https://archive.ics.uci.edu/ml/machine-learning-databases/iris/` was used as the datafile for this program. Each line in the file contains a record of information corresponding to some plant's data. Each record in the file has the following format:

```
sepal_length, sepal_width, petal_length, petal_width, class
```

Expected input of the query example

The second command-line argument (argv[2]) passed to the program is the name of the file containing the query examples. The query examples have a similar format to the format of the training set seen above. Each line has a single record with four fields, separated by commas.

```
sepal_length, sepal_width, petal_length, petal_width
```

The only difference between records in the training set and records in the query set is that each record in the query set doesn't have a class field.

An analysis of the performance enhancement obtained from the parallelization.

This program ran slow on my personal machine.