Ionospheric Radar Returns Classification using SGD

High Performance Computing Project Report

Problem Statement: Classifying signals based on high frequency antenna responses and determining line of best fit using parallel computing.

Faculty Guide: Dr. Noor Mahammad

By,

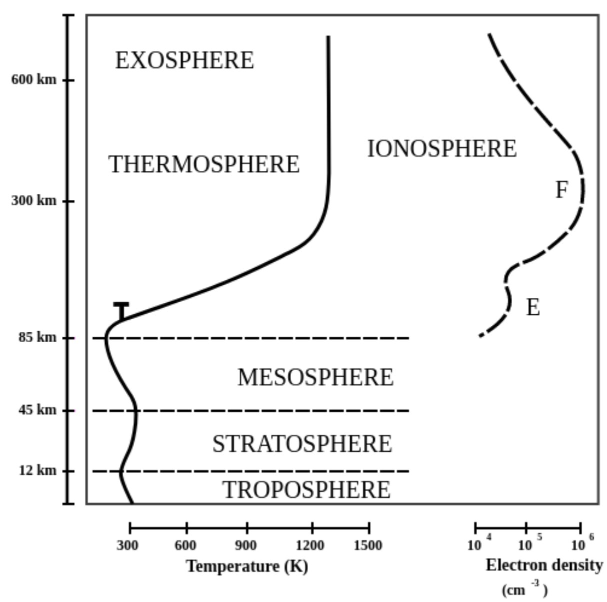
Sujoy Datta
CED18I063

Introduction

The Ionosphere is at the horizon of the atmosphere and outer space. Interestingly, Ionosphere exploration falls into the category of Solar System Exploration. The ionosphere is the ionized part of the Earth's atmosphere from 48 km to 965 km, which includes the thermosphere and parts of the mesosphere and exosphere.

Reasons to classify signals for the Ionosphere-

1. It houses all the charged particles of the Earth's atmosphere.
2. It is the boundary between Earth's atmosphere and space.
3. The orbital drag is felt in this region.
4. The favourite hangout place for our Earth Orbiting satellites.
5. It's influenced by fluctuations in our weather conditions on Earth.
6. Radio and GPS signals are disrupted by radiation in the Ionosphere.
7. Influenced by weather conditions in space.

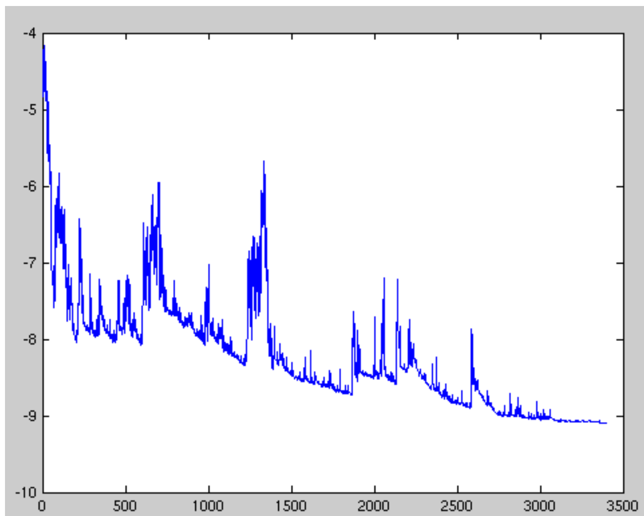


In Ionospheric research, we need to classify the signals as useful(good) or useless(bad) for further analysis. Often in such analysis manual intervention is necessary and it's a painful time-consuming task. The John Hopkins Applied Physics Laboratory has made the data collected from Goose Bay, Labrador radar in the UCI machine learning repository.

This problem from the Geophysics domain can be mapped into a binary classification problem in Machine learning.

Stochastic Gradient Descent Algorithm

Both statistical estimation and machine learning consider the problem of minimizing an objective function that has the form of a sum. Evaluating the sum-gradient may require expensive evaluations of the gradients from all summand functions. Parallelising this operation can improve computational time.



Fluctuations in the total objective function as gradient steps with respect to mini-batches.

Gradient descent is an iterative algorithm that starts from a random point on a function and travels down its slope in steps until it reaches the lowest point of that function. Iteration mechanism is sigmoid.

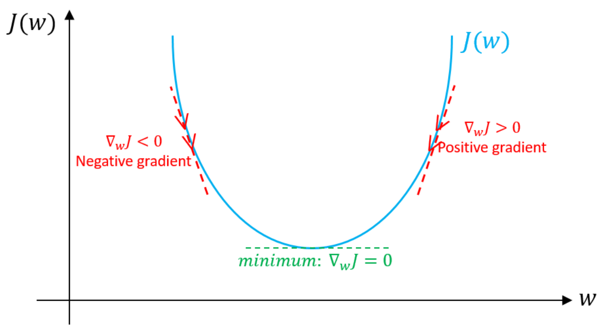
Mathematical Formula-

$$p(\mathbf{x}) = \frac{1}{1 + \exp(-f(\mathbf{x}))}$$

$$f(\mathbf{x}) = b_0 + b_1 x_1 + \cdots + b_r x_r$$

For a dataset with r feature dependencies the line of best fit can be determined using the stochastic gradient descent algorithm. The value of constants [b][0...r] can be calculated using a fixed learning rate and predicted initial values. The sum-minimization problem also arises for empirical risk minimization.

Why use SDG for Ionospheric Regression?



Relevance

1. Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties. So, in SGD, we find out the gradient of the cost function of a single example at each iteration instead of the sum of the gradient of the cost function of all the examples.
2. It can be regarded as a stochastic approximation of gradient descent optimization. In SGD, since only one sample from the dataset is chosen at random for each iteration, the path taken by the algorithm to reach the minima is usually noisier than your typical Gradient Descent algorithm.
3. Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate. But that doesn't matter all that much because the path taken by the algorithm does not matter, as long as we reach the minimum and with significantly shorter training time.

Classification of Radar Returns

This radar data is collected by a system in Goose Bay, Labrador. The system consists of a phased array of 16 High-frequency antennas with a total transmission power of 6.4 kilowatts. The targets were free electrons in the ionosphere.

Received signals were processed using an autocorrelation function whose arguments are the time of a pulse and the pulse number. There were 17 pulse numbers for the Goose Bay system. Instances in this database are described by 2 attributes per pulse number, corresponding to the complex values returned by the function resulting from the complex electromagnetic signal.

The Goose Bay Laboratory has 34 high frequency antennas that detect the probability of free electrons present in the atmosphere. For the ionospheric classification we can use the exploratory data analysis to determine the most influential data features.

1st high freq antenna	2nd high freq antenna	3rd high freq antenna	Signal classification
0.42267	-0.54487	0.18641	g
-0.16626	-0.06288	-0.13738	b
0.60436	-0.2418	0.56045	g
0.25682	1	-0.32382	b
-0.05707	-0.59573	-0.04608	g
0	0	-0.00039	b
-0.04262	-0.81318	-0.13832	g
1	1	0	b
0.45114	-0.72779	0.38895	g
0.16595	0.24086	-0.08208	b
0.30996	-0.89093	0.22995	g
1	-1	1	b
0.68714	-0.64537	0.64727	g
1	0.88428	1	b
1	0.32492	1	g
1	0.23188	0	b

Received signals were processed using an autocorrelation function whose arguments are the time of a pulse and the pulse number. There were 34 high freq antennas for the Goose Bay system.

Algorithm

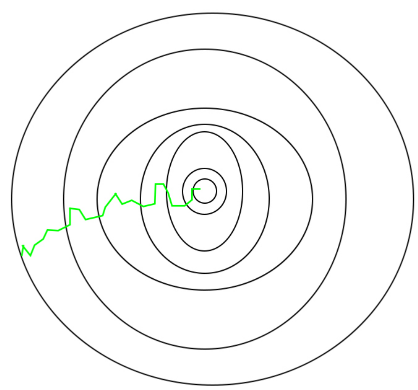
This is the crux of the algorithm. The general idea is to start with a random point (in our parabola example start with a random “x”) and find a way to update this point with each iteration such that we descend the slope.

$$for\ i\ in\ range\,(m):$$

$$\theta_j = \theta_j - \alpha\,(\hat{y}^i - y^i)\,x_j^i$$

The steps of the algorithm are

1. Find the slope of the objective function with respect to each parameter/feature. In other words, compute the gradient of the function.
2. Pick a random initial value for the parameters. (To clarify, in the parabola example, differentiate “y” with respect to “x”. If we had more features like x1, x2 etc., we take the partial derivative of “y” with respect to each of the features.)
3. Update the gradient function by plugging in the parameter values.
4. Calculate the step sizes for each feature as : step size = gradient * learning rate.
5. Calculate the new parameters as : new params = old params -step size
6. Repeat steps 3 to 5 until the gradient is almost 0.



Path taken by Stochastic Gradient Descent

Softwares

1. C/C++, programming language
2. OpenMP, API for shared-memory parallel programming
3. MPI, High performance Message Passing library
4. Cuda C/C++, API for utilizing CUDA-enabled GPU for computation

---

Core Meaning

Stochastic gradient descent algorithms are a modification of gradient descent. In stochastic gradient descent, you calculate the gradient using just a random small part of the observations instead of all of them. In some cases, this approach can reduce computation time.

Online stochastic gradient descent is a variant of stochastic gradient descent in which you estimate the gradient of the cost function for each observation and update the decision variables accordingly. This can help you find the global minimum, especially if the objective function is convex.

Serial Code

Hardware Configuration:
CPU name: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
CPU type: Intel coffeeLake Processor
CPU stepping: 1
Sockets: 1
Cores per socket: 4
Threads per core: 1
Cache Topology
Level: 1
Size: 48 kB
Cache groups: (0) (1) (2) (3)

Level: 2
Size: 1 MB
Cache groups: (0) (1) (2) (3)

Level: 3
Size: 8 MB
Cache groups: (0) (1) (2) (3)

Code

```
#include <bits/stdc++.h>
#include <iostream>
#include <string>
```

```
#include <iomanip>
#include <omp.h>
#include <sstream>
#include <fstream>
using namespace std;

//Variables for obtaining line of best fit
double b0 = 0;
double b1 = 0;
double b2 = 0;
double b3 = 0;

//Swapping function
void swap(double *xp, double *yp)
{
   double temp = *xp;
   *xp = *yp;
   *yp = temp;
}


// A function to implement bubble sort
void bubbleSort(double arr[], int n)
{
   int i, j;
   for (i = 0; i < n-1; i++)

   //Absolute swapping mechanism
   for (j = 0; j < n-i-1; j++)
      if (abs(arr[j]) > abs(arr[j+1]))
         swap(&arr[j], &arr[j+1]);
}


//Training using the obtained data set
void train(double *x1,double *x2,double *x3,double *y)
{
   double error[17550]; // for storing the error values
```

```
    double err;          // for calculating error on each stage
    double alpha = 0.01; // initializing our learning rate
    double e = 2.718281828;

    /*Training Phase*/
    for (int i = 0; i < 17550; i++)
    { //Since there are 350 values in our dataset and we want to run for 50 batches
so total for loop run 17550 times

        //for accessing index after every batch
        int idx = i % 50;

        //making the prediction
        double p = -(b0 + b1 * x1[idx] + b2 * x2[idx] + b3 * x3[idx]);

        //calculating final prediction applying sigmoid
        double pred = 1 / (1 + pow(e, p));

        err = y[idx] - pred; //calculating the error

        //obtaining the line of best fit
        b0 = b0 - alpha * err * pred * (1 - pred) * 1.0;     //updating b0
        b1 = b1 + alpha * err * pred * (1 - pred) * x1[idx]; //updating b1
        b2 = b2 + alpha * err * pred * (1 - pred) * x2[idx]; //updating b2
        b3 = b3 + alpha * err * pred * (1 - pred) * x3[idx]; //updating b3


        //printing values for each training step
        cout << "\tB0= " << b0 << " " << "\t\tB1= " << b1 << " " << "\t\tB2= " << b2
<< "\t\tB3= " << b3 << "\t\tError=" << err << endl;
        error[i]=err;
    }

    //custom sort based on absolute error difference
    bubbleSort(error,17550);
```

```
    cout << "Final Values are: " << "\tB0=" << b0 << " " << "\tB1=" << b1 << " " <<
"\tB2=" << b2 << "\tB3=" << b3 <<"\tError=" << error[0]<<endl;

}

//Testing the trained Stochastic Model
void test(double test1, double test2, double test3)
{
    //make prediction
    double pred = b0 + b1 * test1 + b2 * test2 + b3*test3;
    char ch;

    cout << "The value predicted by the model= " << pred << endl;
    if (pred > 0.5)
    {
        pred = 1;
        ch='g';
    }
    else
    {
        pred = 0;
        ch='b';
    }
    cout << "The class predicted by the model= " << ch<<endl;
}

int main()
{
    //Input dataset arrays
    double x1[351];
    double x2[351];
    double x3[351];
    double y[351];

    //Reading the data file
    FILE* fp = fopen("ionosphere_data.csv", "r");
    char buffer[1024]; int i=0;
    int row = 0; int column = 0;
```

```
while (fgets(buffer,1024, fp))
{
   column = 0;
   row++;
   if (row == 1)
      continue;

   // Splitting the data
   char* value = strtok(buffer, ",");

   while (value)
   {
      // Column 1
      if (column == 0)
      {
         x1[i]=stod(value);
      }
      // Column 2
      if (column == 1)
      {
         x2[i]=stod(value);
      }
      // Column 3
      if (column ==2)
      {
         x3[i]=stod(value);
      }
      // Column 4
      if (column == 3)
      {
          if (value=="g")
          {
             y[i]=1.0;
          }
          else
          {
             y[i]=0.0;
          }
```

```
            i++;
        }
        value = strtok(NULL, ",");
        column++;
      }
}

   //Close the file
   fclose(fp);


   double start,end;
   start=omp_get_wtime();
   //Training Phase
   train(x1, x2,x3, y);
   end=omp_get_wtime();

   //Testing Phase
   double test1=0.5131, test2=-0.00015, test3=0.52099;
   test(test1, test2, test3);

   //Time Taken
   cout<<"Time "<<end-start<<" seconds"<<endl;
   return 0;
}
```

POSIX Terminal Output

```
        B0= 10.4195             B1= -7.06246            B2= 0.082218            B3= -6.05451            Error=-0.0663766
        B0= 10.4203             B1= -7.06314            B2= 0.0823823           B3= -6.05519            Error=-0.3563
        B0= 10.4206             B1= -7.06343            B2= 0.0820905           B3= -6.05519            Error=-0.968919
        B0= 10.4219             B1= -7.06431            B2= 0.0819575           B3= -6.056              Error=-0.812786
        B0= 10.423              B1= -7.06474            B2= 0.0817927           B3= -6.05712            Error=-0.841387
        B0= 10.423              B1= -7.06474            B2= 0.081792            B3= -6.05712            Error=-0.999904
        B0= 10.4233             B1= -7.06503            B2= 0.0815003           B3= -6.05712            Error=-0.968933
        B0= 10.4233             B1= -7.06503            B2= 0.0814998           B3= -6.05712            Error=-0.999903
        B0= 10.4233             B1= -7.06506            B2= 0.0815319           B3= -6.05715            Error=-0.0583938
        B0= 10.4236             B1= -7.06521            B2= 0.0817516           B3= -6.05729            Error=-0.970965
        B0= 10.4239             B1= -7.06553            B2= 0.0817516           B3= -6.05729            Error=-0.966378
        B0= 10.4253             B1= -7.06663            B2= 0.0824306           B3= -6.05835            Error=-0.581602
        B0= 10.4253             B1= -7.06663            B2= 0.0824306           B3= -6.05835            Error=-0.99999
        B0= 10.4268             B1= -7.06775            B2= 0.0827669           B3= -6.05943            Error=-0.615031
        B0= 10.4268             B1= -7.06775            B2= 0.0827669           B3= -6.05943            Error=-0.99997
Final Values are:       B0=10.4268      B1=-7.06775     B2=0.0827669    B3=-6.05943     Error=-0.0583938
The value predicted by the model= 3.64339
The class predicted by the model= g
Time 3.61101 seconds
```

Functional Profiling Output-

Profiling allows us to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster. It can also tell you which functions are being called more or less often than you expected. This may help you spot bugs that had otherwise been unnoticed.

Since the profiler uses information collected during the actual execution of your program, it can be used on programs that are too large or too complex to analyze by reading the source. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature.

Steps to profile-

Step 1. Enable profiling during compilation (use -pg option)
$ gcc -pg -o TestGprof TestGprof.c

Step 2. Execute the binary so that profiling data is generated
$ ./TestGprof

If the profiling is enabled then on executing the program, file gmon.out will be generated.

$ ls
gmon.out TestGprof TestGprof.c

Step 4. Obtain the profiling results in a txt file
$ gprof ./a.out | grep -v std | grep -v static | grep -v cxx > analysis.txt

---

1. **Flat Profile**
   (Excluding the built-in STL functions)

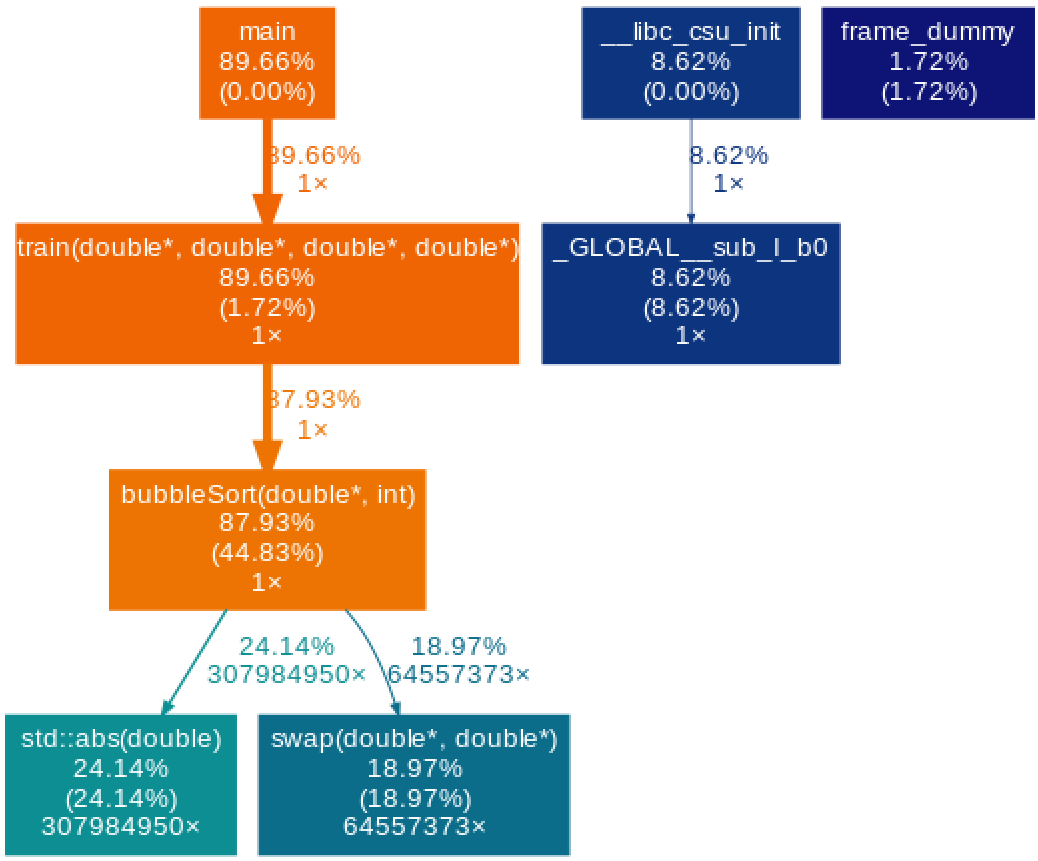
% time	**Cumulative seconds**	**Self Seconds**	**Calls**	**Self ms/call**	**Total ns/call**	**Name**
46.56	0.26	0.26	1	260.72	501.38	bubbleSort(double*, int)
18.80	0.50	0.11	64557373	0.00	0.00	swap(double*, double*)
8.06	0.55	0.05	1	45.12	45.12	_GLOBAL__sub_I_b0
1.79	0.56	0.01	1	10.03	511.40	train(double*, double*, double*, double*)
0.90	0.56	0.01				frame_dummy
0.00	0.56	0.00	1	0.00	0.00	test(double, double, double)

Terminology-

1. % time- the percentage of the total running time of the program used by this function.
2. Cumulative seconds- a running sum of the number of seconds accounted for by this function and those listed above it.
3. self seconds- the number of seconds accounted for by this function alone. This is the major sort for this listing.
4. Calls- the number of times this function was invoked, if this function is profiled, else blank.
5. self ms/call- the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
6. Total ms/call- the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
7. Name- the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the

index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

2. **Functional Graph Diagram-**
   It displays the executed code in a visual diagram where each node corresponds to a function or method and their relations show the code flow.



Line Profiling Output-

Line profiling is a process where we analyze time taken by various parts of our code for every line of the code. It can help us better understand the time/space complexity of our code.

```
sujoydatta@sujoydatta-VirtualBox:~/Desktop/HPC Project$ ls
analysis.txt  gd.cpp  gd.gcda  gd.gcno  gmon.out  gprof2dot.py  ionosphere_data.csv  main.gprof  output.svg  sgd
sujoydatta@sujoydatta-VirtualBox:~/Desktop/HPC Project$ gcov -b -c gd.cpp
File 'gd.cpp'
Lines executed:95.71% of 70
Branches executed:100.00% of 110
Taken at least once:60.00% of 110
Calls executed:91.55% of 71
Creating 'gd.cpp.gcov'
```

```
        -:    0:Source:gd.cpp
        -:    0:Graph:gd.gcno
        -:    0:Data:gd.gcda
        -:    0:Runs:1
        -:    1:#include <bits/stdc++.h>
        -:    2:#include <iostream>
        -:    3:#include <string>
        -:    4:#include <iomanip>
        -:    5:#include <omp.h>
        -:    6:#include <sstream>
        -:    7:#include <fstream>
        -:    8:using namespace std;
        -:    9:
        -:   10://Variables for obtaining line of best fit
        -:   11:double b0 = 0;
        -:   12:double b1 = 0;
        -:   13:double b2 = 0;
        -:   14:double b3 = 0;
        -:   15:
        -:   16://Swapping function
function _Z4swapPdS_ called 64557373 returned 100% blocks executed 100%
 64557373:   17:void swap(double *xp, double *yp)
        -:   18:{
 64557373:   19:    double temp = *xp;
 64557373:   20:    *xp = *yp;
 64557373:   21:    *yp = temp;
 64557373:   22:}
        -:   23:
        -:   24:
        -:   25:// A function to implement bubble sort
function _Z10bubbleSortPdi called 1 returned 100% blocks executed 100%
        1:   26:void bubbleSort(double arr[], int n)
```

```
        -:   27:{
        -:   28:    int i, j;
    17550:   29:    for (i = 0; i < n-1; i++)
branch  0 taken 17549 (fallthrough)
branch  1 taken 1
        -:   30:
        -:   31:    //Absolute swapping mechanism
154010024:   32:    for (j = 0; j < n-i-1; j++)
branch  0 taken 153992475 (fallthrough)
branch  1 taken 17549
153992475:   33:        if (abs(arr[j]) > abs(arr[j+1]))
call    0 returned 153992475
call    1 returned 153992475
branch  2 taken 64557373 (fallthrough)
branch  3 taken 89435102
 64557373:   34:            swap(&arr[j], &arr[j+1]);
call    0 returned 64557373
        1:   35:}
        -:   36:
        -:   37:
        -:   38://Training using the obtained data set
function _Z5trainPdS_S_S_ called 1 returned 100% blocks executed 100%
        1:   39:void train(double *x1,double *x2,double *x3,double *y)
        -:   40:{
        -:   41:    double error[17550]; // for storing the error values
        -:   42:    double err;          // for calculating error on each stage
        1:   43:    double alpha = 0.01; // initializing our learning rate
        1:   44:    double e = 2.718281828;
        -:   45:
        -:   46:    /*Training Phase*/
    17551:   47:    for (int i = 0; i < 17550; i++)
branch  0 taken 17550 (fallthrough)
branch  1 taken 1
        -:   48:    { //Since there are 350 values in our dataset and we want to run for 50 batches so total for loop run 17550 times
        -:   49:
        -:   50:        //for accessing index after every batch
    17550:   51:        int idx = i % 50;
```

```
        -:   52:
        -:   53:        //making the prediction
    17550:   54:        double p = -(b0 + b1 * x1[idx] + b2 * x2[idx] + b3 * x3[idx]);
        -:   55:
        -:   56:        //calculating final prediction applying sigmoid
    17550:   57:        double pred = 1 / (1 + pow(e, p));
        -:   58:
    17550:   59:        err = y[idx] - pred; //calculating the error
        -:   60:
        -:   61:        //obtaining the line of best fit
    17550:   62:        b0 = b0 - alpha * err * pred * (1 - pred) * 1.0;     //updating b0
    17550:   63:        b1 = b1 + alpha * err * pred * (1 - pred) * x1[idx]; //updating b1
    17550:   64:        b2 = b2 + alpha * err * pred * (1 - pred) * x2[idx]; //updating b2
    17550:   65:        b3 = b3 + alpha * err * pred * (1 - pred) * x3[idx]; //updating b3
        -:   66:
        -:   67:
        -:   68:        //printing values for each training step
    17550:   69:        cout << "\tB0= " << b0 << " " << "\t\tB1= " << b1 << " " <<
"\t\tB2= " << b2 << "\t\tB3= " << b3 << "\t\tError=" << err << endl;
call    0 returned 17550
branch  1 taken 17550 (fallthrough)
branch  2 taken 0 (throw)
call    3 returned 17550
branch  4 taken 17550 (fallthrough)
branch  5 taken 0 (throw)
call    6 returned 17550
branch  7 taken 17550 (fallthrough)
branch  8 taken 0 (throw)
call    9 returned 17550
branch 10 taken 17550 (fallthrough)
branch 11 taken 0 (throw)
call   12 returned 17550
branch 13 taken 17550 (fallthrough)
branch 14 taken 0 (throw)
call   15 returned 17550
branch 16 taken 17550 (fallthrough)
branch 17 taken 0 (throw)
call   18 returned 17550
```

```
branch 19 taken 17550 (fallthrough)
branch 20 taken 0 (throw)
call   21 returned 17550
branch 22 taken 17550 (fallthrough)
branch 23 taken 0 (throw)
call   24 returned 17550
branch 25 taken 17550 (fallthrough)
branch 26 taken 0 (throw)
call   27 returned 17550
branch 28 taken 17550 (fallthrough)
branch 29 taken 0 (throw)
call   30 returned 17550
branch 31 taken 17550 (fallthrough)
branch 32 taken 0 (throw)
call   33 returned 17550
branch 34 taken 17550 (fallthrough)
branch 35 taken 0 (throw)
call   36 returned 17550
branch 37 taken 17550 (fallthrough)
branch 38 taken 0 (throw)
    17550:   70:        error[i]=err;
        -:   71:    }
        -:   72:
        -:   73:    //custom sort based on absolute error difference
        1:   74:    bubbleSort(error,17550);
call    0 returned 1
        -:   75:
        -:   76:
        1:   77:    cout << "Final Values are: " << "\tB0=" << b0 << " " << "\tB1=" << b1
<< " " << "\tB2=" << b2 << "\tB3=" << b3 <<"\tError=" << error[0]<<endl;
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
call    3 returned 1
branch  4 taken 1 (fallthrough)
branch  5 taken 0 (throw)
call    6 returned 1
branch  7 taken 1 (fallthrough)
```

```
branch  8 taken 0 (throw)
call    9 returned 1
branch 10 taken 1 (fallthrough)
branch 11 taken 0 (throw)
call   12 returned 1
branch 13 taken 1 (fallthrough)
branch 14 taken 0 (throw)
call   15 returned 1
branch 16 taken 1 (fallthrough)
branch 17 taken 0 (throw)
call   18 returned 1
branch 19 taken 1 (fallthrough)
branch 20 taken 0 (throw)
call   21 returned 1
branch 22 taken 1 (fallthrough)
branch 23 taken 0 (throw)
call   24 returned 1
branch 25 taken 1 (fallthrough)
branch 26 taken 0 (throw)
call   27 returned 1
branch 28 taken 1 (fallthrough)
branch 29 taken 0 (throw)
call   30 returned 1
branch 31 taken 1 (fallthrough)
branch 32 taken 0 (throw)
call   33 returned 1
branch 34 taken 1 (fallthrough)
branch 35 taken 0 (throw)
call   36 returned 1
branch 37 taken 1 (fallthrough)
branch 38 taken 0 (throw)
call   39 returned 1
branch 40 taken 1 (fallthrough)
branch 41 taken 0 (throw)
        -:   78:
        1:   79:}
        -:   80:
        -:   81://Testing the trained Stochastic Model
```

```
function _Z4testddd called 1 returned 100% blocks executed 90%
        1:   82:void test(double test1, double test2, double test3)
        -:   83:{
        -:   84:    //make prediction
        1:   85:    double pred = b0 + b1 * test1 + b2 * test2 + b3*test3;
        -:   86:    char ch;
        -:   87:
        1:   88:    cout << "The value predicted by the model= " << pred << endl;
call    0 returned 1
call    1 returned 1
call    2 returned 1
        1:   89:    if (pred > 0.5)
branch  0 taken 1 (fallthrough)
branch  1 taken 0
        -:   90:    {
        1:   91:        pred = 1;
        1:   92:        ch='g';
        -:   93:    }
        -:   94:    else
        -:   95:    {
    #####:   96:        pred = 0;
    #####:   97:        ch='b';
        -:   98:    }
        1:   99:    cout << "The class predicted by the model= " << ch<<endl;
call    0 returned 1
call    1 returned 1
call    2 returned 1
        1:  100:}
        -:  101:
function main called 1 returned 100% blocks executed 72%
        1:  102:int main()
        -:  103:{
        -:  104:    //Input dataset arrays
        -:  105:    double x1[351];
        -:  106:    double x2[351];
        -:  107:    double x3[351];
        -:  108:    double y[351];
        -:  109:
```

```
        -:  110:    //Reading the data file
        1:  111:    FILE* fp = fopen("ionosphere_data.csv", "r");
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        1:  112:    char buffer[1024]; int i=0;
        1:  113:    int row = 0; int column = 0;
      353:  114:    while (fgets(buffer,1024, fp))
call    0 returned 353
branch  1 taken 353 (fallthrough)
branch  2 taken 0 (throw)
branch  3 taken 352 (fallthrough)
branch  4 taken 1
        -:  115:    {
      352:  116:        column = 0;
      352:  117:        row++;
      352:  118:        if (row == 1)
branch  0 taken 1 (fallthrough)
branch  1 taken 351
        1:  119:            continue;
        -:  120:
        -:  121:        // Splitting the data
      351:  122:        char* value = strtok(buffer, ",");
call    0 returned 351
        -:  123:
     1755:  124:        while (value)
branch  0 taken 1404 (fallthrough)
branch  1 taken 351
        -:  125:        {
        -:  126:            // Column 1
     1404:  127:            if (column == 0)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  128:            {
      351:  129:                x1[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
```

```
branch  3 taken 0 (throw)
call    4 returned 351
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
       -:  130:          }
       -:  131:          // Column 2
    1404:  132:          if (column == 1)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
       -:  133:          {
     351:  134:              x2[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
branch  3 taken 0 (throw)
call    4 returned 351
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
       -:  135:          }
       -:  136:          // Column 3
    1404:  137:          if (column ==2)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
       -:  138:          {
     351:  139:              x3[i]=stod(value);
call    0 returned 351
call    1 returned 351
branch  2 taken 351 (fallthrough)
branch  3 taken 0 (throw)
call    4 returned 351
```

```
branch  5 taken 351 (fallthrough)
branch  6 taken 0 (throw)
call    7 returned 351
call    8 returned 351
call    9 never executed
call   10 never executed
        -:  140:        }
        -:  141:        // Column 4
     1404:  142:        if (column == 3)
branch  0 taken 351 (fallthrough)
branch  1 taken 1053
        -:  143:        {
      351:  144:            if (value=="g")
branch  0 taken 0 (fallthrough)
branch  1 taken 351
        -:  145:            {
    #####:  146:                y[i]=1.0;
        -:  147:            }
        -:  148:            else
        -:  149:            {
      351:  150:                y[i]=0.0;
        -:  151:            }
      351:  152:            i++;
        -:  153:        }
     1404:  154:        value = strtok(NULL, ",");
call    0 returned 1404
     1404:  155:        column++;
        -:  156:    }
        -:  157:}
        -:  158:
        -:  159:    //Close the file
        1:  160:    fclose(fp);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        -:  161:
        -:  162:
        -:  163:    double start,end;
```

```
        1:  164:    start=omp_get_wtime();
call    0 returned 1
        -:  165:    //Training Phase
        1:  166:    train(x1, x2,x3, y);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        1:  167:    end=omp_get_wtime();
call    0 returned 1
        -:  168:
        -:  169:    //Testing Phase
        1:  170:    double test1=0.5131, test2=-0.00015, test3=0.52099;
        1:  171:    test(test1, test2, test3);
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
        -:  172:
        -:  173:    //Time Taken
        1:  174:    cout<<"Time "<<end-start<<" seconds"<<endl;
call    0 returned 1
branch  1 taken 1 (fallthrough)
branch  2 taken 0 (throw)
call    3 returned 1
branch  4 taken 1 (fallthrough)
branch  5 taken 0 (throw)
call    6 returned 1
branch  7 taken 1 (fallthrough)
branch  8 taken 0 (throw)
call    9 returned 1
branch 10 taken 1 (fallthrough)
branch 11 taken 0 (throw)
        1:  175:    return 0;
        -:  176:}
```

Processor Utilization Report-

Inferences-

Conclusion-

Based on the reading and searching for existing serial codes for the Ionospheric Radar Returns Classification using SGD, it can be said that the current parallel execution of discrete time of the problem statement is not found by the student on an open-source platform. This gives a vast amount of room to work with given that there are no existing codes to base the current understanding on how to execute the problem parallelly.

Based on the research document, it would be interesting to analyse the cause of bad returns using serial codes, incoherent scattering, absorption of radar pulses, and interference from the transmitters. I am interested in studying these methods and implementing and analysing a parallel stochastic gradient descent algorithm to probe the way to new research avenues in the study of the Ionosphere.

References-

1. https://www.appliedaicourse.com/ Includes a better overview of the larger problem statement in thought.
2. UCI Machine Learning Repository: Ionosphere Data Set- This radar data was collected by a system in Goose Bay, Labrador. This system consists of a phased array of 16 high-frequency antennas with a total transmitted power on the order of 6.4 kilowatts.
3. https://www.jhuapl.edu/Content/techdigest/pdf/V10-N03/10-03-Sigillito_Class.pdf - Distributed Stochastic Neighborhood Embedding

---