Parallelized Sequential Minimial Optimization for Training of Support Vector Machines

R Mukesh (CED15I002)

IIITDM Kancheepuram

Abstract

Support Vector Machine (SVM) is among is the most popular algorithms in machine learning literature. This papers aims to analyze the performance of the parallelized sequential minimal optimization (SMO) algorithm implemented using MPI (Message Passing Interface).

1 Theory

1.1 Overview of Support Vector Machines

Consider a binary classification dataset,

$$\{(x^{(i)}, y^{(i)}) \mid i = 1, \ldots, m\}$$

where, $x^{(i)} \in \mathbb{R}^n$ is the feature vector representing i$^{\text{th}}$ training instance.
$y^{(i)} \in \{-1, 1\}$ is the class label corresponding to i$^{\text{th}}$ training instance.

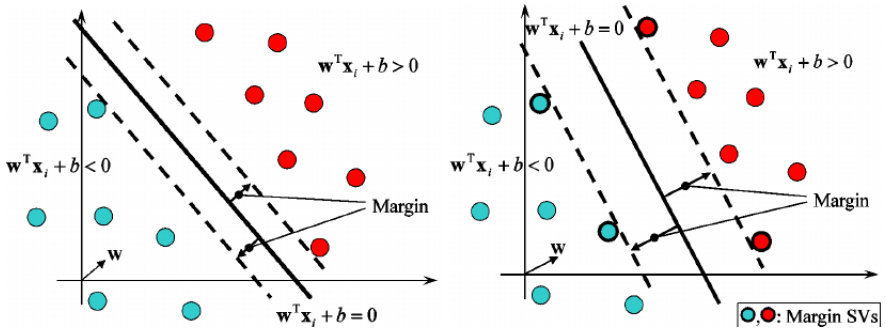

Figure 1: Optimal margin linear seperating hyperplane.

The SVM learning algorithm learns the parameters (w, b) of the optimal margin linear seperating hyperplane (for the data transformed to an higher dimensional feature space using the kernel trick).

Maximizing the margin of the linear seperating hyperplane can be expressed as an constrained optimization problem:

$$\begin{aligned} \underset{w,b}{minimize} \quad & \frac{1}{2}||w||^2 \\ \text{subject to} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \; i = 1, \ldots, m. \end{aligned}$$

Using the langragian, the dual form of the optimization is formulated as a quadratic programming problem:

$$\begin{aligned} \underset{\alpha}{maximize} \quad & \Psi(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} \quad & \alpha_i \geq 0, \; i = 1, \ldots, m. \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \end{aligned}$$

The parameter w that characterizes the optimal margin linear seperating hyperplane is computed:

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}$$

To make the algorithm work for non-linearly seperable datasets as well as less sensitive to outliers, we reformulate the optimization (using $\mathbf{l}_1$ regularization):

$$\begin{aligned} \underset{\alpha}{maximize} \quad & \Psi(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, \ i = 1, \ldots, m. \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \\ & \text{where, } \mathbf{C} \text{ is the regularization parameter.} \end{aligned} \tag{1}$$

The Karush-Kuhn-Tucker(KKT) conditions are necessary and sufficient for the optimal value of a positive-definite quadratic programming problem. The KKT (or convergence) conditions for the quadratic programming problem (1) for $i = 1, \ldots, m$ are:

$$\begin{aligned} \alpha_i = 0 &\iff y^{(i)}(w^T x^{(i)} + b) \geq 1 \\ \alpha_i = C &\iff y^{(i)}(w^T x^{(i)} + b) \leq 1 \\ 0 < \alpha_i < C &\iff y^{(i)}(w^T x^{(i)} + b) = 1 \end{aligned} \tag{2}$$

1.2 Modified SMO Algorithm

Let us consider the partition of training samples into $I_0 = \{i : y = 1, 0 < \alpha_i < C\} \bigcup \{i : y = -1, 0 < \alpha_i < C\}$, $I_1 = \{i : y = 1, \alpha_i = 0\}$, $I_2 = \{i : y = -1, \alpha_i = C\}$, $I_3 = \{i : y = 1, \alpha_i = C\}$ and $I_4 = \{i : y = -1, \alpha_i = 0\}$.

Let $f_i = \sum_{j=1}^{m} \alpha_j y_j \langle X_j, X_i \rangle - y_i$ denotes the error on the i$^{\text{th}}$ training sample.

Parameters b_{up} and b_{low}, and the indices of corresponding training samples, I_{up} and I_{low} are computed:

$$\begin{aligned} b_{up} &= min\{f_i : i \in I_0 \cup I_1 \cup I_2\}, I_{up} = \underset{i \in I_0 \cup I_1 \cup I_2}{\arg\min} f_i \\ b_{low} &= max\{f_i : i \in I_0 \cup I_3 \cup I_4\}, I_{low} = \underset{i \in I_0 \cup I_3 \cup I_4}{\arg\min} f_i \end{aligned}$$

The modified SMO algorithm optimizes the α_is associated with b_{up} and b_{low}, i.e., $\alpha_{I_{low}}$ and $\alpha_{I_{up}}$ at each step.

$$\begin{aligned} \alpha_2^{new} &= \alpha_2^{old} - \frac{y_2(f_1^{old} - f_2^{old})}{\eta} \\ \alpha_1^{new} &= \alpha_1^{old} + s(\alpha_2^{old} - \alpha_2^{new}) \end{aligned} \tag{3}$$

where variables associated with the α_is to be updated are represented using subscript "1" and "2", $\eta = 2\langle X_1, X_2 \rangle - \langle X_1, X_1 \rangle - \langle X_2, X_2 \rangle$, $s = y_1 y_2$. α_1 and α_2 are clipped to $\left[\ 0,\ \mathrm{C}\ \right]$.

After each step, f_i denoting the error on the i$^{\text{th}}$ training sample is updated:

$$f_i^{new} = f_i^{old} + (\alpha_1^{new} - \alpha_1^{old}) y_1 \langle X_1, X_i \rangle + (\alpha_2^{new} - \alpha_2^{old}) y_2 \langle X_2, X_i \rangle. \tag{4}$$

The value of the objective in (1), represented by *Dual* is updated:

$$Dual^{new} = Dual^{new} - \frac{\alpha_1^{new} - \alpha_1^{old}}{y_1}(f_1^{old} - f_2^{old}) + \frac{1}{2}\eta \left(\frac{\alpha_1^{new} - \alpha_1^{old}}{y_1} \right)^2 \tag{5}$$

DualityGap, representing the difference betweeen the primal and dual objective function is calculated:

$$\begin{aligned} DualityGap &= \sum_{i=0}^{m} \alpha_i y_i f_i + \sum_{i=0}^{m} \epsilon_i \\ &\text{where } \epsilon_i = C \max(0, y_i(b - f_i)) \end{aligned} \tag{6}$$

Dual and *DualityGap* are used to check for convergence of the α_is to optimum value of (1).

Modified SMO Algorithm

```
1   Initialize  α_i = 0, f_i = −y_i for i = 1, ..., m
2   Initialize  Dual = 0
3   Calculate  b_up, I_up, b_low, I_low, DualityGap
4   Until  DualityGap ≤ τ|Dual|
5           1)  Optimize  α_{I_up}, α_{I_low}
6           2)  Update  f_i,  for i = 1, ..., m
7           3)  Calculate  b_up, I_up, b_low, I_low, DualityGap  and  Update  Dual.
8   Repeat
```

2 Parallelizing the SMO Algorithm

The entire training data set is equally partitioned into smaller subsets according to the number of processor used. Then, each of the partitioned subsets is distributed into one CPU processor. The attributes associated with each training sample, i.e., α_i, f_i, are maintained locally by the processor to which the sample was distributed. A globally synchronized copy of the attributes b, *Dual* and *DualityGap* is maintained across all processors.

By executing, the program of calculating b_{up}, b_{low}, I_{up} and I_{low} using all processors, each processor could obtain one b_{up} and one b_{low} as well as the associated I_{up} and I_{low} based on its assigned training data patterns. The global b_{up} and global b_{low} are, respectively the minimum value of b_{up} of each processor and maximum value of b_{low} of each processor. By determining the global b_{up} and the global b_{low}, the associated I_{up} and I_{low} could be found out. The corresponding two α_i, are then optimized by using any one CPU processor.

Similarly, each processor computes the *DualityGap* for a subset of training data sample. The value of *DualityGap* on entire training data patterns is the sum of *DualityGap* on all the processors.

The computation time of sequential SMO is dominantly used for updating the f_i array. By executing the program of updating f_i array using all processors, each processor will update a different subset of f_i array based on its training data pattern.

Parallel SMO Algorithm

```
1   Notation: p is the total number of processes used.
2   l^k is the subset of all training data samples indices assigned to processor k.
3   ⋃_{k=1→p} l^k = l denotes the entire set of all training data samples indices.
4
5   f_i^k, α_i^k and i ∈ l^k denote the variables associated with samples in processor k.
6   b_up^k, I_up^k, b_low^k, I_low^k, DualityGap^k denote the variables associated with processor k.
7   f_i^k = Σ_{j=1}^{|l|} α_j y_j ⟨X_j, X_i⟩ − y_i array denotes error on training samples with processor k.
8   b_up^k = min{f_i^k : i ∈ l^k and i ∈ I_0 ∪ I_1 ∪ I_2}, I_up^k is the index of the corresponding sample.
9
10  b_up^k = max{f_i^k : i ∈ l^k and i ∈ I_0 ∪ I_3 ∪ I_4}, I_low^k is the index of the corresponding sample.
11  b_up, I_up, b_low, I_low, and DualityGap denote the variables on entire training data.
12  b_up = min{b_up^k}, b_low = max{b_low^k}, and I_up, I_low denote the indices of corresponding samples.
13  DualityGap = Σ_{k=1}^{p} DualityGap^k
14
15  Initialize  α_i^k = 0,  f_i^k = −y_i,  Dual = 0,  for  i ∈ l^k,  k = 1, ..., p
16  Calculate  b_up^k,  I_up^k,  b_low^k,  I_low^k  and  DualityGap^k.
17
18  Obtain  b_up,  I_up,  b_low,  I_low  and  DualityGap.
19  Until  DualityGap ≤ τ|Dual|
20  1)  Optimize  α_{I_up},  α_{I_low}
21  2)  Update  f_i^k, i ∈ l^k
22  3)  Calculate  b_up^k,  I_up^k,  b_low^k,  I_low^k  and  DualityGap^k.
23  4)  Obtain  b_up,  I_up,  b_low,  I_low  and  DualityGap,  and update  Dual.
24  Repeat
```

3 Inferences

The execution time of the parallelized SMO algorithm is dratically reduced for large number of training samples. However, the accuracy of prediction by the modified SMO algorithm is lower than that of the original SMO algorithm.

Appendix A Downloading and Running Source Code

A.1 Downloading the Source Code

The source code of the parallel implementation of SMO algorithm in C using OpenMPI can be downloaded **here**.

Please visit the github repository **elixir-code/HPC-Lab** for supplementary resources.

A.2 Running the Source Code

1. Extract the zip file containing the source code.

   ```
   $ unzip <path to downloaded source code zip>/svm-openmpi-codes.zip
   ```

2. Open the terminal and change the current directory to the extracted directory that contains source code.

   ```
   $ cd <path where source code zip was extracted>/svm-openmpi-codes
   ```

3. Compile the source code using the OpenMPI's "wrapper" compiler to generate the object file.

   ```
   $ mpicc -o <name of object file> mpisvm.c
   ```

4. Launch the OpenMPI parallel jobs by executing the generated object file.

   ```
   $ mpirun -np <number of processes> <name of generated object file>
   ```

Appendix B Complete Source Code

mpisvm.c

```
/* Implementation of Parallel SMO for training SVM

Reference: Parallel Sequential Minimal Optimization for Training of Support Vector
     Machines
- L.J. Cao, S.S. Keerthi, Chong-Jin Ong, J.Q. Zhang, Uvaraj Periyathamby, Xiu Ju
    Fu, and H.P. Lee

Code: R Mukesh (IIITDM, Kancheepuram)
*/

#include <mpi.h>
#define MASTER 0
#define FROM_MASTER 1

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 1024

#include <errno.h>
extern int errno;

#include <math.h>

#include <assert.h>

/* Read first 'n_data' data points from current file ptr position 'file' into pre-
    allocated arrays 'data' and 'target' */
int readLibsvmDataset(FILE *file, int n_data, int n_features, double data[n_data][
    n_features], int target[n_data]);

/* Read first 'n_data' data points from current file ptr position 'file' into pre-
    allocated arrays 'data' and 'target' */
int readLibsvmDatasetPointers(FILE *file, int n_data, int n_features, double **
    data, int *target);

/* Compute the duality gap for the data subset using alphas, target and fcache
    error values */
double computeDualityGap(int n_data, double alphas[n_data], int target[n_data],
    double fcache[n_data], double b, double C);

/* Compute the index of b_up (I_up) for the data subset using alphas, target and
    fcache error values */
int computebupIndex(int n_data, double alphas[n_data], int target[n_data], double
    fcache[n_data], double C);

/* Compute the index of b_loe (I_low) for the data subset using alphas, target and
     fcache error values */
int computeblowIndex(int n_data, double alphas[n_data], int target[n_data], double
     fcache[n_data], double C);

/* Defines a structure for a index, value pair */
struct RankIndexValuePair
{
  double value;
  int index;
  int rank;
};

/* Define reduction function to find MINIMUM, MAXIMUM among RankIndexValuePair
    structures */
void minlocRankIndexValuePair(void *in, void *inout, int *len, MPI_Datatype *type)
    ;
void maxlocRankIndexValuePair(void *in, void *inout, int *len, MPI_Datatype *type)
    ;

/* Kernel Function to compute dot product of data samples */
double dotProduct(double *data1, double *data2, int n_features);

/* Compute the predictions of the trained SVM model for test data */
```

```
int *computePredictions(int n_features, int train_n_data, double train_data[
    train_n_data][n_features], int train_target[train_n_data], int test_n_data,
    double **test_data, double alphas[train_n_data], double b, double (*
    kernelFunction)(double *, double *, int));

/* Compute the number of data samples with same target and prediction values */
int countCorrectPreds(int n_data, int target[n_data], int predictions[n_data]);

/* Compute the number of data samples with same target and prediction values equal
     to 1 */
int countTruePositivePreds(int n_data, int target[n_data], int predictions[n_data
    ]);

/* Compute the number of data samples with positive target value */
int countPositiveTargets(int n_data, int target[n_data], int predictions[n_data]);

/* Compute the number of data samples with positive prediction value */
int countPositivePreds(int n_data, int target[n_data], int predictions[n_data]);


int main(int argc, char **argv)
{
  int i, j;

  int n_features = 123;

  const char filename[] = "datasets/a1a";
  int n_data = 1605;

  const char test_filename[] = "datasets/a1a.t";
  int test_n_data = 30956;

  // Parameters in training the SVM model
  double C = 1.0;
  double tol = 0.001;
  double eps = 0.001;
  double (*kernelFunction)(double *, double *, int) = dotProduct;

  int world_rank, world_size;
  int n_data_per_worker, n_data_master;

  int test_n_data_per_worker, test_n_data_master;

  MPI_Status status;

  double b, dual;
  b = 0;
  dual = 0;

  double local_duality_gap, global_duality_gap;

  // Define Iup, bup, Ilow, blow for local training sample
  int local_I_up, local_I_low;
  double local_b_up, local_b_low;

  struct RankIndexValuePair local_rank_b_up, local_rank_b_low, global_rank_b_up,
    global_rank_b_low;

  // Variables associated with pair of alpha to update
  double alpha1, alpha2, f1, f2;
  double data1[n_features], data2[n_features];
  int y1, y2;

  int num_changed = 1;
  double updated_alpha1, updated_alpha2;

  // local and global variables associated with validation
  int local_n_correct_preds, global_n_correct_preds;
  int local_n_true_pos_preds, global_n_true_pos_preds;
  int local_n_pos_targets, global_n_pos_targets;
  int local_n_pos_preds, global_n_pos_preds;

  MPI_Init(&argc, &argv);

  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

```
/* Create the new data type to pass b_up, i_up and processor rank */
MPI_Datatype MPI_DOUBLE_INT_INT;
MPI_Datatype datatypes[] = {MPI_DOUBLE, MPI_INT, MPI_INT};
MPI_Aint offsets[] = {offsetof(struct RankIndexValuePair, value), offsetof(
  struct RankIndexValuePair, index), offsetof(struct RankIndexValuePair, rank)};
int lenghts[] = {1, 1, 1};
MPI_Type_create_struct(3, lenghts, offsets, datatypes, &MPI_DOUBLE_INT_INT);
MPI_Type_commit(&MPI_DOUBLE_INT_INT);

/* Create the MINIMUM and MAXIMUM reduction operator for RankIndexValuePair */
MPI_Op MPI_MINLOC_RANKINDEXVALUE, MPI_MAXLOC_RANKINDEXVALUE;
MPI_Op_create(minlocRankIndexValuePair, 1, &MPI_MINLOC_RANKINDEXVALUE);
MPI_Op_create(maxlocRankIndexValuePair, 1, &MPI_MAXLOC_RANKINDEXVALUE);

// Uniformly divide data points to all workers and remaining data handled by
  master
n_data_per_worker = n_data/(world_size-1);
n_data_master = n_data % (world_size-1);

double data[n_data_per_worker][n_features];
int target[n_data_per_worker];

test_n_data_per_worker = test_n_data/(world_size-1);
test_n_data_master = test_n_data % (world_size-1);

double **test_data = (double **)malloc(sizeof(double *)*test_n_data_per_worker);
for(j=0; j<test_n_data_per_worker; ++j)
  test_data[j] = (double *)malloc(sizeof(double)*n_features);

int *test_target = (int *)malloc(sizeof(int)*test_n_data_per_worker);
int *predictions;

if(world_rank == MASTER)
{
  FILE *file = fopen(filename, "r");
  if(file == NULL)
  {
    perror("ERROR: Training dataset file couldn't be opened");
    // Assumption: No code beyond MPI_Abort will execute
    MPI_Abort(MPI_COMM_WORLD, errno);
  }

  for(i=1; i<world_size; ++i)
  {
    readLibsvmDataset(file, n_data_per_worker, n_features, data, target);
    MPI_Send(data, n_data_per_worker*n_features, MPI_DOUBLE, i, FROM_MASTER,
 MPI_COMM_WORLD);
    MPI_Send(target, n_data_per_worker, MPI_INT, i, FROM_MASTER, MPI_COMM_WORLD)
  ;
  }

  readLibsvmDataset(file, n_data_master, n_features, data, target);
  fclose(file);

  // Initialise parameters alpha, fcache error value, duality gap for all local
  training samples
  double alphas[n_data_master], fcache[n_data_master];
  for(j=0; j<n_data_master; ++j)
  {
    alphas[j] = 0;
    fcache[j] = -1*target[j];
  }

  local_duality_gap = computeDualityGap(n_data_master, alphas, target, fcache, b
  , C);
  MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE, MPI_SUM,
   MPI_COMM_WORLD);

  local_I_up = computebupIndex(n_data_master, alphas, target, fcache, C);
  local_I_low = computeblowIndex(n_data_master, alphas, target, fcache, C);

  // Compute (b_low, i_low) and (b_up, i_up)
  if(local_I_up == -1) local_b_up = INFINITY;
  else local_b_up = fcache[local_I_up];
```

```
if(local_I_low == -1) local_b_low = -INFINITY;
else local_b_low = fcache[local_I_low];

local_rank_b_up.rank = local_rank_b_low.rank = MASTER;

local_rank_b_up.value = local_b_up;
local_rank_b_up.index = local_I_up;

local_rank_b_low.value = local_b_low;
local_rank_b_low.index = local_I_low;

MPI_Allreduce(&local_rank_b_up, &global_rank_b_up, 1, MPI_DOUBLE_INT_INT,
MPI_MINLOC_RANKINDEXVALUE, MPI_COMM_WORLD);
MPI_Allreduce(&local_rank_b_low, &global_rank_b_low, 1, MPI_DOUBLE_INT_INT,
MPI_MAXLOC_RANKINDEXVALUE, MPI_COMM_WORLD);

// Assert that global (bup, blow) and (Iup, Ilow) are valid
assert(global_rank_b_up.index >= 0);
assert(global_rank_b_low.index >= 0);

if(global_rank_b_up.rank == MASTER)
{
  alpha1 = alphas[local_I_up];
  y1 = target[local_I_up];
  f1 = fcache[local_I_up];
  memcpy(data1, data[local_I_up], sizeof(double)*n_features);
}

MPI_Bcast(&alpha1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
MPI_Bcast(&y1, 1, MPI_INT, global_rank_b_up.rank, MPI_COMM_WORLD);
MPI_Bcast(&f1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
MPI_Bcast(&data1, n_features, MPI_DOUBLE, global_rank_b_up.rank,
MPI_COMM_WORLD);

if(global_rank_b_low.rank == MASTER)
{
  alpha2 = alphas[local_I_low];
  y2 = target[local_I_low];
  f2 = fcache[local_I_low];
  memcpy(data2, data[local_I_low], sizeof(double)*n_features);
}

MPI_Bcast(&alpha2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
MPI_Bcast(&y2, 1, MPI_INT, global_rank_b_low.rank, MPI_COMM_WORLD);
MPI_Bcast(&f2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
MPI_Bcast(&data2, n_features, MPI_DOUBLE, global_rank_b_low.rank,
MPI_COMM_WORLD);

int s;
double gamma, L, H;
double K11, K22, K12, eta;
double slope, change;

// Sequential Minimal Optimization - Update alpha1, alpha2 pairs
while( (global_duality_gap > tol*fabs(dual)) && (num_changed != 0) )
{
  // Start of Procedure: Take Step
  if((global_rank_b_up.rank == global_rank_b_low.rank) && (global_rank_b_up.
index == global_rank_b_low.index))
    num_changed = 0;

  else
  {
    s = y1 * y2;
    if(s == 1)
    {
      gamma = alpha1 + alpha2;

      if(gamma > C)
      {
        L = gamma - C;
        H = C;
      }

      else
      {
```

```
        L = 0;
        H = gamma;
      }

    }

    else
    {
      gamma = alpha1 - alpha2;

      if(gamma < 0)
      {
        L = -gamma;
        H = C;
      }

      else
      {
        L = 0;
        H = C - gamma;
      }
    }

    if(H <= L) num_changed = 0;

    else
    {
      K11 = kernelFunction(data1, data1, n_features);
      K22 = kernelFunction(data2, data2, n_features);
      K12 = kernelFunction(data1, data2, n_features);
      eta = 2*K12 - K11 - K22;

      if(eta < eps*(K11 + K12))
      {
        updated_alpha2 = alpha2 - y2*(f1 - f2)/eta;

        if(updated_alpha2 < L) updated_alpha2 = L;
        else if(updated_alpha2 > H) updated_alpha2 = H;
      }

      else
      {
        slope = y2*(f1 - f2);
        change = slope*(H - L);

        if(change != 0)
          if(slope > 0) updated_alpha2 = H;
          else updated_alpha2 = L;

        else
          updated_alpha2 = alpha2;
      }

      if(updated_alpha2 > (1 - eps)*C) updated_alpha2 = C;
      else if(updated_alpha2 < eps*C) updated_alpha2 = 0;

      if(fabs(updated_alpha2 - alpha2) < eps*(updated_alpha2 + alpha2 + eps))
        num_changed = 0;

      else
      {
        if(s == 1) updated_alpha1 = gamma - updated_alpha2;
        else updated_alpha1 = gamma + updated_alpha2;

        if(updated_alpha1 > (1-eps)*C) updated_alpha1 = C;
        else if(updated_alpha1 < eps*C) updated_alpha1 = 0;

        // Update the value of dual
        dual += -1*(updated_alpha1 - alpha1)*(f1 - f2)/y1 + 0.5*eta*((
updated_alpha1 - alpha1)/y1)*((updated_alpha1 - alpha1)/y1);
        num_changed = 1;
      }
    }

  } // End of Procedure: Take Step
```

```
    MPI_Bcast(&num_changed, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

    if(num_changed == 1)
    {
      MPI_Bcast(&updated_alpha1, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
      MPI_Bcast(&updated_alpha2, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
      MPI_Bcast(&dual, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);

      if(global_rank_b_up.rank == MASTER)
        alphas[local_I_up] = updated_alpha1;

      if(global_rank_b_low.rank == MASTER)
        alphas[local_I_low] = updated_alpha2;

      // update fcache error values for all local training data samples
      for(j=0; j<n_data_master; ++j)
        fcache[j] += (updated_alpha1 - alpha1)*y1*kernelFunction(data1, data[j],
 n_features) + (updated_alpha2 - alpha2)*y2*kernelFunction(data2, data[j],
n_features);

      local_I_up = computebupIndex(n_data_master, alphas, target, fcache, C);
      local_I_low = computeblowIndex(n_data_master, alphas, target, fcache, C);

      // Compute (b_low, i_low) and (b_up, i_up)
      if(local_I_up == -1) local_b_up = INFINITY;
      else local_b_up = fcache[local_I_up];

      if(local_I_low == -1) local_b_low = -INFINITY;
      else local_b_low = fcache[local_I_low];

      local_rank_b_up.rank = local_rank_b_low.rank = MASTER;

      local_rank_b_up.value = local_b_up;
      local_rank_b_up.index = local_I_up;

      local_rank_b_low.value = local_b_low;
      local_rank_b_low.index = local_I_low;

      MPI_Allreduce(&local_rank_b_up, &global_rank_b_up, 1, MPI_DOUBLE_INT_INT,
MPI_MINLOC_RANKINDEXVALUE, MPI_COMM_WORLD);
      MPI_Allreduce(&local_rank_b_low, &global_rank_b_low, 1, MPI_DOUBLE_INT_INT
, MPI_MAXLOC_RANKINDEXVALUE, MPI_COMM_WORLD);

      // Assert that global (bup, blow) and (Iup, Ilow) are valid
      assert(global_rank_b_up.index >= 0);
      assert(global_rank_b_low.index >= 0);

      if(global_rank_b_up.rank == MASTER)
      {
        alpha1 = alphas[local_I_up];
        y1 = target[local_I_up];
        f1 = fcache[local_I_up];
        memcpy(data1, data[local_I_up], sizeof(double)*n_features);
      }

      MPI_Bcast(&alpha1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
      MPI_Bcast(&y1, 1, MPI_INT, global_rank_b_up.rank, MPI_COMM_WORLD);
      MPI_Bcast(&f1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
      MPI_Bcast(&data1, n_features, MPI_DOUBLE, global_rank_b_up.rank,
MPI_COMM_WORLD);

      if(global_rank_b_low.rank == MASTER)
      {
        alpha2 = alphas[local_I_low];
        y2 = target[local_I_low];
        f2 = fcache[local_I_low];
        memcpy(data2, data[local_I_low], sizeof(double)*n_features);
      }

      MPI_Bcast(&alpha2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
      MPI_Bcast(&y2, 1, MPI_INT, global_rank_b_low.rank, MPI_COMM_WORLD);
      MPI_Bcast(&f2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
      MPI_Bcast(&data2, n_features, MPI_DOUBLE, global_rank_b_low.rank,
MPI_COMM_WORLD);

      b = (global_rank_b_low.value + global_rank_b_up.value)/2;
```

```
    local_duality_gap = computeDualityGap(n_data_master, alphas, target,
fcache, b, C);
    MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE,
MPI_SUM, MPI_COMM_WORLD);
  }


} // end of while loop

b = (global_rank_b_low.value + global_rank_b_up.value)/2;

local_duality_gap = computeDualityGap(n_data_master, alphas, target, fcache, b
, C);
MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE, MPI_SUM,
 MPI_COMM_WORLD);

printf("Training Phase Completed\n");

// Start of TESTING phase
FILE *test_file = fopen(test_filename, "r");
if(test_file == NULL)
{
  perror("ERROR: Testing dataset file couldn't be opened");
  // Assumption: No code beyond MPI_Abort will execute
  MPI_Abort(MPI_COMM_WORLD, errno);
}

// Read blocks of data and asyncronously distribute it uniformly to all
workers
for(i=1; i<world_size; ++i)
{
  readLibsvmDatasetPointers(test_file, test_n_data_per_worker, n_features,
test_data, test_target);

  for(j=0; j<test_n_data_per_worker; ++j)
    MPI_Send(test_data[j], n_features, MPI_DOUBLE, i, FROM_MASTER,
MPI_COMM_WORLD);

  MPI_Send(test_target, test_n_data_per_worker, MPI_INT, i, FROM_MASTER,
MPI_COMM_WORLD);
}

readLibsvmDatasetPointers(test_file, test_n_data_master, n_features, test_data
, test_target);
fclose(test_file);

// Predict the ouput values for local testing data samples and validate
int *predictions = computePredictions(n_features, n_data_master, data, target,
 test_n_data_master, test_data, alphas, b, kernelFunction);

local_n_correct_preds = countCorrectPreds(test_n_data_master, test_target,
predictions);
MPI_Reduce(&local_n_correct_preds, &global_n_correct_preds, 1, MPI_INT,
MPI_SUM, MASTER, MPI_COMM_WORLD);

local_n_true_pos_preds = countTruePositivePreds(test_n_data_master,
test_target, predictions);
MPI_Reduce(&local_n_true_pos_preds, &global_n_true_pos_preds, 1, MPI_INT,
MPI_SUM, MASTER, MPI_COMM_WORLD);

local_n_pos_targets = countPositiveTargets(test_n_data_master, test_target,
predictions);
MPI_Reduce(&local_n_pos_targets, &global_n_pos_targets, 1, MPI_INT, MPI_SUM,
MASTER, MPI_COMM_WORLD);

local_n_pos_preds = countCorrectPreds(test_n_data_master, test_target,
predictions);
MPI_Reduce(&local_n_pos_preds, &global_n_pos_preds, 1, MPI_INT, MPI_SUM,
MASTER, MPI_COMM_WORLD);

double accuracy, precision, recall, f1_score;

accuracy = global_n_correct_preds*1.00/test_n_data;
precision = global_n_true_pos_preds*1.00/global_n_pos_preds;
recall = global_n_true_pos_preds*1.00/global_n_pos_targets;
```

```
    f1_score = (2*precision*recall)/(precision+recall);

    printf("Accuracy = %lf\n", accuracy);
    printf("Precision = %lf\n", precision);
    printf("Recall = %lf\n", recall);
    printf("F1-Score = %lf\n", f1_score);

  } // end of MASTER code

  else
  {
    MPI_Recv(data, n_data_per_worker*n_features, MPI_DOUBLE, MASTER, FROM_MASTER,
    MPI_COMM_WORLD, &status);
    MPI_Recv(target, n_data_per_worker, MPI_INT, MASTER, FROM_MASTER,
    MPI_COMM_WORLD, &status);

    // Initialise parameters alpha, fcache error value, duality gap for all local
    training samples
    double alphas[n_data_per_worker], fcache[n_data_per_worker];
    for(j=0; j<n_data_per_worker; ++j)
    {
      alphas[j] = 0;
      fcache[j] = -1*target[j];
    }

    local_duality_gap = computeDualityGap(n_data_per_worker, alphas, target,
    fcache, b, C);
    MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE, MPI_SUM,
     MPI_COMM_WORLD);

    local_I_up = computebupIndex(n_data_per_worker, alphas, target, fcache, C);
    local_I_low = computeblowIndex(n_data_per_worker, alphas, target, fcache, C);

    // Compute (b_low, i_low) and (b_up, i_up)
    if(local_I_up == -1) local_b_up = INFINITY;
    else local_b_up = fcache[local_I_up];

    if(local_I_low == -1) local_b_low = -INFINITY;
    else local_b_low = fcache[local_I_low];

    local_rank_b_up.rank = local_rank_b_low.rank = world_rank;

    local_rank_b_up.value = local_b_up;
    local_rank_b_up.index = local_I_up;

    local_rank_b_low.value = local_b_low;
    local_rank_b_low.index = local_I_low;

    MPI_Allreduce(&local_rank_b_up, &global_rank_b_up, 1, MPI_DOUBLE_INT_INT,
    MPI_MINLOC_RANKINDEXVALUE, MPI_COMM_WORLD);
    MPI_Allreduce(&local_rank_b_low, &global_rank_b_low, 1, MPI_DOUBLE_INT_INT,
    MPI_MAXLOC_RANKINDEXVALUE, MPI_COMM_WORLD);

    if(global_rank_b_up.rank == world_rank)
    {
      alpha1 = alphas[local_I_up];
      y1 = target[local_I_up];
      f1 = fcache[local_I_up];
      memcpy(data1, data[local_I_up], sizeof(double)*n_features);
    }

    MPI_Bcast(&alpha1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&y1, 1, MPI_INT, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&f1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&data1, n_features, MPI_DOUBLE, global_rank_b_up.rank,
    MPI_COMM_WORLD);

    if(global_rank_b_low.rank == world_rank)
    {
      alpha2 = alphas[local_I_low];
      y2 = target[local_I_low];
      f2 = fcache[local_I_low];
      memcpy(data2, data[local_I_low], sizeof(double)*n_features);
    }

    MPI_Bcast(&alpha2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
```

```
MPI_Bcast(&y2, 1, MPI_INT, global_rank_b_low.rank, MPI_COMM_WORLD);
MPI_Bcast(&f2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
MPI_Bcast(&data2, n_features, MPI_DOUBLE, global_rank_b_low.rank,
MPI_COMM_WORLD);

// Sequential Minimal Optimization - Update alpha1, alpha2 pairs
while( (global_duality_gap > tol*fabs(dual)) && (num_changed != 0) )
{
  MPI_Bcast(&num_changed, 1, MPI_INT, MASTER, MPI_COMM_WORLD);

  if(num_changed == 1)
  {
    MPI_Bcast(&updated_alpha1, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    MPI_Bcast(&updated_alpha2, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);
    MPI_Bcast(&dual, 1, MPI_DOUBLE, MASTER, MPI_COMM_WORLD);

    if(global_rank_b_up.rank == world_rank)
      alphas[local_I_up] = updated_alpha1;

    if(global_rank_b_low.rank == world_rank)
      alphas[local_I_low] = updated_alpha2;

    // update fcache error values for all local training data samples
    for(j=0; j<n_data_per_worker; ++j)
      fcache[j] += (updated_alpha1 - alpha1)*y1*kernelFunction(data1, data[j],
 n_features) + (updated_alpha2 - alpha2)*y2*kernelFunction(data2, data[j],
n_features);

    local_I_up = computebupIndex(n_data_per_worker, alphas, target, fcache, C)
;
    local_I_low = computeblowIndex(n_data_per_worker, alphas, target, fcache,
C);

    // Compute (b_low, i_low) and (b_up, i_up)
    if(local_I_up == -1) local_b_up = INFINITY;
    else local_b_up = fcache[local_I_up];

    if(local_I_low == -1) local_b_low = -INFINITY;
    else local_b_low = fcache[local_I_low];

    local_rank_b_up.rank = local_rank_b_low.rank = world_rank;

    local_rank_b_up.value = local_b_up;
    local_rank_b_up.index = local_I_up;

    local_rank_b_low.value = local_b_low;
    local_rank_b_low.index = local_I_low;

    MPI_Allreduce(&local_rank_b_up, &global_rank_b_up, 1, MPI_DOUBLE_INT_INT,
MPI_MINLOC_RANKINDEXVALUE, MPI_COMM_WORLD);
    MPI_Allreduce(&local_rank_b_low, &global_rank_b_low, 1, MPI_DOUBLE_INT_INT
, MPI_MAXLOC_RANKINDEXVALUE, MPI_COMM_WORLD);

    if(global_rank_b_up.rank == world_rank)
    {
      alpha1 = alphas[local_I_up];
      y1 = target[local_I_up];
      f1 = fcache[local_I_up];
      memcpy(data1, data[local_I_up], sizeof(double)*n_features);
    }

    MPI_Bcast(&alpha1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&y1, 1, MPI_INT, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&f1, 1, MPI_DOUBLE, global_rank_b_up.rank, MPI_COMM_WORLD);
    MPI_Bcast(&data1, n_features, MPI_DOUBLE, global_rank_b_up.rank,
MPI_COMM_WORLD);

    if(global_rank_b_low.rank == world_rank)
    {
      alpha2 = alphas[local_I_low];
      y2 = target[local_I_low];
      f2 = fcache[local_I_low];
      memcpy(data2, data[local_I_low], sizeof(double)*n_features);
    }

    MPI_Bcast(&alpha2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
```

```
        MPI_Bcast(&y2, 1, MPI_INT, global_rank_b_low.rank, MPI_COMM_WORLD);
        MPI_Bcast(&f2, 1, MPI_DOUBLE, global_rank_b_low.rank, MPI_COMM_WORLD);
        MPI_Bcast(&data2, n_features, MPI_DOUBLE, global_rank_b_low.rank,
    MPI_COMM_WORLD);

        b = (global_rank_b_low.value + global_rank_b_up.value)/2;

        local_duality_gap = computeDualityGap(n_data_per_worker, alphas, target,
    fcache, b, C);
        MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE,
    MPI_SUM, MPI_COMM_WORLD);
      }
    } // end of while loop

    b = (global_rank_b_low.value + global_rank_b_up.value)/2;

    local_duality_gap = computeDualityGap(n_data_per_worker, alphas, target,
    fcache, b, C);
    MPI_Allreduce(&local_duality_gap, &global_duality_gap, 1, MPI_DOUBLE, MPI_SUM,
     MPI_COMM_WORLD);

    // Start of TESTING phase
    for(j=0; j<test_n_data_per_worker; ++j)
      MPI_Recv(test_data[j], n_features, MPI_DOUBLE, MASTER, FROM_MASTER,
    MPI_COMM_WORLD, &status);

    MPI_Recv(test_target, test_n_data_per_worker, MPI_INT, MASTER, FROM_MASTER,
    MPI_COMM_WORLD, &status);

    // Predict the ouput values for local testing data samples
    int *predictions = computePredictions(n_features, n_data_per_worker, data,
    target, test_n_data_per_worker, test_data, alphas, b, kernelFunction);

    local_n_correct_preds = countCorrectPreds(test_n_data_per_worker, test_target,
     predictions);
    MPI_Reduce(&local_n_correct_preds, &global_n_correct_preds, 1, MPI_INT,
    MPI_SUM, MASTER, MPI_COMM_WORLD);

    local_n_true_pos_preds = countTruePositivePreds(test_n_data_per_worker,
    test_target, predictions);
    MPI_Reduce(&local_n_true_pos_preds, &global_n_true_pos_preds, 1, MPI_INT,
    MPI_SUM, MASTER, MPI_COMM_WORLD);

    local_n_pos_targets = countPositiveTargets(test_n_data_per_worker, test_target
    , predictions);
    MPI_Reduce(&local_n_pos_targets, &global_n_pos_targets, 1, MPI_INT, MPI_SUM,
    MASTER, MPI_COMM_WORLD);

    local_n_pos_preds = countCorrectPreds(test_n_data_per_worker, test_target,
    predictions);
    MPI_Reduce(&local_n_pos_preds, &global_n_pos_preds, 1, MPI_INT, MPI_SUM,
    MASTER, MPI_COMM_WORLD);
  }


  MPI_Finalize();

  return 0;
}

/* Read first 'n_data' data points from current file ptr position 'file' into pre-
    allocated arrays 'data' and 'target' */
int readLibsvmDataset(FILE *file, int n_data, int n_features, double data[n_data][
    n_features], int target[n_data])
{
  char buffer[BUFFER_SIZE];

  char *feature;
  int feature_index;
  double feature_value;

  size_t delimiter_pos, seperator_pos;

  int i=0, j;
  while( (i<n_data) && (fgets(buffer, sizeof(buffer), file) != NULL))
  {
```

```
    delimiter_pos = strcspn(buffer, " ");
    buffer[delimiter_pos] = '\0';
    target[i] = atoi(buffer);

    // skip spaces to first feature index:value pair
    feature = buffer + delimiter_pos+1;
    while(*feature == ' ')
      feature++;

    // initialise all features of ith data as 0
    for(j=0; j<n_features; ++j)
      data[i][j] = 0;

    while((delimiter_pos = strcspn(feature, " ")) != strlen(feature))
    {
      feature[delimiter_pos] = '\0';

      // extract the feature index and feature value
      seperator_pos = strcspn(feature, ":");
      feature[seperator_pos] = '\0';

      feature_index = atoi(feature);
      feature_value = atof(feature + seperator_pos+1);
      data[i][feature_index] = feature_value;

      feature += delimiter_pos+1;
    }

    i++;
  }

  return i;
}

int readLibsvmDatasetPointers(FILE *file, int n_data, int n_features, double **
    data, int *target)
{
  char buffer[BUFFER_SIZE];

  char *feature;
  int feature_index;
  double feature_value;

  size_t delimiter_pos, seperator_pos;

  int i=0, j;
  while( (i<n_data) && (fgets(buffer, sizeof(buffer), file) != NULL))
  {
    delimiter_pos = strcspn(buffer, " ");
    buffer[delimiter_pos] = '\0';
    target[i] = atoi(buffer);

    // skip spaces to first feature index:value pair
    feature = buffer + delimiter_pos+1;
    while(*feature == ' ')
      feature++;

    // initialise all features of ith data as 0
    for(j=0; j<n_features; ++j)
      data[i][j] = 0;

    while((delimiter_pos = strcspn(feature, " ")) != strlen(feature))
    {
      feature[delimiter_pos] = '\0';

      // extract the feature index and feature value
      seperator_pos = strcspn(feature, ":");
      feature[seperator_pos] = '\0';

      feature_index = atoi(feature);
      feature_value = atof(feature + seperator_pos+1);
      data[i][feature_index] = feature_value;

      feature += delimiter_pos+1;
    }
```

```
    i++;
  }

  return i;
}

double computeDualityGap(int n_data, double alphas[n_data], int target[n_data],
    double fcache[n_data], double b, double C)
{
  double duality_gap, eps;
  duality_gap = 0;

  int j;
  for(j=0; j<n_data; ++j)
  {
    duality_gap += alphas[j] * target[j] * fcache[j];

    if(target[j] * (b - fcache[j]) > 0)
      eps = C * target[j] * (b - fcache[j]);
    else
      eps = 0;

    duality_gap += eps;
  }

  return duality_gap;
}

int computebupIndex(int n_data, double alphas[n_data], int target[n_data], double
    fcache[n_data], double C)
{
  int j;

  int I_up = -1;
  double b_up = INFINITY;

  for(j=0; j<n_data; ++j)
    if( ((alphas[j] > 0) && (alphas[j] < C)) || ((target[j] == 1) && (alphas[j] ==
     0)) || ((target[j] == -1) && (alphas[j] == C)) )
      if(fcache[j] < b_up)
      {
        I_up = j;
        b_up = fcache[j];
      }

  return I_up;
}

int computeblowIndex(int n_data, double alphas[n_data], int target[n_data], double
     fcache[n_data], double C)
{
  int j;

  int I_low = -1;
  double b_low = -INFINITY;

  for(j=0; j<n_data; ++j)
    if( ((alphas[j] > 0) && (alphas[j] < C)) || ((target[j] == 1) && (alphas[j] ==
     C)) || ((target[j] == -1) && (alphas[j] == 0)) )
      if(fcache[j] > b_low)
      {
        I_low = j;
        b_low = fcache[j];
      }

  return I_low;
}

void minlocRankIndexValuePair(void *in, void *inout, int *len, MPI_Datatype *type)
{
  struct RankIndexValuePair *invals = in;
  struct RankIndexValuePair *inoutvals = inout;

  int i;
  for(i=0; i<*len; ++i)
    if(inoutvals[i].value > invals[i].value)
```

```
    {
      inoutvals[i].value = invals[i].value;
      inoutvals[i].index = invals[i].index;
      inoutvals[i].rank = invals[i].rank;
    }
}

void maxlocRankIndexValuePair(void *in, void *inout, int *len, MPI_Datatype *type)
{
  struct RankIndexValuePair *invals = in;
  struct RankIndexValuePair *inoutvals = inout;

  int i;
  for(i=0; i<*len; ++i)
    if(inoutvals[i].value < invals[i].value)
    {
      inoutvals[i].value = invals[i].value;
      inoutvals[i].index = invals[i].index;
      inoutvals[i].rank = invals[i].rank;
    }
}

double dotProduct(double *data1, double *data2, int n_features)
{
  double dotProduct = 0;

  int i;
  for(i=0; i<n_features; ++i)
    dotProduct += data1[i] * data2[i];

  return dotProduct;
}

int *computePredictions(int n_features, int train_n_data, double train_data[
    train_n_data][n_features], int train_target[train_n_data], int test_n_data,
    double **test_data, double alphas[train_n_data], double b, double (*
    kernelFunction)(double *, double *, int))
{
  int *predictions = (int *)malloc(sizeof(int)*test_n_data);

  int i, j;

  double output;
  for(i=0; i<test_n_data; ++i)
  {
    output = 0;
    for(j=0; j<train_n_data; ++j)
      output += alphas[j]*train_target[j]*kernelFunction(train_data[j], test_data[
    i], n_features);
    output += b;

    if(output >= 0) predictions[i] = 1;
    else predictions[i] = -1;
  }

  return predictions;
}

int countCorrectPreds(int n_data, int target[n_data], int predictions[n_data])
{
  int j;

  int n_correct_preds = 0;
  for(j=0; j<n_data; ++j)
    if(target[j] == predictions[j])
      n_correct_preds += 1;

  return n_correct_preds;
}

int countTruePositivePreds(int n_data, int target[n_data], int predictions[n_data
    ])
{
  int j;

  int n_true_pos_preds = 0;
```

```
  for(j=0; j<n_data; ++j)
    if((target[j] > 0) && (target[j] == predictions[j]))
      n_true_pos_preds += 1;

  return n_true_pos_preds;
}

int countPositiveTargets(int n_data, int target[n_data], int predictions[n_data])
{
  int j;

  int n_pos_targets = 0;
  for(j=0; j<n_data; ++j)
    if(target[j] > 0)
      n_pos_targets += 1;

  return n_pos_targets;
}

int countPositivePreds(int n_data, int target[n_data], int predictions[n_data])
{
  int j;

  int n_pos_preds = 0;
  for(j=0; j<n_data; ++j)
    if(predictions[j] > 0)
      n_pos_preds += 1;

  return n_pos_preds;
}
```