Analyzing the Parallelizablity of SVM Classification Algorithm using OpenMP

R Mukesh (CED15I002)

IIITDM Kancheepuram

Abstract

Support Vector Machine (SVM) is among the most popular algorithms in machine learning literature. This paper aims to analyze the performance of parallelized SVM classification algorithm using OpenMP.

1 Theory

1.1 Overview of Support Vector Machines

Consider a binary classification dataset,

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

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.

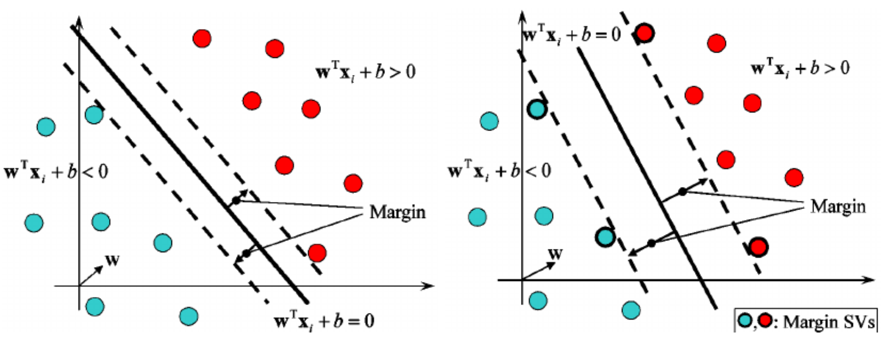


Figure 1: Optimal margin linear seperating hyperplane.

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:

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

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

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

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 The SMO Algorithm

Sequential Minimal Optimization (SMO) is an efficient algorithm to the solve the SVM quadratic programming problem in (1).

Repeat until convergence {

1. Choose a pair of langrage multipliers α_i, α_j to jointly optimize (using an heuristic that maximizes the size of step in optimizing $\Psi(\alpha)$).
2. If the choosen pair of langrage multipliers α_i, α_j can make a positive step in optimizing $\Psi(\alpha)$, reoptimize $\Psi(\alpha)$ with respect to α_i, α_j, while holding all other $\alpha_k s (k \neq i, j)$ fixed.

}

For convergence, the KKT conditions in (2) must be satisfied for all $\alpha_i s$.

2 Parallelizing the SMO Algorithm

Sequential Minimal Optimization (SMO) is an **iterative** optimization algorithm and the scope for parallelization is limited. The operations involved in an iteration of the optimization (such as searching α_i, α_j pair to update and updating error between predicted and actual value for each training instance) can be performed parallely:

1. **Initializing and updating error cache for each training instance:**

   In each update of an α_i, α_j pair, the w and b paramters of the model are also updated. Consequentially, the error cache that represents the difference between the model's predicted and actual value for each training instance is updated parallely.

   The update of the error cache for each training instance was parallelized using the OpenMP construct: `#pragma omp parallel for`.

```
#pragma omp parallel for private(i)
for(i = 0; i< m; ++i)
{

  Initialize or update error cache for the ith training instance

}
```

2. **Searching across α_i, α_j pairs for one that can make positive step in optimizing the objective $\Psi(\alpha)$:**

   Pairs of α_i, α_j are parallely tested until a pair is found that can be updated to make positive step in optimizing the problem in (1). Consequentially, the time taken to find the α_i, α_j pair to update and thereby the time taken for the convergence of the SMO algorithm are drastically reduced.

   The searching of the α_i, α_j pair to update was parallelized using the OpenMP constructs: #pragma omp parallel for, #pragma omp critical, #pragma omp cancel for, #pragma omp cancellation point for, and using private and shared thread syncronization variables.

```
Find an alpha[i] that violates the KKT-conditions

// shared sync var to indicate success in finding alpha pair to update
valid_alphaj_found = false;

#pragma omp parallel
{
  #pragma omp for priavte(j)
  for(j=0; j<m; ++j)
  {
    #pragma omp cancellation point for

     if((i != j) && (valid_alphaj_found == false))
       if(alpha[i], alpha[j] can make positive step in optimization)
       {
         // local sync var to indicate success in finding alpha pair to
update by this iteration
         bool thread_valid_alphaj_found = false;

         #pragma omp critical
         {
           if(valid_alphaj_found == false)
           {
             valid_alphaj_found = true;
             thread_valid_alphaj_found = true;

             Update the alpha pair, model parameters and error cache
           }
         } // end of omp critical

         if(thread_valid_alphaj_found == true)
         {
           #pragma omp cancel for
         }
       }
  } // end of omp for
} // end of omp parallel

if vaid_alphaj_found became true
  a valid alpha[i], alpha[j] was found and updated

else
  no such alpha[i], alpha[j] pair exists for the given alpha[i]
```

3. **Computing dot product of feature vectors:**

   The popularity of SVM can be largely attributed to the kernel trick that enables us find the optimal margin linear seperating hyperplane in a **higher dimensional feature space**. The kernel function (such as linear kernel) often involves the computation of the dot product of the the feature vectors.

   The computation of the dot product of the feature vectors can be parallelized using OpenMP construct: #pragma omp parallel for with reduction.

```
dot_product = 0;

#pragma omp parallel for private(i) reduction(+:dot_product)
for(i=0; i<n; ++i)
  dot_product += feature_vector1[i] * feature_vector2[i];
```

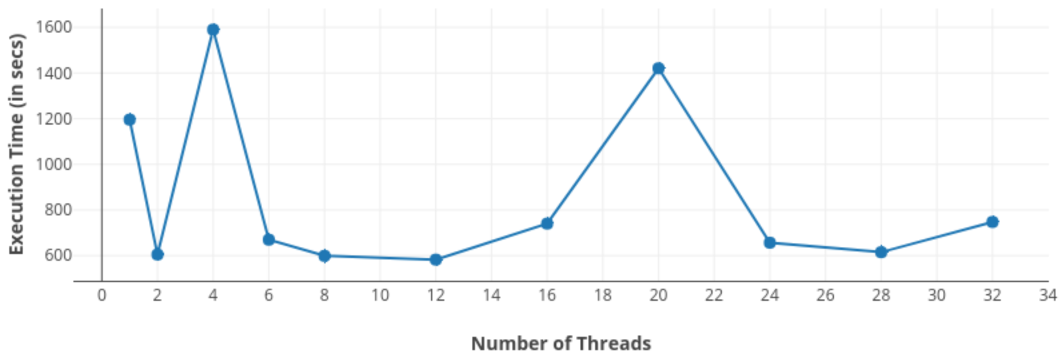
3 Results [1]

The experiments were performed on pre-processed excerpts of the Adult Data Set from UCI Machine Learning Repository. The pre-processed training dataset a1a and testing dataset a1a.t can be downloaded from the LIBSVM datasets page.

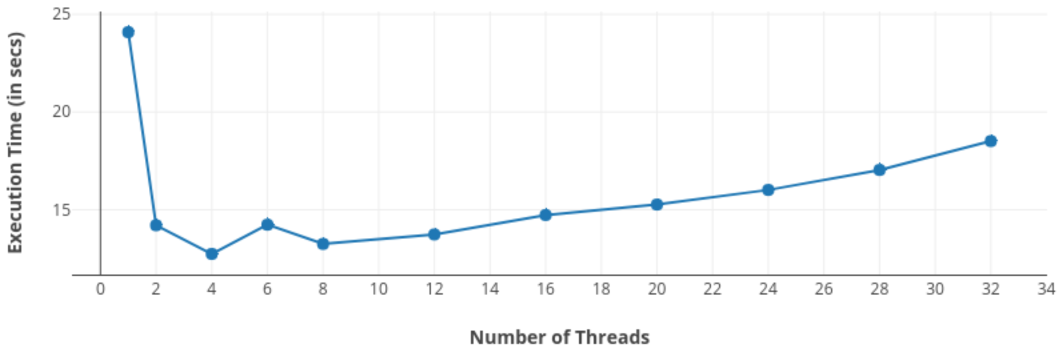
Number of classes : 2
Number of data : 1,605 / 30,956 (testing)
Number of features : 123 / 123 (testing)

Number of Threads	**Execution Time (in seconds)**	
	Fitting with training data	**Predicting for testing data**
1	1195.73	24.0849
2	604.719	14.2085
4	1589.9	12.7448
6	669.508	14.2376
8	598.881	13.2541
12	582.259	13.7397
16	740.158	14.7316
20	1420.45	15.2676
24	656.442	16.0162
28	614.798	17.0384
32	747.298	18.5145

The speed of convergence of the fitting of SVM model is dictated by the path taken (i.e., the choice of α_i, α_j pairs at each iteration of the optimization algorithm). Since, the choice of α_i, α_j pairs is partly random, the time taken for the convergence of the SVM fitting is likely to vary across runs.



(a) Performance of the fit method in SVM learning algorithm



(b) Performance of the predict method in SVM learning algorithm

Figure 2: Number of Threads vs Execution Time for fit and predict methods of SVM learning algorithm in OpenMP

[1]The experiments were conducted on a computer with 6^{th} Generation Intel(R) Core(TM) i7-6500U Processor (4M Cache, upto 3.10 GHz) and 8GB Single Channel DDR3L 1600M Hz (4GBx2) RAM.

4 Calculation

The parallel fraction of the algorithm can be computed using the following equation:

$$f = \frac{(1 - T_p/T_1)}{(1 - 1/p)} \tag{3}$$

For Fit Method of SVM Learning Algorithm,

$$f = \frac{(1 - 582.259/1195.73)}{(1 - 1/12)} = \mathbf{0.559692482\ (55.97\ \%)}$$

For Predict Method of SVM Learning Algorithm,

$$f = \frac{(1 - 12.7448/24.0849)}{(1 - 1/4)} = \mathbf{0.627784767\ (62.78\ \%)}$$

5 Inferences

1. The **fit and predict methods** of the parallelized SVM classification algorithm have parallel fractions of **55.97 %** and **62.78 %** respectively.

2. The execution time of the fit method of the parallelized SVM classification algorithm varies significantly across runs (for a fixed number of threads) due to the randomness associated with the **path taken towards convergence**.

3. The parallelization of dot product computation deteriorates the performance of the algorithm due to the limitations of the hardware and the **overhead** associated with creation and switching of threads.

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 OpenMP 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-openmp-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-openmp-codes
   ```

3. Build the code using the make utility.

   ```
   $ make
   ```

   This generates the intermediate object files and executables for the test programs.

4. Execute the test program executable generated by the make script.

   For example: To run the sample test program '**test1.cpp**' located in the '**test**' directory.

   ```
   $ ./build/bin/test1
   ```

   Write your own test programs using the functions defined in the header files in the '**include**' directory. Rebuild the code using the make utility and execute the generated test program executable located in '**build/bin**' directory.

Appendix B Complete Source Code

Structure of the project:

```
├── include
│   ├── dataset.hpp
│   ├── classifier.hpp
│   ├── kernel.hpp
│   └── validation.hpp
├── src
│   ├── dataset.cpp
│   ├── classifier.cpp
│   ├── svm.cpp
│   ├── kernel.cpp
│   └── validation.cpp
├── test
│   └── test1.cpp
└── datasets
    ├── a1a
    └── a1a.t
```

include/dataset.hpp

```cpp
#ifndef DATASET_HPP
#define DATASET_HPP

#include <vector>
using namespace std;

/* An classification dataset with exclusively numerical features */
template <typename T>
class Dataset
{
public:
  vector<int> target;
  vector<vector<T> > data;

  int n_data;
  int n_features;
};

/* Read a classification dataset from a libsvm file format */
template <typename T>
Dataset<T> readLibsvmDataset(const char *filename, int n_data, int n_features);

#endif
```

include/classifier.hpp

```cpp
#ifndef CLASSIFIER_HPP
#define CLASSIFIER_HPP

#include <vector>
using namespace std;

#include "dataset.hpp"

/* Classifier base class from which all classifiers derive */
template <typename T>
class Classifier
{
  // <define the parameters of the derived classifiers here>

public:
  // Fit the model with the dataset to learn parameters
  virtual void fit(const Dataset<T>& dataset) = 0;

  // Predict the class label for test data
  virtual int predict(const vector<T>& data) = 0;

  // Predict the class labels for test dataset
```

```
  vector<int> predict(const Dataset<T>& dataset);
};

/* Support Vector Machine Classification Model */
template <typename T>
class SVC: public Classifier<T>
{
  // Parameters of the SVC classifier: C, tol, Kernel function
  double C, tol, (*kernelFunction)(vector<T>, vector<T>), eps;

  // Dataset with which model was fitted (used in prediction)
  Dataset<T> dataset;

  // Parameters to be learnt from dataset: alphas[n_data], b
  double *alphas, b;

  // error between prediction f(x) and actual value for examples in training data
  double *errors;

  // Helper functions to find alpha i (given alpha j) and update alpha i, alpha j
    pair
  int findUpdateAlphaPair(int alpha2_index);
  int updateAlphaPair(int alpha1_index, int alpha2_index);

  // Syncronization variables to indicate that alpha i that can make positive
    progress was found
  int valid_alpha1_found;

public:

  // Initialize the parameters for SVM Classifier
  SVC(double C, double tol, double eps, double (*kernelFunction)(vector<T>, vector
    <T>));

  void fit(const Dataset<T>& dataset);
  int predict(const vector<T>& data);

  using Classifier<T>::predict;
};

#endif
```

include/kernels.hpp

```
/* The file contains various default kernels for use in SVM classifier */
#ifndef KERNELS_HPP
#define KERNELS_HPP

#include <vector>
using namespace std;

template <typename T>
double dotProduct(vector<T>, vector<T>);

#endif
```

include/validation.hpp

```
#ifndef VALIDATION_HPP
#define VALIDATION_HPP

#include <vector>
using namespace std;

// Computes the accuracy of the predictions by the classifier against true target
    values
double computeAccuracy(const vector<int>& target, const vector<int>& predictions);

// Computes the precision of the predictions by the classifier against true target
     values
double computePrecision(const vector<int>& target, const vector<int>& predictions)
    ;

// Computes the recall of the predictions by the classifier against true target
    values
```

```
double computeRecall(const vector<int>& target, const vector<int>& predictions);

// Computes the F1-score of the predictions by the classifier against true target
    values
double computeF1Score(const vector<int>& target, const vector<int>& predictions);

#endif
```

src/dataset.cpp

```
#include "dataset.hpp"

#include <fstream>
#include <string>

/* Read a classification dataset from a libsvm file format */
template <typename T>
Dataset<T> readLibsvmDataset(const char *filename, int n_data, int n_features)
{
  Dataset<T> dataset;

  ifstream file(filename);
  if(file.is_open())
  {
    string line, feature;
    size_t current_pos, delimiter_pos, feature_delimiter_pos;

    int i = 0, target, feature_index;
    T feature_value;
    vector<T> feature_vector(n_features);

    while((i < n_data || n_data == -1) && getline(file, line))
    {
      fill(feature_vector.begin(), feature_vector.end(), 0);

      delimiter_pos = line.find(' ');
      target = stoi(line.substr(0, delimiter_pos));

      current_pos = delimiter_pos + 1;

      // skip the spaces after target
      while((current_pos < line.length()) && (line[current_pos] == ' '))
        current_pos++;

      while((delimiter_pos != string::npos) && (current_pos < line.length()))
      {
        delimiter_pos = line.find(' ', current_pos);
        feature = line.substr(current_pos, delimiter_pos - current_pos);

        feature_delimiter_pos = feature.find(':');

        feature_index = stoi(feature.substr(0, feature_delimiter_pos));
        feature_value = (T)stod(feature.substr(feature_delimiter_pos + 1));
        feature_vector[feature_index-1] = feature_value;

        current_pos = delimiter_pos + 1;
      }

      dataset.data.push_back(feature_vector);
      dataset.target.push_back(target);

      i++;
    }

    dataset.n_data = i;
    dataset.n_features = n_features;
  }

  return dataset;
}

// Explicit declaration of templated class and function
template class Dataset<int>;
template Dataset<int> readLibsvmDataset<int>(const char *, int, int);

template class Dataset<float>;
```

```
template Dataset<float> readLibsvmDataset<float>(const char *, int, int);

template class Dataset<double>;
template Dataset<double> readLibsvmDataset<double>(const char *, int, int);
```

src/classifier.cpp

```
#include "classifier.hpp"

template <typename T>
vector<int> Classifier<T>::predict(const Dataset<T>& dataset)
{
  int prediction;
  vector<int> predictions;

  for(int i=0; i<(int)dataset.data.size(); ++i)
  {
    prediction = predict(dataset.data[i]);
    predictions.push_back(prediction);
  }

  return predictions;
}

// Explicit instantiation of templated class and function
template class Classifier<int>;
template class Classifier<float>;
template class Classifier<double>;
```

src/svm.cpp

```
#include "classifier.hpp"

#include <cstdlib>
#include <algorithm>
#include <cmath>

#include <iostream>
using namespace std;

/* SVM Classifier: Methods to initialize parameters, fit model and predict labels
    -- START */
template <typename T>
SVC<T>::SVC(double C, double tol, double eps, double (*kernelFunction)(vector<T>,
    vector<T>))
{
  SVC::C = C;
  SVC::tol = tol;
  SVC::kernelFunction = kernelFunction;
  SVC::eps = eps;

}

template <typename T>
void SVC<T>::fit(const Dataset<T>& dataset)
{
  // Copy the dataset to classifier
  SVC::dataset = dataset;

  int i, j;

  alphas = (double *)malloc(sizeof(double) * dataset.n_data);
  fill_n(alphas, dataset.n_data, 0);
  b = 0;

  // Calculate the error for each data sample
  errors = (double *)malloc(sizeof(double) * dataset.n_data);

  // #pragma omp parallel for default(none) private(i) shared(dataset, errors)
  #pragma omp parallel for private(i)
  for(i=0; i<dataset.n_data; ++i)
    errors[i] = -1 * dataset.target[i];

  // Perform optimization of alpha pairs using SMO (Sequential Minimal
    Optimization)
```

```
  int examine_all = 1, num_changed = 0;
  double kkt_parameter;

  while((num_changed > 0) || (examine_all == 1))
  {
    num_changed = 0;

    for(j=0; j<dataset.n_data; ++j)
    {
      // Iterates alternatively single scans through entire dataset and multiple
    scans through non-bound training examples
      if( (examine_all == 1) || ((alphas[j] > 0) && (alphas[j] < C)) )
      {
        // Identify the training examples (alpha j) that violate KKT conditions
        kkt_parameter = dataset.target[j] * errors[j];
        if( ((kkt_parameter < -1*tol) && (alphas[j] < C)) || ((kkt_parameter > tol
    ) && (alphas[j] > 0)) )
          num_changed += findUpdateAlphaPair(j);
      }
    }

    if(examine_all == 1)
      examine_all = 0;

    else if(num_changed == 0)
      examine_all = 1;
  }

  free(errors);
}

template <typename T>
int SVC<T>::predict(const vector<T>& data)
{
  double output = 0;

  int i;

  // #pragma omp parallel for default(none) private(i) shared(dataset, alphas,
    kernelFunction, data) reduction(+:output)
  #pragma omp parallel for private(i) reduction(+:output)
  for(i=0; i<dataset.n_data; ++i)
    if(alphas[i] > 0)
      output += alphas[i] * dataset.target[i] * kernelFunction(dataset.data[i],
    data);

  output -= b;

  if(output >= 0)
    return 1;

  else
    return -1;
}

template <typename T>
int SVC<T>::findUpdateAlphaPair(int alpha2_index)
{
  int i;

  double estimated_step, max_estimated_step = -1;
  int alpha1_index = -1;

  // Iterate through all non-bound training examples to find other example for
    optimization using heuristic
  for(i=0; i<dataset.n_data; ++i)
    if((i!=alpha2_index) && (alphas[i] > 0) && (alphas[i] < C))
    {
      // Estimated step size of optimization
      estimated_step = errors[i] - errors[alpha2_index];
      if(estimated_step < 0)
        estimated_step *= -1;

      if(estimated_step > max_estimated_step)
        #pragma omp critical
        if(estimated_step > max_estimated_step)
```

```
      {
        max_estimated_step = estimated_step;
        alpha1_index = i;
      }
  }

  if((alpha1_index >= 0) && updateAlphaPair(alpha1_index, alpha2_index))
    return 1;

  // int random_offset = random() % dataset.n_data;

  // Parallely search across non-bound alphas for alpha1 that makes positive step
  valid_alpha1_found = 0;

  // #pragma omp parallel for default(none) private(i, alpha1_index) shared(
    dataset, alphas, alpha2_index, C)
  # pragma omp parallel
  {
    #pragma omp for private(i, alpha1_index)
    for(i=0; i<dataset.n_data; ++i)
    {
      #pragma omp cancellation point for

      // alpha1_index = (i + random_offset) % dataset.n_data;
      alpha1_index = i;

      if((valid_alpha1_found == 0) && (alpha1_index != alpha2_index) && (alphas[
    alpha1_index] > 0) && (alphas[alpha1_index] < C))
      {
        // SVC<T>::updateAlphaPair has been expanded to circumvent oprphaned
    cancellation point problem
        #pragma omp cancellation point for

        if(valid_alpha1_found == 0)
        {
          double alpha1_value, alpha2_value, updated_alpha1_value,
    updated_alpha2_value, old_b_value;
          alpha1_value = alphas[alpha1_index];
          alpha2_value = alphas[alpha2_index];
          old_b_value = b;

          int s = dataset.target[alpha1_index] * dataset.target[alpha2_index];
          double L, H;

          if(s > 0)
          {
            L = max(0.0, alpha2_value + alpha1_value - C);
            H = min(C, alpha2_value + alpha1_value);
          }

          else
          {
            L = max(0.0, alpha2_value - alpha1_value);
            H = min(C, C + alpha2_value - alpha1_value);
          }

          if(L < H)
          {
            #pragma omp cancellation point for

            if(valid_alpha1_found == 0)
            {
              double k11, k12, k22, eta;

              k11 = kernelFunction(dataset.data[alpha1_index], dataset.data[
    alpha1_index]);
              k12 = kernelFunction(dataset.data[alpha1_index], dataset.data[
    alpha2_index]);
              k22 = kernelFunction(dataset.data[alpha2_index], dataset.data[
    alpha2_index]);

              eta = k11 + k22 - 2*k12;
              if(eta > 0)
              {
                updated_alpha2_value = alpha2_value + dataset.target[alpha2_index
    ]*(errors[alpha1_index] - errors[alpha2_index])/eta;
```

```
      if(updated_alpha2_value < L) updated_alpha2_value = L;
      else if(updated_alpha2_value > H) updated_alpha2_value = H;
    }

    else
    {
      double f1, f2, L1, H1, Lobj, Hobj;
      f1 = dataset.target[alpha1_index]*(errors[alpha1_index]+b) -
alpha1_value*k11 - s*alpha2_value*k12;
      f2 = dataset.target[alpha2_index]*(errors[alpha2_index]+b) - s*
alpha1_value*k12 - alpha2_value*k22;

      L1 = alpha1_value + s*(alpha2_value - L);
      H1 = alpha1_value + s*(alpha2_value - H);

      Lobj = L1*f1 + L*f2 + (L1*L1*k11)/2 + (L*L*k22)/2 + s*L*L1*k12;
      Hobj = H1*f1 + H*f2 + (H1*H1*k11)/2 + (H*H*k22)/2 + s*H*H1*k12;

      if(Lobj < Hobj-eps)
        updated_alpha2_value = L;
      else if(Lobj > Hobj+eps)
        updated_alpha2_value = H;
      else
        updated_alpha2_value = alpha2_value;
    }

    if(fabs(updated_alpha2_value - alpha2_value) >= eps*(
updated_alpha2_value + alpha2_value + eps))
    {
      int thread_valid_alpha1_found = -1;

      #pragma omp critical
      {
        if(valid_alpha1_found == 0)
        {
          valid_alpha1_found = 1;
          thread_valid_alpha1_found = 1;

          updated_alpha1_value = alpha1_value + s*(alpha2_value -
updated_alpha2_value);

          // update the threshold value b
          if((updated_alpha1_value > 0) && (updated_alpha1_value < C))
            b += errors[alpha1_index] + dataset.target[alpha1_index]*(
updated_alpha1_value - alpha1_value)*k11 + dataset.target[alpha2_index]*(
updated_alpha2_value - alpha2_value)*k12;

          else if((updated_alpha2_value > 0) && (updated_alpha2_value <
C))
            b += errors[alpha2_index] + dataset.target[alpha1_index]*(
updated_alpha1_value - alpha1_value)*k12 + dataset.target[alpha2_index]*(
updated_alpha2_value - alpha2_value)*k22;

          else
            b += (errors[alpha1_index] + errors[alpha2_index])/2 +
dataset.target[alpha1_index]*(updated_alpha1_value - alpha1_value)*(k11+k12)/2
 + dataset.target[alpha2_index]*(updated_alpha2_value - alpha2_value)*(k12+k22
)/2;

          // update the error cache using new langrange multipliers
          double errors_delta_b, errors_delta_alpha1,
errors_delta_alpha2;

          errors_delta_b = old_b_value - b;
          errors_delta_alpha1 = (updated_alpha1_value - alpha1_value) *
dataset.target[alpha1_index];
          errors_delta_alpha2 = (updated_alpha2_value - alpha2_value) *
dataset.target[alpha2_index];

          int i;

          // #pragma omp parallel for default(none) private(i) shared(
dataset, errors, errors_delta_b, errors_delta_alpha1, errors_delta_alpha2,
kernelFunction)
          #pragma omp parallel for private(i)
          for(i=0; i<dataset.n_data; ++i)
```

```
                    {
                      errors[i] += errors_delta_b;
                      errors[i] += errors_delta_alpha1 * kernelFunction(dataset.
  data[alpha1_index], dataset.data[i]);
                      errors[i] += errors_delta_alpha2 * kernelFunction(dataset.
  data[alpha2_index], dataset.data[i]);
                    }

                    // store alpha values in alphas array
                    alphas[alpha1_index] = updated_alpha1_value;
                    alphas[alpha2_index] = updated_alpha2_value;
                  } // end of inner most if(valid_alpha1_found == 0)

                } // end of #pragma omp critical

                if(thread_valid_alpha1_found)
                {
                  #pragma omp cancel for
                }
              }

            } // end of inner if(valid_alpha1_found == 0)
          } // end of if(L < H)
        } // end of outter if(valid_alpha1_found == 0)
      }
    }
}


if(valid_alpha1_found)
  return 1;

// Parallely search across all alphas for alpha1 that makes positive step
valid_alpha1_found = 0;

// #pragma omp parallel for default(none) private(i, alpha1_index) shared(
  dataset, alphas, alpha2_index, C)
#pragma omp parallel
{
  #pragma omp for private(i, alpha1_index)
  for(i=0; i<dataset.n_data; ++i)
  {
    #pragma omp cancellation point for

    // alpha1_index = (i + random_offset) % dataset.n_data;
    alpha1_index = i;

    if((valid_alpha1_found == 0) && (alpha1_index != alpha2_index) && ((alphas[
  alpha1_index] == 0) || (alphas[alpha1_index] == C)) )
    {
      // SVC<T>::updateAlphaPair has been expanded to circumvent oprphaned
  cancellation point problem
      #pragma omp cancellation point for

      if(valid_alpha1_found == 0)
      {
        double alpha1_value, alpha2_value, updated_alpha1_value,
  updated_alpha2_value, old_b_value;
        alpha1_value = alphas[alpha1_index];
        alpha2_value = alphas[alpha2_index];
        old_b_value = b;

        int s = dataset.target[alpha1_index] * dataset.target[alpha2_index];
        double L, H;

        if(s > 0)
        {
          L = max(0.0, alpha2_value + alpha1_value - C);
          H = min(C, alpha2_value + alpha1_value);
        }

        else
        {
          L = max(0.0, alpha2_value - alpha1_value);
          H = min(C, C + alpha2_value - alpha1_value);
        }
```

```
    if(L < H)
    {
      #pragma omp cancellation point for

      if(valid_alpha1_found == 0)
      {
        double k11, k12, k22, eta;

        k11 = kernelFunction(dataset.data[alpha1_index], dataset.data[
alpha1_index]);
        k12 = kernelFunction(dataset.data[alpha1_index], dataset.data[
alpha2_index]);
        k22 = kernelFunction(dataset.data[alpha2_index], dataset.data[
alpha2_index]);

        eta = k11 + k22 - 2*k12;
        if(eta > 0)
        {
          updated_alpha2_value = alpha2_value + dataset.target[alpha2_index
]*(errors[alpha1_index] - errors[alpha2_index])/eta;
          if(updated_alpha2_value < L) updated_alpha2_value = L;
          else if(updated_alpha2_value > H) updated_alpha2_value = H;
        }

        else
        {
          double f1, f2, L1, H1, Lobj, Hobj;
          f1 = dataset.target[alpha1_index]*(errors[alpha1_index]+b) -
alpha1_value*k11 - s*alpha2_value*k12;
          f2 = dataset.target[alpha2_index]*(errors[alpha2_index]+b) - s*
alpha1_value*k12 - alpha2_value*k22;

          L1 = alpha1_value + s*(alpha2_value - L);
          H1 = alpha1_value + s*(alpha2_value - H);

          Lobj = L1*f1 + L*f2 + (L1*L1*k11)/2 + (L*L*k22)/2 + s*L*L1*k12;
          Hobj = H1*f1 + H*f2 + (H1*H1*k11)/2 + (H*H*k22)/2 + s*H*H1*k12;

          if(Lobj < Hobj-eps)
            updated_alpha2_value = L;
          else if(Lobj > Hobj+eps)
            updated_alpha2_value = H;
          else
            updated_alpha2_value = alpha2_value;
        }

        if(fabs(updated_alpha2_value - alpha2_value) >= eps*(
updated_alpha2_value + alpha2_value + eps))
        {
          int thread_valid_alpha1_found = -1;

          #pragma omp critical
          {
            if(valid_alpha1_found == 0)
            {
              valid_alpha1_found = 1;
              thread_valid_alpha1_found = 1;

              updated_alpha1_value = alpha1_value + s*(alpha2_value -
updated_alpha2_value);

              // update the threshold value b
              if((updated_alpha1_value > 0) && (updated_alpha1_value < C))
                b += errors[alpha1_index] + dataset.target[alpha1_index]*(
updated_alpha1_value - alpha1_value)*k11 + dataset.target[alpha2_index]*(
updated_alpha2_value - alpha2_value)*k12;

              else if((updated_alpha2_value > 0) && (updated_alpha2_value <
C))
                b += errors[alpha2_index] + dataset.target[alpha1_index]*(
updated_alpha1_value - alpha1_value)*k12 + dataset.target[alpha2_index]*(
updated_alpha2_value - alpha2_value)*k22;

              else
                b += (errors[alpha1_index] + errors[alpha2_index])/2 +
```

```
    dataset.target[alpha1_index]*(updated_alpha1_value - alpha1_value)*(k11+k12)/2
     + dataset.target[alpha2_index]*(updated_alpha2_value - alpha2_value)*(k12+k22
    )/2;

                    // update the error cache using new langrange multipliers
                    double errors_delta_b, errors_delta_alpha1,
    errors_delta_alpha2;

                    errors_delta_b = old_b_value - b;
                    errors_delta_alpha1 = (updated_alpha1_value - alpha1_value) *
    dataset.target[alpha1_index];
                    errors_delta_alpha2 = (updated_alpha2_value - alpha2_value) *
    dataset.target[alpha2_index];

                    int i;

                    // #pragma omp parallel for default(none) private(i) shared(
    dataset, errors, errors_delta_b, errors_delta_alpha1, errors_delta_alpha2,
    kernelFunction)
                    #pragma omp parallel for private(i)
                    for(i=0; i<dataset.n_data; ++i)
                    {
                      errors[i] += errors_delta_b;
                      errors[i] += errors_delta_alpha1 * kernelFunction(dataset.
    data[alpha1_index], dataset.data[i]);
                      errors[i] += errors_delta_alpha2 * kernelFunction(dataset.
    data[alpha2_index], dataset.data[i]);
                    }

                    // store alpha values in alphas array
                    alphas[alpha1_index] = updated_alpha1_value;
                    alphas[alpha2_index] = updated_alpha2_value;

                  } // end of inner most if(valid_alpha1_found == 0)

                } // end of #pragma omp critical

                if(thread_valid_alpha1_found)
                {
                  #pragma omp cancel for
                }
              }

            } // end of inner if(valid_alpha1_found == 0)
          } // end of if(L < H)
        } // end of outter if(valid_alpha1_found == 0)
      }
    }
  }


  if(valid_alpha1_found)
    return 1;

  return 0;
}

template <typename T>
int SVC<T>::updateAlphaPair(int alpha1_index, int alpha2_index)
{
  double alpha1_value, alpha2_value, updated_alpha1_value, updated_alpha2_value,
    old_b_value;
  alpha1_value = alphas[alpha1_index];
  alpha2_value = alphas[alpha2_index];
  old_b_value = b;

  int s = dataset.target[alpha1_index] * dataset.target[alpha2_index];
  double L, H;

  if(s > 0)
  {
    L = max(0.0, alpha2_value + alpha1_value - C);
    H = min(C, alpha2_value + alpha1_value);
  }

  else
```

```
{
  L = max(0.0, alpha2_value - alpha1_value);
  H = min(C, C + alpha2_value - alpha1_value);
}

if(L == H)
  return 0;

double k11, k12, k22, eta;

k11 = kernelFunction(dataset.data[alpha1_index], dataset.data[alpha1_index]);
k12 = kernelFunction(dataset.data[alpha1_index], dataset.data[alpha2_index]);
k22 = kernelFunction(dataset.data[alpha2_index], dataset.data[alpha2_index]);

eta = k11 + k22 - 2*k12;
if(eta > 0)
{
  updated_alpha2_value = alpha2_value + dataset.target[alpha2_index]*(errors[
  alpha1_index] - errors[alpha2_index])/eta;
  if(updated_alpha2_value < L) updated_alpha2_value = L;
  else if(updated_alpha2_value > H) updated_alpha2_value = H;
}

else
{
  double f1, f2, L1, H1, Lobj, Hobj;
  f1 = dataset.target[alpha1_index]*(errors[alpha1_index]+b) - alpha1_value*k11
  - s*alpha2_value*k12;
  f2 = dataset.target[alpha2_index]*(errors[alpha2_index]+b) - s*alpha1_value*
  k12 - alpha2_value*k22;

  L1 = alpha1_value + s*(alpha2_value - L);
  H1 = alpha1_value + s*(alpha2_value - H);

  Lobj = L1*f1 + L*f2 + (L1*L1*k11)/2 + (L*L*k22)/2 + s*L*L1*k12;
  Hobj = H1*f1 + H*f2 + (H1*H1*k11)/2 + (H*H*k22)/2 + s*H*H1*k12;

  if(Lobj < Hobj-eps)
    updated_alpha2_value = L;
  else if(Lobj > Hobj+eps)
    updated_alpha2_value = H;
  else
    updated_alpha2_value = alpha2_value;
}

if(fabs(updated_alpha2_value - alpha2_value) < eps*(updated_alpha2_value +
  alpha2_value + eps))
  return 0;


updated_alpha1_value = alpha1_value + s*(alpha2_value - updated_alpha2_value);

// update the threshold value b
if((updated_alpha1_value > 0) && (updated_alpha1_value < C))
  b += errors[alpha1_index] + dataset.target[alpha1_index]*(updated_alpha1_value
   - alpha1_value)*k11 + dataset.target[alpha2_index]*(updated_alpha2_value -
  alpha2_value)*k12;

else if((updated_alpha2_value > 0) && (updated_alpha2_value < C))
  b += errors[alpha2_index] + dataset.target[alpha1_index]*(updated_alpha1_value
   - alpha1_value)*k12 + dataset.target[alpha2_index]*(updated_alpha2_value -
  alpha2_value)*k22;

else
  b += (errors[alpha1_index] + errors[alpha2_index])/2 + dataset.target[
  alpha1_index]*(updated_alpha1_value - alpha1_value)*(k11+k12)/2 + dataset.
  target[alpha2_index]*(updated_alpha2_value - alpha2_value)*(k12+k22)/2;

// update the error cache using new langrange multipliers
double errors_delta_b, errors_delta_alpha1, errors_delta_alpha2;

errors_delta_b = old_b_value - b;
errors_delta_alpha1 = (updated_alpha1_value - alpha1_value) * dataset.target[
  alpha1_index];
errors_delta_alpha2 = (updated_alpha2_value - alpha2_value) * dataset.target[
  alpha2_index];
```

```
  int i;

  // #pragma omp parallel for default(none) private(i) shared(dataset, errors,
    errors_delta_b, errors_delta_alpha1, errors_delta_alpha2, kernelFunction)
  #pragma omp parallel for private(i)
  for(i=0; i<dataset.n_data; ++i)
  {
    errors[i] += errors_delta_b;
    errors[i] += errors_delta_alpha1 * kernelFunction(dataset.data[alpha1_index],
    dataset.data[i]);
    errors[i] += errors_delta_alpha2 * kernelFunction(dataset.data[alpha2_index],
    dataset.data[i]);
  }

  // store alpha values in alphas array
  alphas[alpha1_index] = updated_alpha1_value;
  alphas[alpha2_index] = updated_alpha2_value;

  return 1;
}
/* SVM Classifier: Methods to initialize parameters, fit model and predict labels
    -- END */

// explicit instantiation of template class and function
template class SVC<int>;
template class SVC<float>;
template class SVC<double>;
```

src/kernels.cpp

```
#include "kernels.hpp"

/* All Kernels assume that both vectors have same length */


// Perform the for product of two feature vectors
template <typename T>
double dotProduct(vector<T> feature_vector1, vector<T> feature_vector2)
{
  int i;

  double dot_product = 0;

  // #pragma omp parallel for private(i) reduction(+:dot_product)
  for(i=0; i<(int)feature_vector1.size(); ++i)
    dot_product += feature_vector1[i] * feature_vector2[i];

  return dot_product;
}

template double dotProduct(vector<int>, vector<int>);
template double dotProduct(vector<float>, vector<float>);
template double dotProduct(vector<double>, vector<double>);
```

src/validation.cpp

```
#include "validation.hpp"

#include <cassert>

// Assumption: Number of elements in target and predictions are the same
double computeAccuracy(const vector<int>& target, const vector<int>& predictions)
{
  // Assert that the target and prediction vector have the same size
  assert(target.size() == predictions.size());

  int n_correct_preds = 0;

  int i;

  #pragma omp parallel for private(i) reduction(+:n_correct_preds)
  for(i=0; i<(int)target.size(); ++i)
    if(target[i] == predictions[i])
      n_correct_preds++;
```

```
  double accuracy = (double)n_correct_preds/target.size();
  return accuracy;
}


// Assumption: Number of elements in target and predictions are the same
double computePrecision(const vector<int>& target, const vector<int>& predictions)
{
  // Assert that the target and prediction vector have the same size
  assert(target.size() == predictions.size());

  int i;
  int n_pred_positive = 0, n_true_positive = 0;

  #pragma omp parallel for private(i) reduction(+:n_pred_positive, n_true_positive
    )
  for(i=0; i<(int)target.size(); ++i)
    if(predictions[i] > 0)
    {
      n_pred_positive ++;
      if(target[i] > 0)
        n_true_positive ++;
    }

  double precision = (double)n_true_positive/n_pred_positive;
  return precision;
}

// Assumption: Number of elements in target and predictions are the same
double computeRecall(const vector<int>& target, const vector<int>& predictions)
{
  // Assert that the target and prediction vector have the same size
  assert(target.size() == predictions.size());

  int i;
  int n_target_positive = 0, n_true_positive = 0;

  for(i=0; i<(int)target.size(); ++i)
    if(target[i] > 0)
    {
      n_target_positive ++;
      if(predictions[i] > 0)
        n_true_positive ++;
    }

  double recall = (double)n_true_positive/n_target_positive;
  return recall;
}

double computeF1Score(const vector<int>& target, const vector<int>& predictions)
{
  // Assert that the target and prediction vector have the same size
  assert(target.size() == predictions.size());

  double precision, recall, f1_score;

  precision = computePrecision(target, predictions);
  recall = computeRecall(target, predictions);
  f1_score = 2*precision*recall/(precision + recall);

  return f1_score;
}
```

test/test1.cpp

```
#include <iostream>
using namespace std;

#include "dataset.hpp"
#include "classifier.hpp"
#include "kernels.hpp"
#include "validation.hpp"

#include <omp.h>
```

```
int main(int argc, char *argv[])
{
  // Read training dataset from file
  Dataset<int> training_dataset = readLibsvmDataset<int>("./datasets/a1a", −1,
    123);

  if(training_dataset.data.size() == 0)
  {
    cout << "ERROR: Failed to read training dataset from file" << endl;
    return 1;
  }

  // Create the SVM Classifier with parameters ana fit the data
  double C, tol, eps;
  C = 1.0;
  tol = 0.001;
  eps = 0.001;

  SVC<int> classifier(C, tol, eps, dotProduct);

  double start_time, end_time;

  start_time = omp_get_wtime();
  classifier.fit(training_dataset);
  end_time = omp_get_wtime();

  cout << "Fitted SVM model with training data in " << (end_time − start_time) <<
    " seconds." << endl;

  // Read the testing dataset from file
  Dataset<int> testing_dataset = readLibsvmDataset<int>("./datasets/a1a.t", −1,
    123);

  if(testing_dataset.data.size() == 0)
  {
    cout << "ERROR: Failed to read testing dataset from file" << endl;
    return 1;
  }

  start_time = omp_get_wtime();
  vector<int> predictions = classifier.predict(testing_dataset);
  end_time = omp_get_wtime();

  cout << "Predicted class labels for testing data in " << (end_time − start_time)
     << " seconds." << endl;

  // Compute the accuracy, precision, recall and f1−score of the classifier
  double accuracy, precision, recall, f1_score;
  accuracy = computeAccuracy(testing_dataset.target, predictions);
  precision = computePrecision(testing_dataset.target, predictions);
  recall = computeRecall(testing_dataset.target, predictions);
  f1_score = computeF1Score(testing_dataset.target, predictions);

  cout << "Accuracy = " << accuracy << endl;
  cout << "Precision = " << precision << endl;
  cout << "Recall = " << recall << endl;
  cout << "F1−Score = " << f1_score << endl;

  return 0;
}
```