DATA MINING

ASSIGNMENT 3 REPORT

C Rajmohan SR No. : 10164

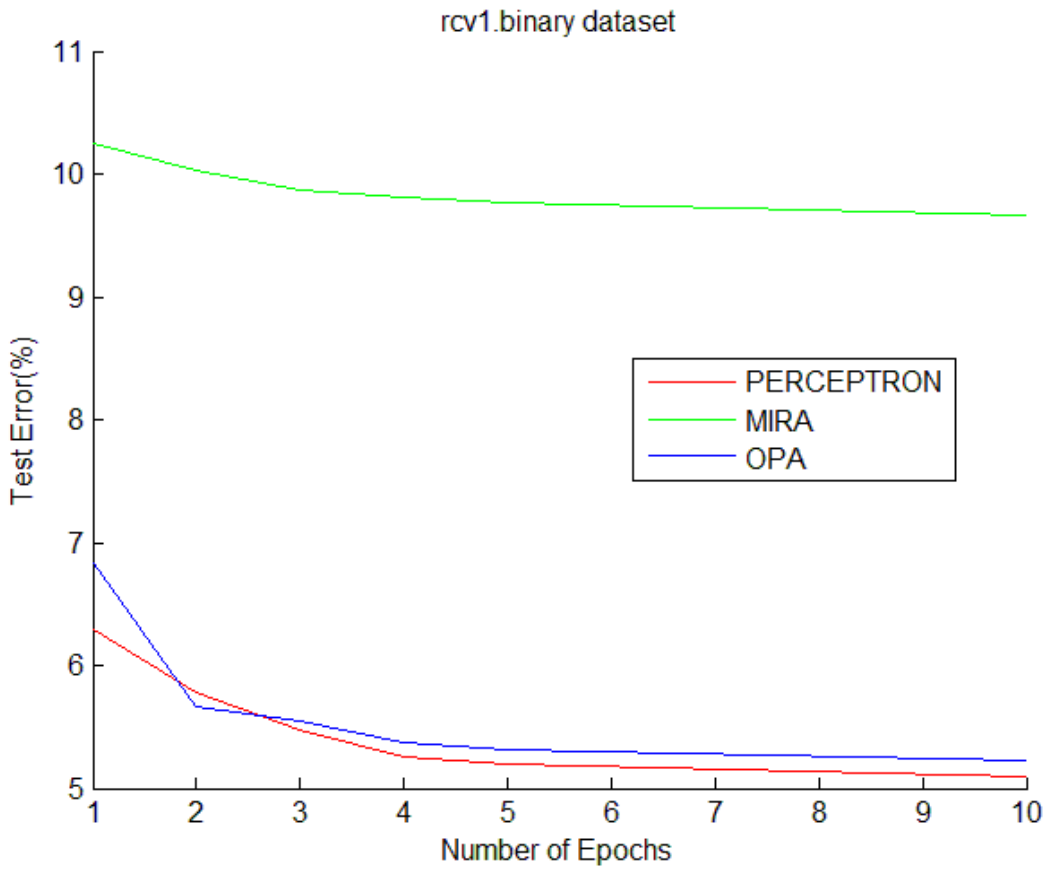
Question 1:

Compare the Performance of Perceptron based Binary Classification Algorithms

Rcv1.binary dataset:

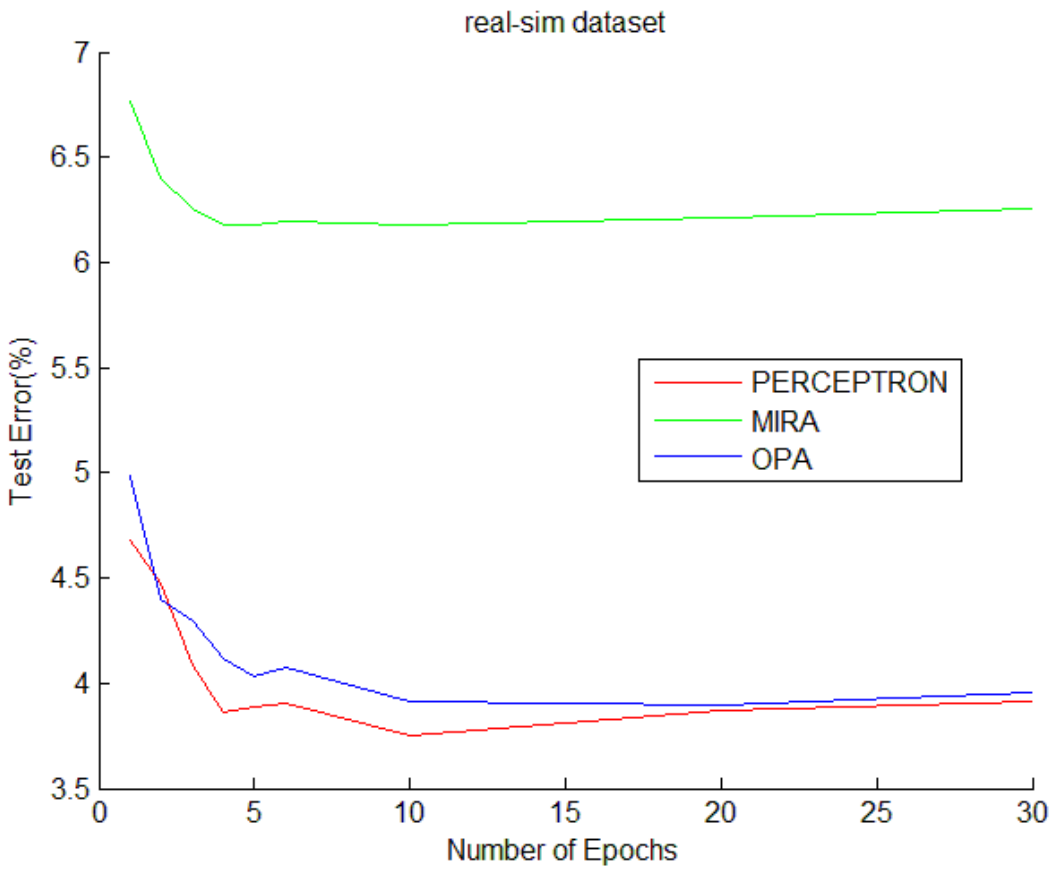
of classes: 2 # of features: 47,236

# of Training data samples: 20,242 # of Test data samples : 677,399



Real-Sim Dataset:

of classes: 2
# of data: 72,309 (training : 57847, test: 14462)
# of features: 20,958



Algorithm and Source code: Refer to Appendix A

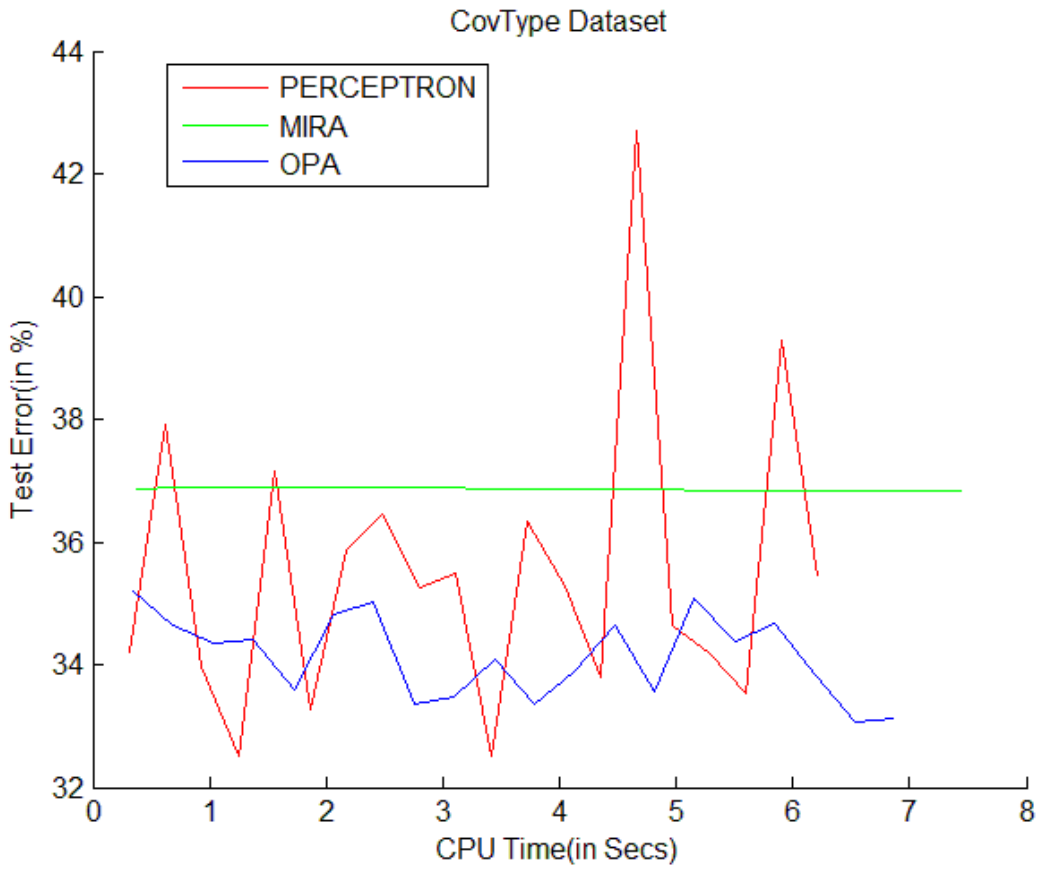
Question 2:

Parallel Implementation of Perceptron Algorithms on CovType dataset

of classes: 2
# of data: 581,012 (training : 80%, testing : 20%)
# of features: 54

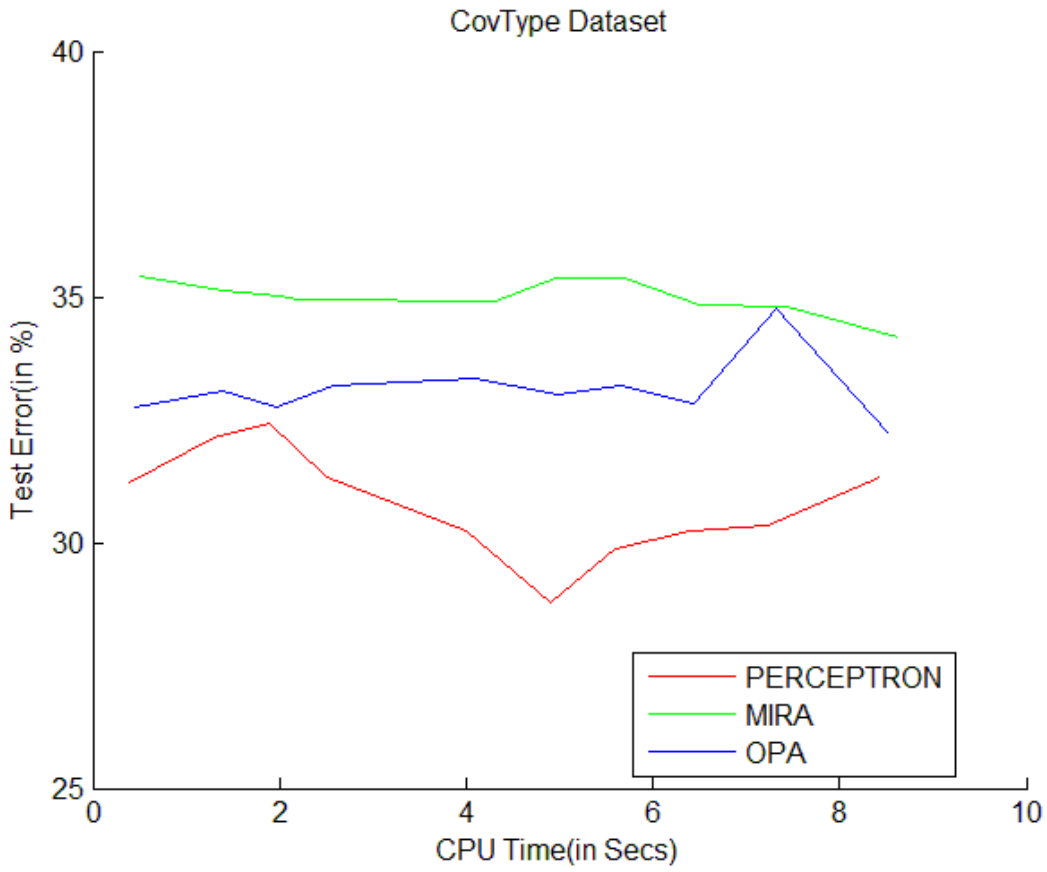
covtype.libsvm.binary.bz2
covtype.libsvm.binary.scale.bz2 (scaled to [0, 1])

Test Error vs. CPU Time(Training)



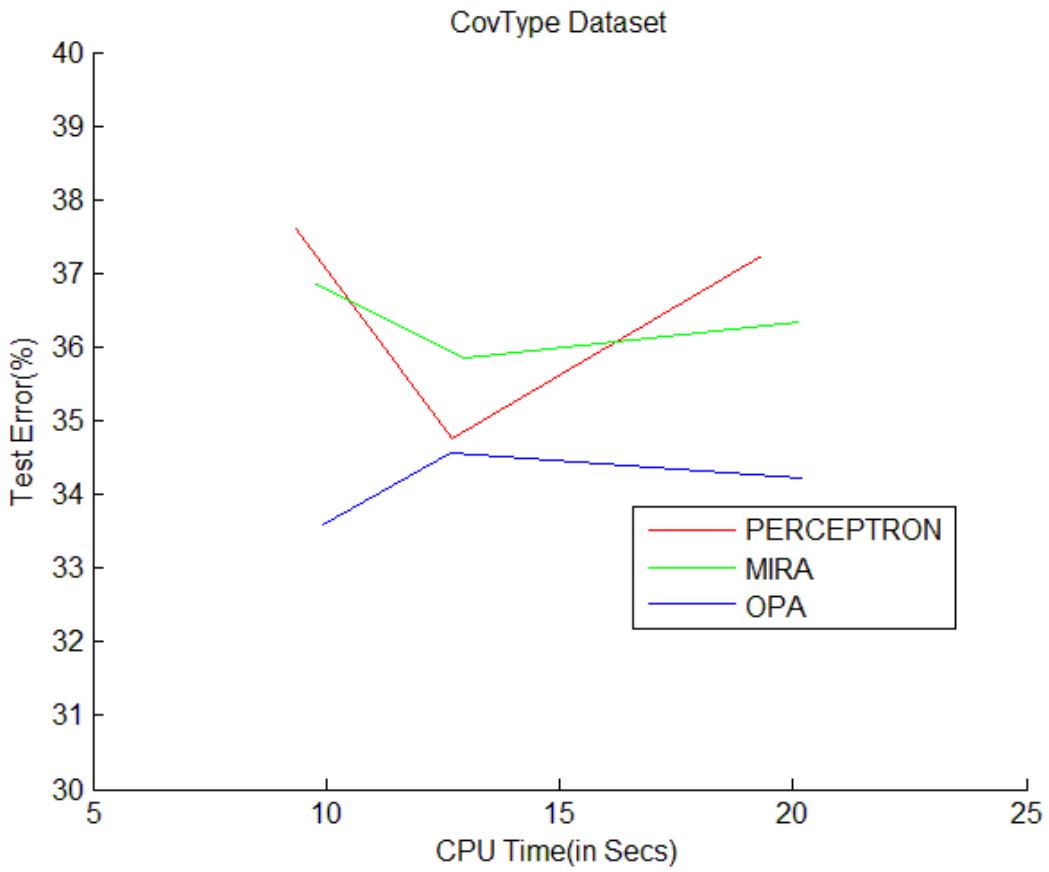
Test Error vs CPU Training time:

- Run on 4 cluster machines.
- Weight vector W is shared among processors at the end of each epoch and average of all the weight vectors is used for next epoch



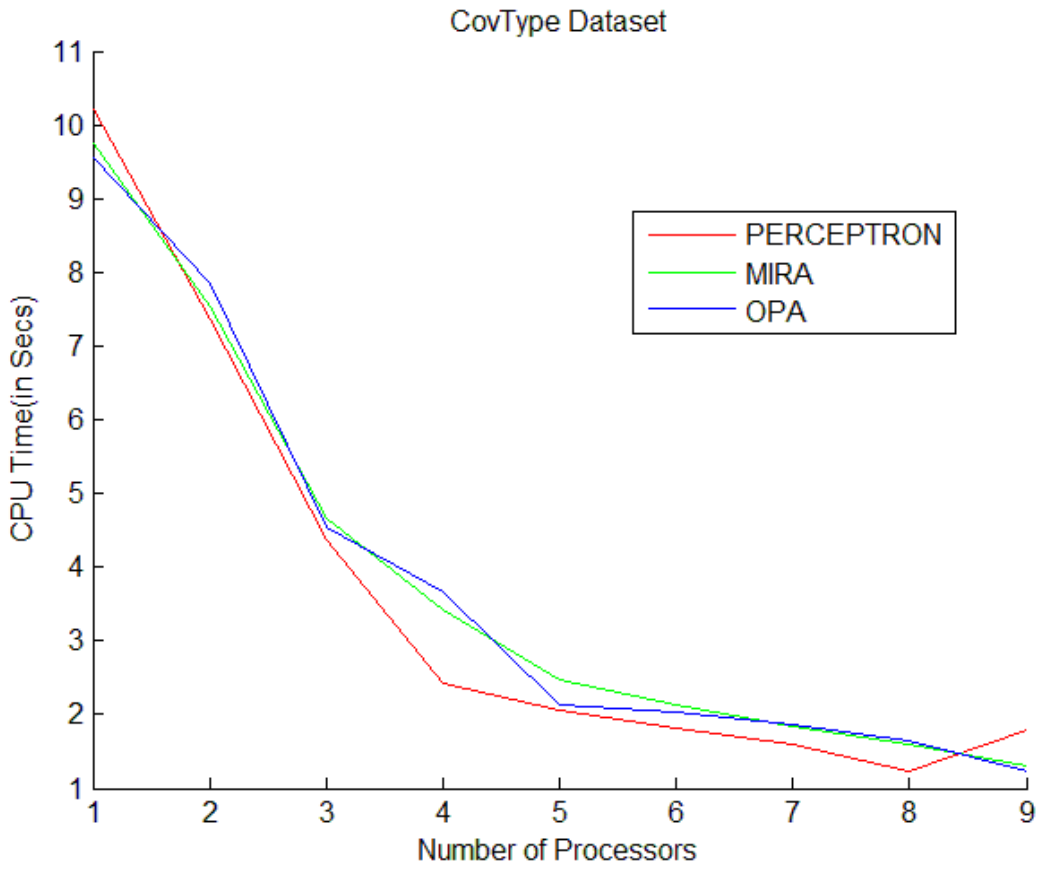
Test error vs. CPU Training time

- W Shared at the end of ‘T’ epochs
- T = 10, 20, 30



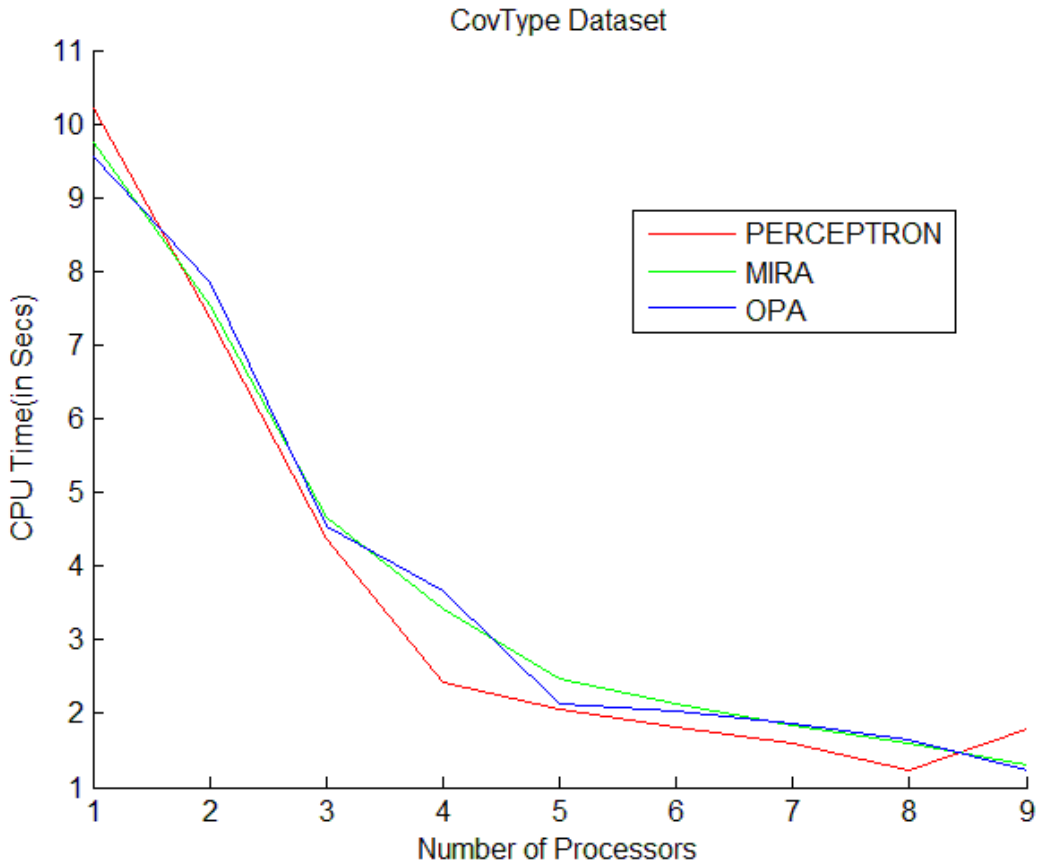
No of processors vs. CPU training time

- Training time includes communication cost
- Weight vector is shared at the end of each epoch among nodes.
- Note that since MPI_Barrier is used to synchronize nodes at the end of each epoch, so that new weight vector can be shared among them. Hence training time is affected by any node which is slowest among all.



No of processors vs. CPU training time

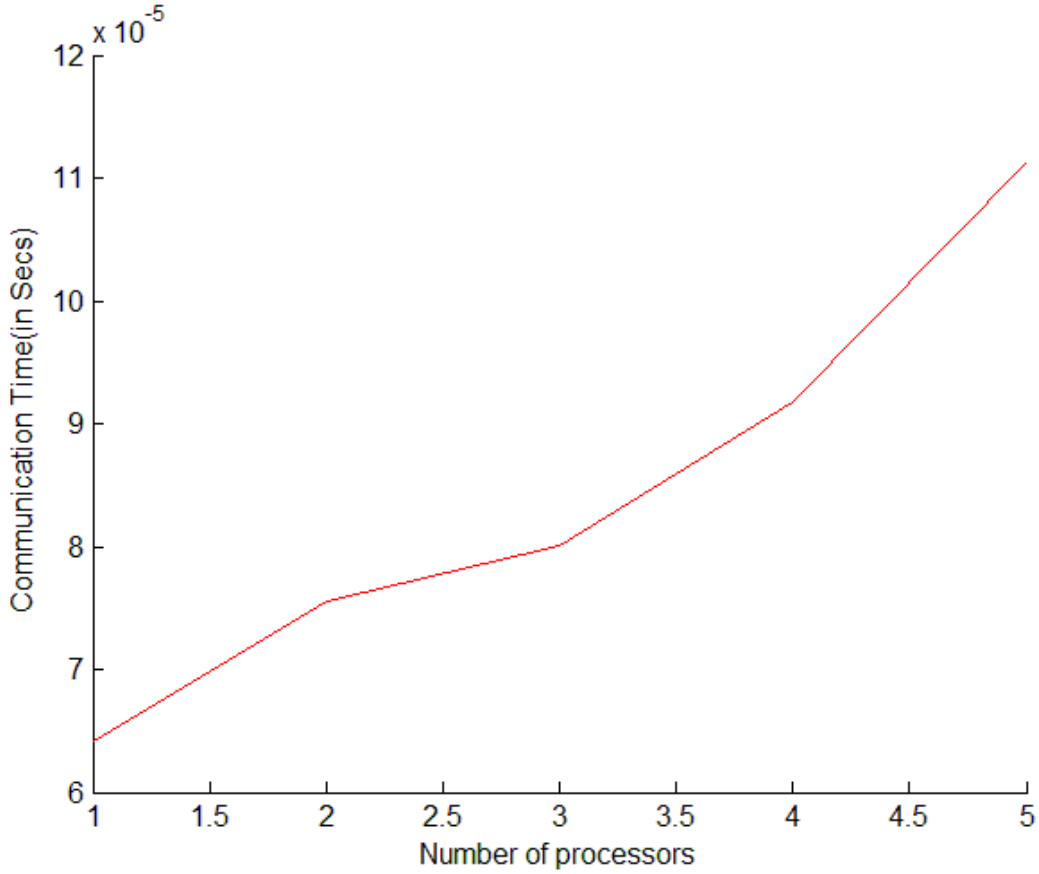
- Weight vector is only shared among nodes at end of ‘T’ epoch which is then used for testing.



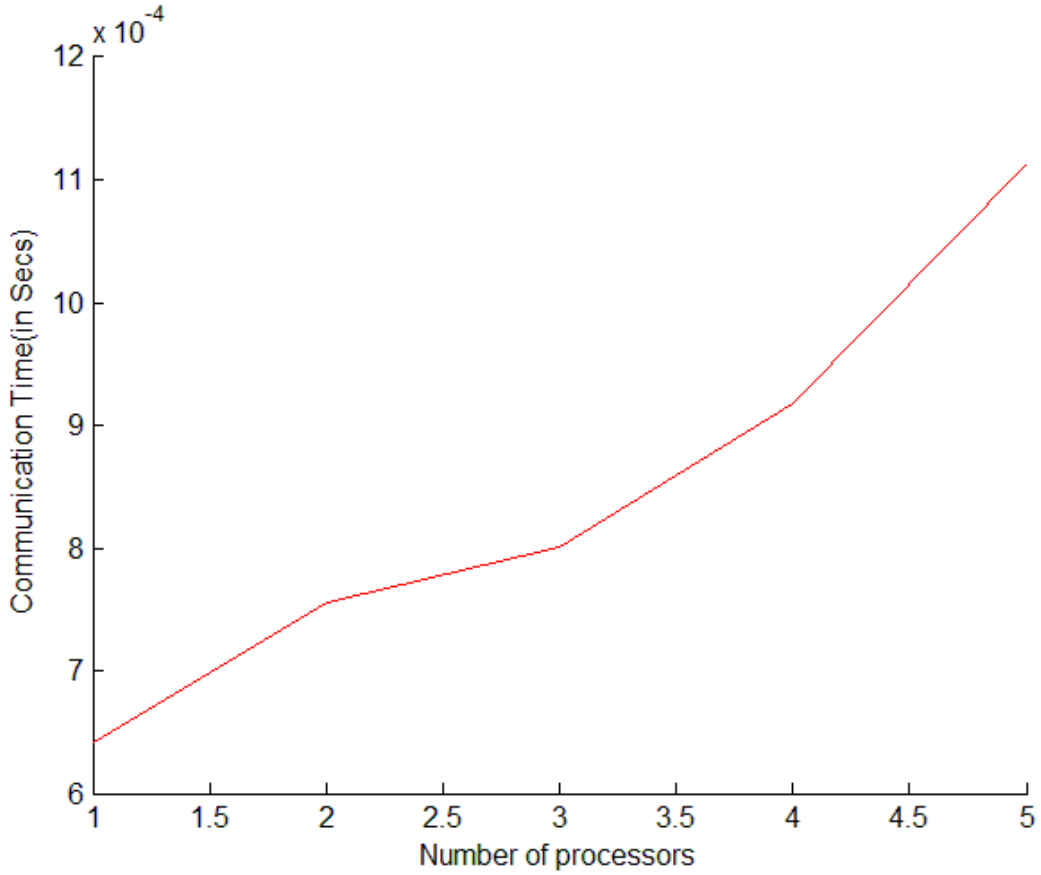
Communication time

- After each epoch (or at the end of 'T' epochs) all nodes except node 0 send their weight vector to node 0 (master). Node 0 computes average of all the weight vectors received and its own weight vector. Then node 0 sends it across to all nodes. All nodes receive this updated weighted vector and use it for next epoch.
- Weight vector is packed and send via MPI functions (656 bytes).

Fig 1 showing communication time when W is sent at end of T epochs(4 nodes).



- communication time when W is shared at the end of each epoch for 10 epochs.



Question 3:

Implement Dual Co-ordinate descent method for low rank linearization approach with different sizes of |Z| chosen

i) Randomly
ii) K-Means

on IJCNN dataset.
Compare with libsvm results

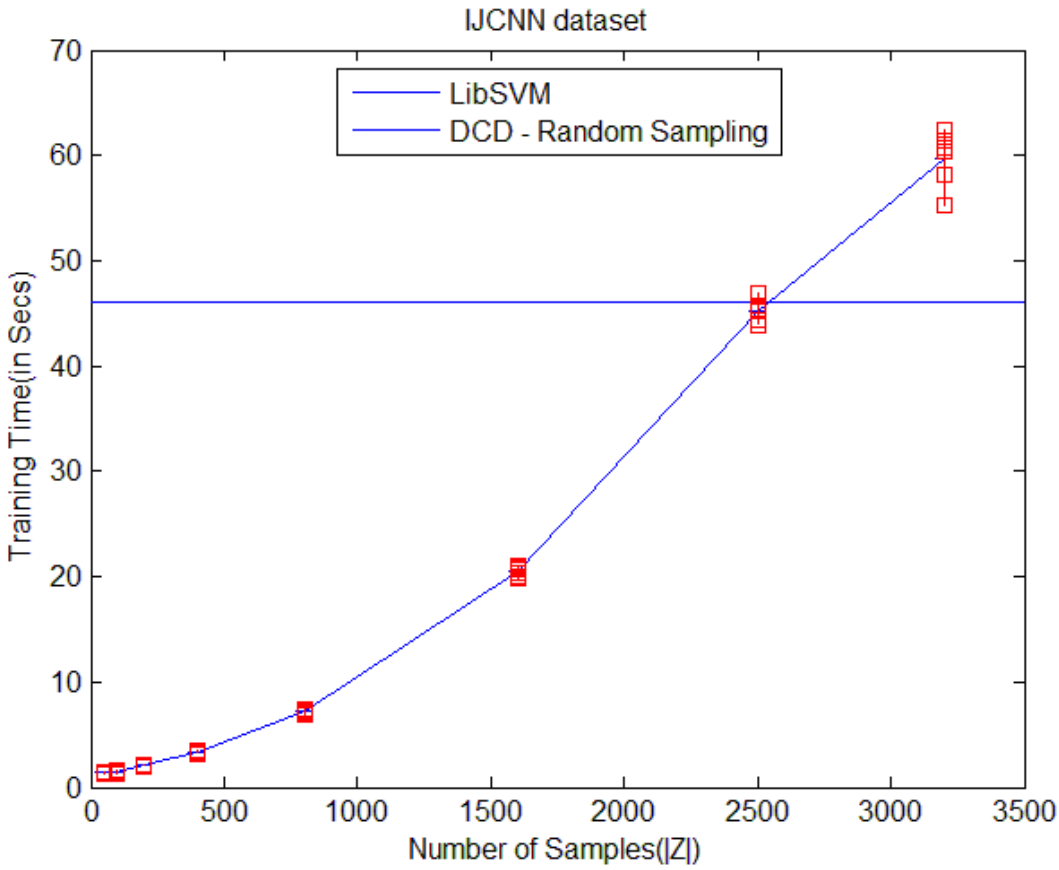
IJCNN1 Dataset:

of classes: 2
# of data: train: 49,990 , test: 91,701
# of features: 22

Let Z denote samples from Xr and |Z| denote number of samples.

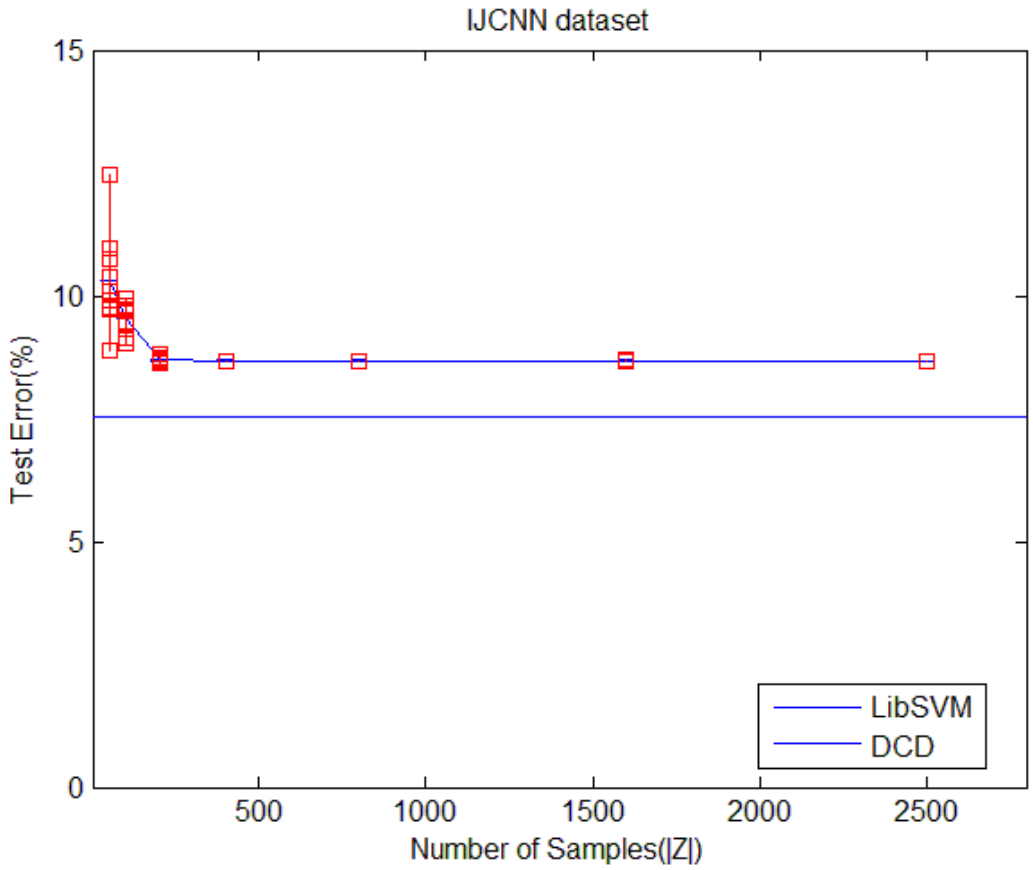
|Z| chosen randomly and Polynomial kernel function

- Polynomial kernel function (gamma*u'*v + coef0)^degree) is used where
- Parameters: gamma = 1/|features| , coef0 = 0, degree = 2.
- C = 10



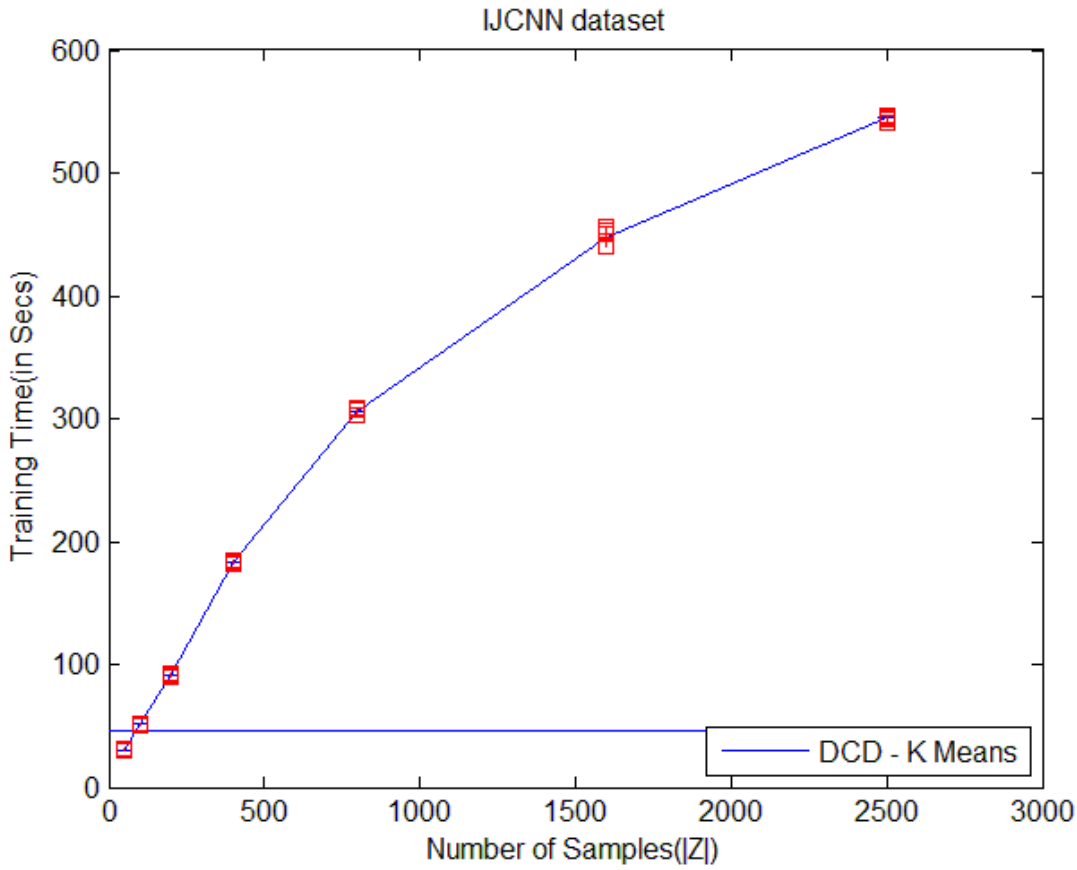
Training Time vs. Number of Samples

|Z| chosen randomly and polynomial Kernel function



Number of Samples vs. Test Error

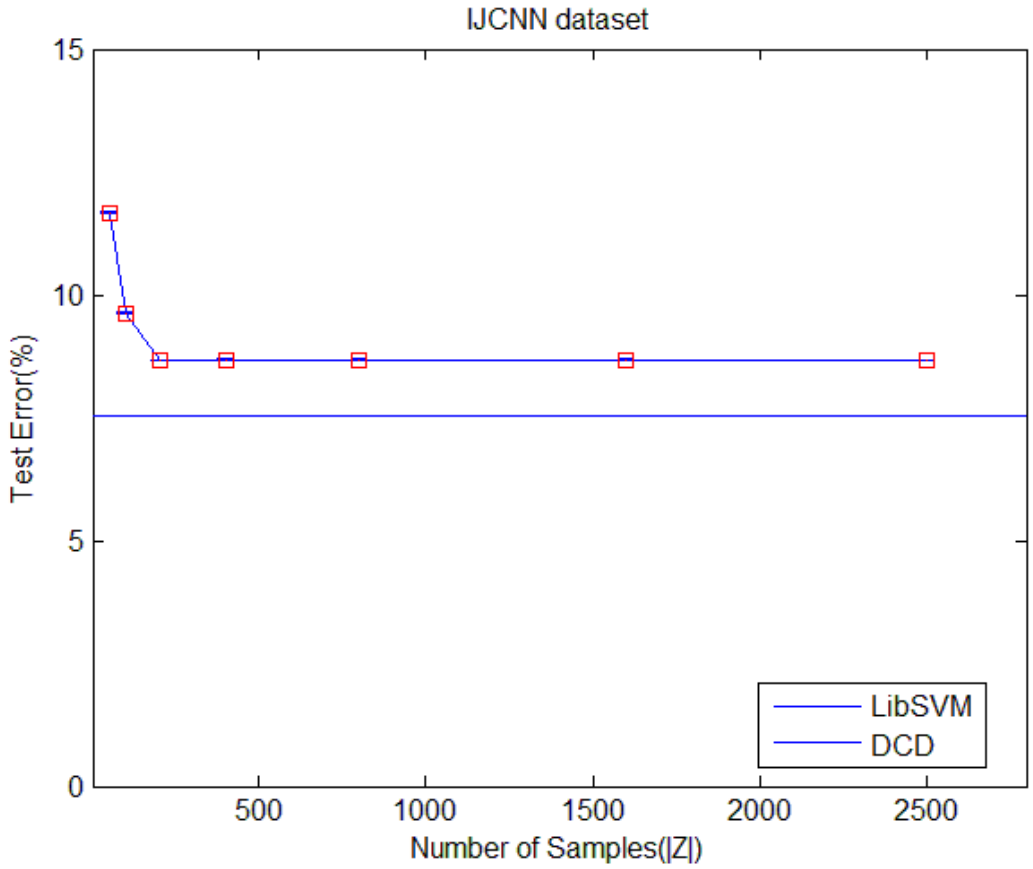
|Z| chosen by k-means and Polynomial kernel function



Number of Samples vs Training time

(note: K-means Algorithm time dominates)

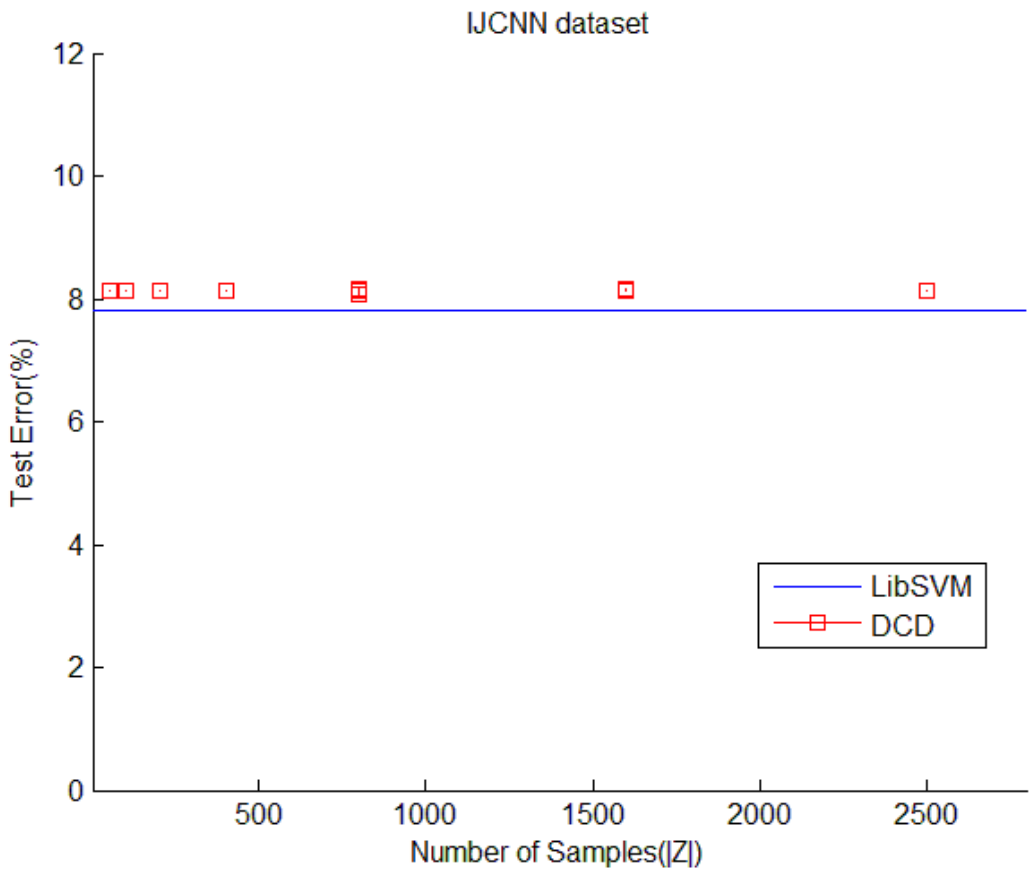
|Z| chosen by k-means and Polynomial kernel function



Number of Samples vs. Test Error (%)

|Z| chosen randomly and linear kernel function

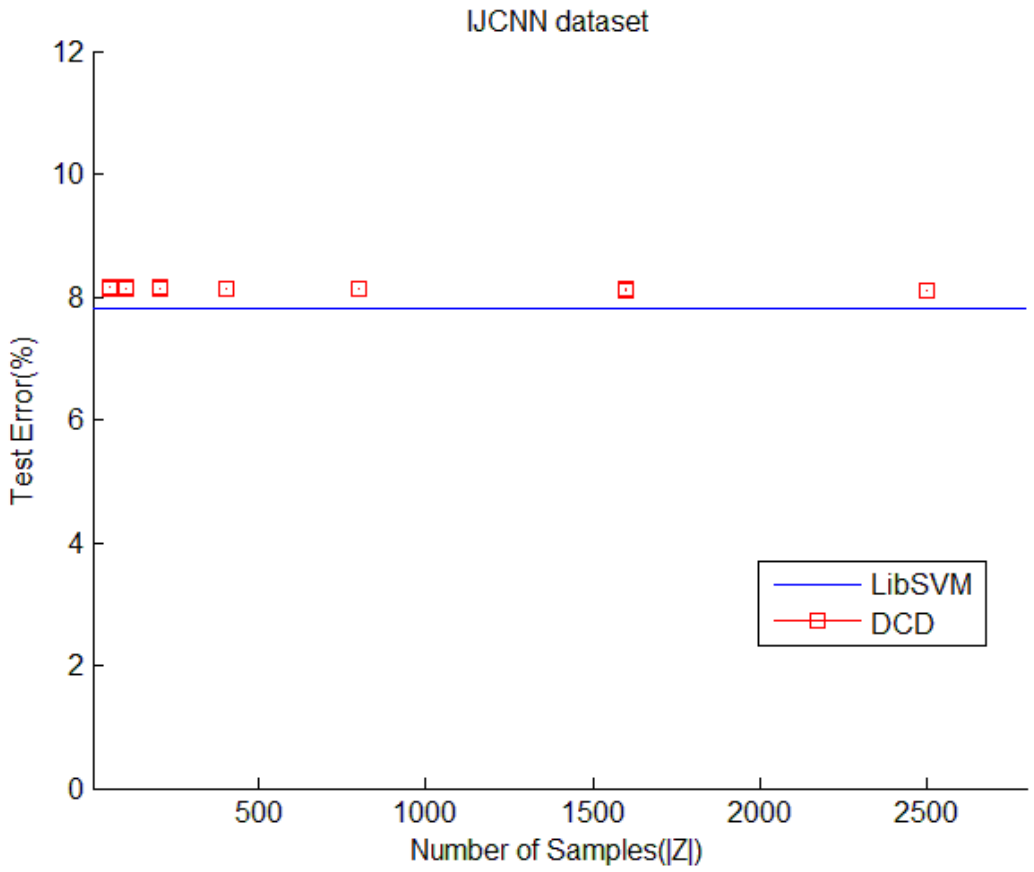
- Linear Kernel function : u'*v
- C = 10



Number of Samples vs. Test Error(%)

|Z| chosen by K-Means and linear kernel function

- Linear Kernel function : u'*v
- C = 10



Number of Samples vs. Test Error(%)

Source code: See appendix part –c.

Appendix: Part-A

perceptron.m

```
% Perceptron Learning Algorithm for Binary Classification

n = size(trainfeatures,1); % no of samples
d = size(trainfeatures,2); % dimension of each sample
max_epoch = 50;
w = zeros(d,1);

perceptron_w = zeros(d,max_epoch);
mistakes_per = zeros(max_epoch,1);
flag = 1;
epoch = 0;

s = rng; % save seed

% randomly permute input samples
rng(s);
perm = randperm(n);
trainfeatures_transpose = trainfeatures';

while(flag == 1)

    flag = 0;
    epoch = epoch + 1;

    for j = 1: n

        i = perm(j);
        actual_label = trainlabels(i,1);
        vect_x = trainfeatures_transpose(:,i);

        wt_x = w' * vect_x;

        if wt_x >= 0
            yicap = 1;
        else
            yicap = -1;
        end

        if (actual_label * yicap) < 0        % if error
            w = w + (actual_label * vect_x);
            flag = 1;
            mistakes_per(epoch) = mistakes_per(epoch) + 1;
        end

    end


perceptron_w(:,epoch) = w;
    if(epoch == max_epoch)
        break;
    end
end
```

mira.m

```
n = size(trainfeatures,1); % no of samples
d = size(trainfeatures,2); % dimension of each sample
max_epoch = 50;
w = rand(d,1);  % w ~= 0
mira_w = zeros(d,max_epoch);
mistakes_mira = zeros(max_epoch,1);
flag = 1;
epoch = 0;
tow = 0;

% randomly permute input samples
rng(s);
perm = randperm(n);
trainfeatures_trasnpose = trainfeatures';

while(flag == 1)

    flag = 0;
    epoch = epoch + 1;
    for j = 1: n

        i = perm(j);
        actual_label = trainlabels(i,1);
        vect_x = trainfeatures_trasnpose(:,i);
        wt_x = w' * vect_x;
        tow = 0;
        % find sign of wt . x
        if wt_x >= 0
            yicap = 1;
        else
            yicap = -1;
        end

        % calculating tow
        yi_wt_xi = actual_label * wt_x;

        if(yi_wt_xi >= 0)
            tow = 0;
        else
            norm_xisqr = (sum(vect_x .^ 2));
            if (norm_xisqr ~= 0)
            temp = yi_wt_xi/norm_xisqr;
            if(temp <= -1)
                tow = 1;
            else
                tow = -temp;
            end
            end
        end

        % updating weight vector
        if (actual_label * yicap) < 0        % if error
            w = w + ((tow * actual_label) * vect_x);
            flag = 1;
            mistakes_mira(epoch) = mistakes_mira(epoch) + 1;
        end
```

```
    end

    mira_w(:,epoch) = w;
    if(epoch == max_epoch)
        break;
    end
end
```

opa.m

```
n = size(trainfeatures,1); % no of samples
d = size(trainfeatures,2); % dimension of each sample
max_epoch = 50;
w = zeros(d,1);  % w ~= 0
opa_w = zeros(d,max_epoch);

flag = 1;
epoch = 0;

rng(s);
perm = randperm(n);
mistakes_opa = zeros(max_epoch,1);
trainfeatures_transpose = trainfeatures';

while(flag == 1)

    flag = 0;
    epoch = epoch + 1;
    for j = 1: n

        i = perm(j);
        tow = 0;
        actual_label = trainlabels(i,1);
        vect_x = trainfeatures_transpose(:,i);
        wt_x = w' * vect_x;

        % find sign of wt . x
        if wt_x >= 0
            yicap = 1;
        else
            yicap = -1;
        end

        % calculating tow
        m_yi_wt_xi = 1 - (actual_label * wt_x);
        norm_xisqr = (sum(vect_x .^ 2));
        if(norm_xisqr ~= 0)
            tow = max(0,m_yi_wt_xi) / norm_xisqr;
        end
        % updating weight vector
        if (actual_label * yicap) < 0        % if error
            w = w + ((tow * actual_label) * trainfeatures_transpose(:,i));
            flag = 1;
            mistakes_opa(epoch) = mistakes_opa(epoch) + 1;
        end
    end
    opa_w(:,epoch) = w;
    if(epoch == max_epoch)
        break;
    end
```

```
end
```

test.m

```
n = size(testfeatures,1); % no of samples
d = size(testfeatures,2);% dimension of each sample

noof_iter = 50; % no of epochs of training (for each w check accuracy)
predictedlabels = int8(ones(n,1));
test_mistakes = zeros(noof_iter,1);
test_error = zeros(noof_iter,1);

testfeatures_transpose = testfeatures';
clear testfeatures;

for iter = 1 : noof_iter
    w = big_w(:,iter);
    for i = 1: n
        actual_label = testlabels(i,1);
        vect_x = testfeatures_transpose(:,i);

        wt_x = w' * vect_x;

        if wt_x <0
            predictedlabels(i,1) = -1;
        else
            predictedlabels(i,1) = 1;
        end
    end

    test_error(iter,1) = 100 - (sum(testlabels == predictedlabels)/n)*100;

end
```

Appendix: Part – B

Parallel perceptron algorithms:

Main.cc (Since code is large, only important code snippets are given here. For complete code kindly refer to attached cc files with mail)

```
// structure to handle properties a feature like id and weight
struct Feature {
  int id;
  double weight;
};

// structure that stores the properties of a document sample, including its document
// id class label, the square of its two norm and all its features.
struct Sample {
  int id;
  int label;
  double two_norm_sq;
  vector<Feature> features;
};

// Dedicated structure which stores the weight vector
struct AVector {
vector<Feature> features;
};

// training samples
vector<Sample> train_samples_;

// test samples
vector<Sample> test_samples_;

AVector weight_vector;
```

Reading Training data:

```
bool read_train_data(void)
{
       int num_processors, myid;
       string train_file_path =
"/home/rajmohan.c/project/perceptron/datasets/covtype/covtype.libsvm.binary.scale/covt
ype.libsvm_train.binary.scale";

       MPI_Comm_size(MPI_COMM_WORLD, &num_processors); // no of processors
       MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id

       // Reading from DATA FILE
       cout << myid << "Reading file : " << train_file_path  << endl;

       const char *filename = train_file_path.c_str();

       File* file = File::Open(filename, "r");
       if (file == NULL)
       {
           cerr << "Cannot find file " << filename << endl;
           MPI_Finalize();
           return 0;
       }
```

```
string line;
  int num_local_pos = 0;
  int num_local_neg = 0;

  while (file->ReadLine(&line))
  {
    // If the sample should be assigned to this processor
    if (num_total_ % num_processors == myid)
    {
      int label = 0;
      const char* start = line.c_str();
      // Extracts the sample's class label
      if (!SplitOneIntToken(&start, " ", &label))
      {
        cerr << "Error parsing line: " << num_total_ + 1 << endl;
        return false;
      }

      // Gets the local number of positive and negative samples
      if (label == 1)
      {
        ++num_local_neg;
      }
      else if (label == 2)
      {
        ++num_local_pos;
      } else
      {
        cerr << "Unknow label in line: " << num_total_ + 1 << label;
        return false;
      }

      // Creates a "Sample" and add to the end of samples_
      train_samples_.push_back(Sample());
      Sample& sample = train_samples_.back();
      sample.label = label;
      sample.id = num_total_;  // Currently num_total_ == sample id
      sample.two_norm_sq = 0.0;

      // Extracts the sample's features
      vector<pair<string, string> > kv_pairs;
      SplitStringIntoKeyValuePairs(string(start), ":", " ", &kv_pairs);
      vector<pair<string, string> >::const_iterator pair_iter;

       for (pair_iter = kv_pairs.begin(); pair_iter !=
kv_pairs.end();++pair_iter)
      {
        Feature feature;
        feature.id = atoi(pair_iter->first.c_str());
        feature.weight = atof(pair_iter->second.c_str());
        sample.features.push_back(feature);
        sample.two_norm_sq += (feature.weight * feature.weight);
      }
    }
    ++num_total_;
  }
  file->Close();
  delete file;

  // Get the global number of positive and negative samples
  int local[2];
  int global[2];
```

```
          local[0] = num_local_pos;
          local[1] = num_local_neg;
          memset(global, 0, sizeof(global[0] * 2));
          MPI_Barrier(MPI_COMM_WORLD);
          MPI_Allreduce(local, global, 2, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
          num_pos_ = global[0];
          num_neg_ = global[1];

          if (myid == 0)
          {
              cout << "Total: " << num_total_
                        << "  Positive: " << num_pos_
                        << "  Negative: " << num_neg_ << endl;
          }

               // initialize the weight vector
               for(int i = 1 ; i <= 54 ; i++)
               {
                      Feature feature;
                      feature.id = i;
                      feature.weight = 0;
               //     feature.weight = (double)rand()/(double)RAND_MAX; // for MIRA
                      weight_vector.features.push_back(feature);
               }

               // initialize the average weight vector
               for(int i = 1 ; i <= 54 ; i++)
               {
                      Feature feature;
                      feature.id = i;
                      feature.weight = 0;
                      avg_weight_vector.features.push_back(feature);
               }

               train_perceptron(); // call one of the algorithm
//             train_mira(); //random init w. dont forget
//             train_opa();
               comm_weight_vector();       // sharing weight vector at end of T epoch
               send_weight_vector_to_all();
       }

}
Note: Same way test data is also read.
```

Training algorithms:

```
void train_perceptron(void)
{
       double wt_x;
       unsigned i, epoch;
       int yicap, mistakes = 0;
       int num_processors, myid;

       MPI_Comm_size(MPI_COMM_WORLD, &num_processors); // no of processors
       MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id


       for(epoch = 1; epoch <= max_epoch; epoch++)
       {
              mistakes = 0;

              for(i = 0; i < train_samples_.size(); i++) // only the samples assigned
              {
```

```
                        double actual_label = (GetLocalSample(i)->label > 1) ? 1 : -1;

                        wt_x = GetInnerProduct(*GetLocalSample(i),weight_vector);
                        if (wt_x >= 0)
                                yicap = 1;
                        else
                                yicap = -1;

                        if (actual_label * yicap < 0)
                        {
                                 Sample *sample = ScalarMultiplyVector(*GetLocalSample(i),
                                                                        actual_label);
                                AddToW(*sample);

                                mistakes = mistakes+1;
                        }
                }
                // enable this if after every epoch w is to be shared
                //comm_weight_vector();
                //send_weight_vector_to_all();
                // test_perceptron();


        }
}

void train_mira(void)
{
        double wt_x, yi_wt_xi,tow, norm_xisqr, temp;
        unsigned i, epoch;
        int yicap, mistakes = 0;
        int num_processors, myid;

        MPI_Comm_size(MPI_COMM_WORLD, &num_processors); // no of processors
        MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id

                for(epoch = 1; epoch <= max_epoch; epoch++)
                {
                        mistakes = 0;
                        for(i = 0; i < train_samples_.size(); i++) // only its samples
                        {
                                 int actual_label = (GetLocalSample(i)->label > 1) ? 1 : -
                                                                                        1;
                                wt_x = GetInnerProduct(*GetLocalSample(i),weight_vector);

                                if (wt_x >= 0)
                                        yicap = 1;
                                else
                                        yicap = -1;


                                // calculating tow
                                yi_wt_xi = actual_label * wt_x;

                                if(yi_wt_xi >= 0)
                                        tow = 0;
                                else
                                {
                                        norm_xisqr = GetLocalSample(i)->two_norm_sq;

                                        temp = yi_wt_xi/norm_xisqr;
                                        if(temp <= -1)
```

```
                                        tow = 1;
                                else
                                        tow = -temp;
                        }



                        // updating weight vector
                        if ((actual_label * yicap) < 0)
                        {

                   Sample *sample = ScalarMultiplyVector(*GetLocalSample(i),(tow *
                                                                        actual_label));
                                AddToW(*sample);
                                mistakes = mistakes + 1;
                        }

                }
                // enable this if after every epoch w is to be shared
                //comm_weight_vector();
                //send_weight_vector_to_all();
                // test_perceptron();


        }
}


void train_opa(void)
{

        double wt_x, m_yi_wt_xi,tow, norm_xisqr;
        unsigned i, epoch;
        int yicap,mistakes,actual_label,myid;

        MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id

                for(epoch = 1; epoch <= max_epoch; epoch++)
                {
                        mistakes = 0;

                        for(i = 0; i < train_samples_.size(); i++) // only its samples
                        {
                                actual_label = (GetLocalSample(i)->label > 1) ? 1 : -1;
                                wt_x = GetInnerProduct(*GetLocalSample(i),weight_vector);

                                if (wt_x >= 0)
                                        yicap = 1;
                                else
                                        yicap = -1;


                                // calculating tow
                                m_yi_wt_xi = 1 - (actual_label * wt_x);
                                norm_xisqr = GetLocalSample(i)->two_norm_sq;

                                if(m_yi_wt_xi < 0)
                                        tow = 0;
                                else
                                        tow = m_yi_wt_xi / norm_xisqr;

                                // updating weight vector
```

```
                        if ((actual_label * yicap) < 0)
                        {
                                Sample *sample =
ScalarMultiplyVector(*GetLocalSample(i),(tow * actual_label));
                                AddToW(*sample);
                                mistakes = mistakes + 1;
                        }

                }
            // enable this if after every epoch w is to be shared
            //comm_weight_vector();
            //send_weight_vector_to_all();
            // test_perceptron();

            }
}


void test_perceptron(void)
{
       int i;
       double wt_x = 0;
       int myid,test_mistakes = 0;
       char actual_label, predicted_label;

       MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id


         for(i = 0; i < test_samples_.size(); i++)
         {
              actual_label = ((GetLocalTestSample(i)->label) > 1) ? 1 : -1;
              wt_x = GetInnerProduct(*GetLocalTestSample(i), weight_vector);

              if(wt_x < 0)
                     predicted_label = -1;
              else
                     predicted_label = 1;
              if(actual_label != predicted_label)
                     test_mistakes = test_mistakes + 1;

         }

       cout << "proc: " << myid << " testing phase completed with "<< test_mistakes <<
"errors" <<endl;
}
```

Communicating weight vector:

```
void send_weight_vector_to_all(void)
{
       int num_processors, myid;
       char *buffer = NULL;
       int buff_size,np;

       MPI_Comm_size(MPI_COMM_WORLD, &num_processors); // no of processors
       MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id

       MPI_Barrier(MPI_COMM_WORLD);

       if(myid == 0)
       {
```

```
		// broadcast Avg_W to all nodes
		buff_size = PackSample(buffer,avg_weight_vector);
		for(np = 1; np < num_processors;np++)
		{
			MPI_Send(buffer, buff_size, MPI_BYTE, np,0, MPI_COMM_WORLD);
		}

	}
	else
	{
		// Get Avg_W and update locally
		int rcv_buff_size = 656;
		char *rcv_buffer = NULL;
		AVector *anewvector;
		anewvector = NULL;

		rcv_buffer = new char[rcv_buff_size];
		MPI_Recv(rcv_buffer, rcv_buff_size, MPI_BYTE,0,0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);

		UnpackSample(anewvector, rcv_buffer);

		vector<Feature>::const_iterator it1 = anewvector->features.begin();
		while(it1 != anewvector->features.end())
		{
			weight_vector.features[(it1->id)-1].id = it1->id;
			weight_vector.features[(it1->id)-1].weight = it1->weight;
			it1++;
		}

	}

}

// Processor 0 gets weight vector from all processors and finds average weight vector
void get_weight_vector_from_all(void)
{
	char *buffer = NULL;
	int buff_size,np;
	int num_processors, myid;
	double reg_prmtr;

	MPI_Comm_size(MPI_COMM_WORLD, &num_processors); // no of processors
	MPI_Comm_rank(MPI_COMM_WORLD, &myid); // current processor id
	reg_prmtr = (1.0/num_processors);

	buff_size = PackSample(buffer,weight_vector);

	MPI_Barrier(MPI_COMM_WORLD);

	if(myid != 0)
	{
		MPI_Send(buffer, buff_size, MPI_BYTE, 0,0, MPI_COMM_WORLD);
	}
	else
	{
		int rcv_buff_size = 656;    // 54 dim
		char *rcv_buff = NULL;
		rcv_buff = new char[rcv_buff_size];
		AVector *avector;

		for(np = 1; np < num_processors; np++)
```

```
		{
			avector = NULL;
			MPI_Recv(rcv_buff, rcv_buff_size, MPI_BYTE,np,0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
			UnpackSample(avector, rcv_buff);

		  // add up all the weight vectors
			vector<Feature>::const_iterator it1 = avector->features.begin();
			while(it1 != avector->features.end())
			{
				avg_weight_vector.features[(it1->id)-1].id = it1->id;
				avg_weight_vector.features[(it1->id)-1].weight =
avg_weight_vector.features[(it1->id)-1].weight + it1->weight;
				it1++;
			}

		}
		// add local machine weight vector finally
		vector<Feature>::const_iterator it3 = weight_vector.features.begin();
		while(it3 != weight_vector.features.end())
		{
			avg_weight_vector.features[(it3->id)-1].id = it3->id;
			avg_weight_vector.features[(it3->id)-1].weight =reg_prmtr *
(weight_vector.features[(it3->id)-1].weight + it3->weight);
			it3++;
		}
		// this will be new weight vector

	}
}
```

Appendix: Part-C

dcd.m

```
C = 10;
p = 1600;    % size(Centroid,1); % Number of Samples
n = 49990;   % # of training data points
d = 22;      % Dimension of training data point
gamma = (1/22);

% load training data
[trainlabels, trainfeatures] = libsvmread('datasets\ijcnn\ijcnn1\ijcnn1');

Z = zeros(p,d);
w = zeros(p,1);
alpha = zeros(1,n);
max_iter = 10;
flag = 1;

% Pick p samples using k-means centroids
% for i = 1 : p
%     Z(i,:) = Centroid(i,:);
% end

%Pick 'p' samples randomly
p_pts = randperm(n,p);
for i = 1 : p
    Z(i,:) = trainfeatures(p_pts(i),:);
end

Zt = Z';
% polynomial kernel
% Find Kzz
Kzz = (single(gamma * (Z * Zt))) .^ 2;  %(Z*Zt) in case of linear

% Eigen value decomposition
[EigVects, EigVals]  = eig(Kzz);  % Kzz = EigVects * EigVals * EigVects'

% Find M
M = EigVects * (EigVals ^ (-1/2));
clear EigVects EigVals Kzz;

% Find Krz
Krz = (gamma * (trainfeatures * Zt)) .^ 2;
clear trainfeatures;

% Find FrCap
FrCap = single(Krz * M);
clear Krz;
```

```
% Train linear SVM using Dual Co-ordinate Descent(DCD) method

for iter = 1:max_iter

    for i = 1:n

        xi = FrCap(i,:);
        xit = xi';
        yi = trainlabels(i);
        yiwtxi = yi * (xi * w);

        % If KKT conditions are not satisfied
        if( ~(((alpha(i) == 0) && (yiwtxi >= 1)) || ((alpha(i) == C) &&
(yiwtxi <= 1)) || ((alpha(i) > 0) && (alpha(i) < C) && (yiwtxi == 1))))
            towcap = (1-yiwtxi) / (xi * xit);

            if(towcap <= -alpha(i))
                tow = -alpha(i);
            elseif(towcap >= C - alpha(i))
                tow = C-alpha(i);
            else
                tow = towcap;
            end

            % Update weight vector and alpha with tow
            w = w + (tow * yi) * xit;
            alpha(i) = alpha(i) + tow;
        end
    end

end
```

testdcd.m(with polynomial kernel)

```
gamma = (1/22);

% load test data
[testlabels, testfeatures] =
libsvmread('datasets\ijcnn\ijcnn1.t\ijcnn1.t');

Kez = (single(gamma * (testfeatures * Z'))) .^ 2;

ycap = sign(Kez * (M * w));

ycap = real(ycap);

idx_actual = find(testlabels == 1);
idx_predicted = find(ycap == 1);

%accuracy = size(idx_predicted,1) / size(idx_actual,1);
accuracy = (sum(ycap == testlabels)) / size(testlabels,1) * 100;
testerr = 100 - accuracy;
```