CRAFTML

An efficient Clustering-based Random Forest for extreme Multi-label Learning.

TEAM MEMBERS

Sayantan Jana (20171185)
sayantan.jana@research.iiit.ac.in

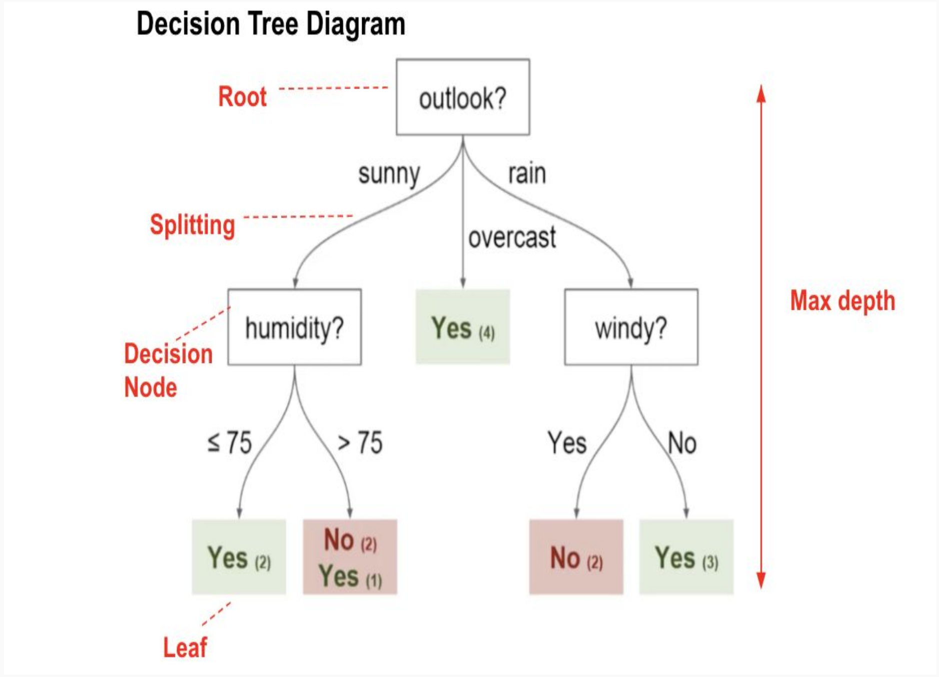
Gaurav Sultane (20171036)
gaurav.sultane@students.iiit.ac.in

Extreme Multi-label classification

- Extreme Multi-label learning consists of large sets of items described by a number of labels that can exceed one million.
- Tree based hierarchical methods have been used in these problems as they reduce the training and testing complexity
- We are using the Random forest based approach which help us to exploit the benefits of tree randomization and also supports parallel computation.

CRAFTML introduction

- CRAFTML is a forest of trees that transform the initial large-scale problem into a series of small-scale sub-problems by hierarchically partitioning the instance set or the label set, closely resembling decision trees . Contrary to most of the current XML methods, CRAFTML does not rely on a complex optimization scheme. It combines simple and fast learning blocks (e.g. k-mean clustering, basic multi-class classifier) which leaves room for extensions to reach performances required by current challenges .



Notations

- We consider a training set of n instances.
- Each instance will be described by 2 vectors, x and y.
- x is the feature vector $\in \mathbb{R}^{dx}$ of d_x features.
- y is the label vector $\in \mathbb{R}^{dy}$ of d_y labels.
- The feature matrix X of the set is n ***x*** d_x matrix where each row corresponds to feature vector of an instance
- The label matrix Y of the set is n ***x*** d_y matrix where each row corresponds to label vector of an instance

Algorithm

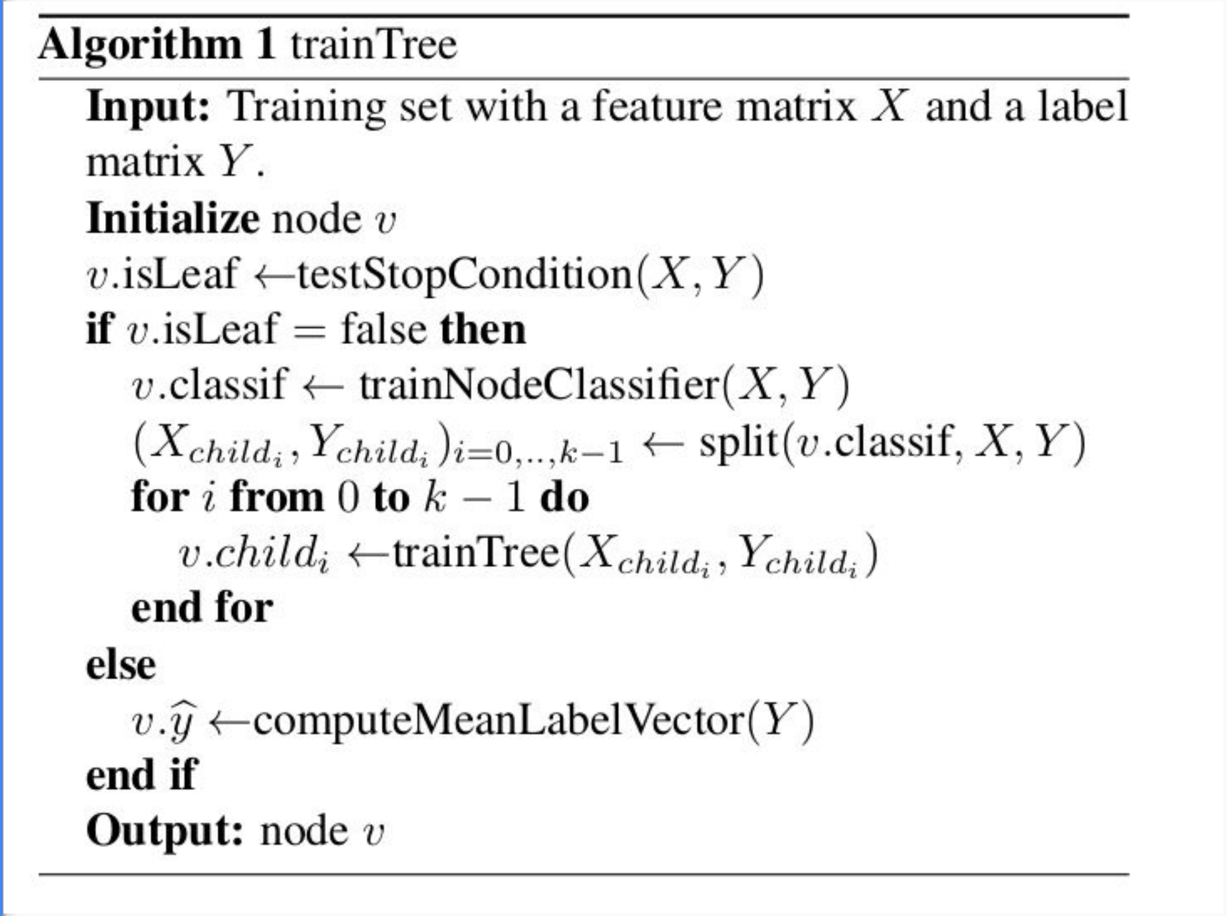
- CRAFTML computes a forest of m_F k-ary instance trees.
- The construction of tree follows common scheme of instance based tree methods.
- K-ary trees means each node of the tree will have k childrens except the leaves.
- So at each node, which is not a leaf. We want to partition the data into k parts which such that those parts are similar to each other.

Training the Decision Tree

A node is classified as a leaf if it obeys any of the following 3 properties

- If cardinality of a node's instance is lower than some threshold $n_{leaf.}$
- All the instances have similar features
- All the instances have similar labels

A leaf will store the average label vector of all the instances which will be used for prediction



Algorithm 1 trainTree

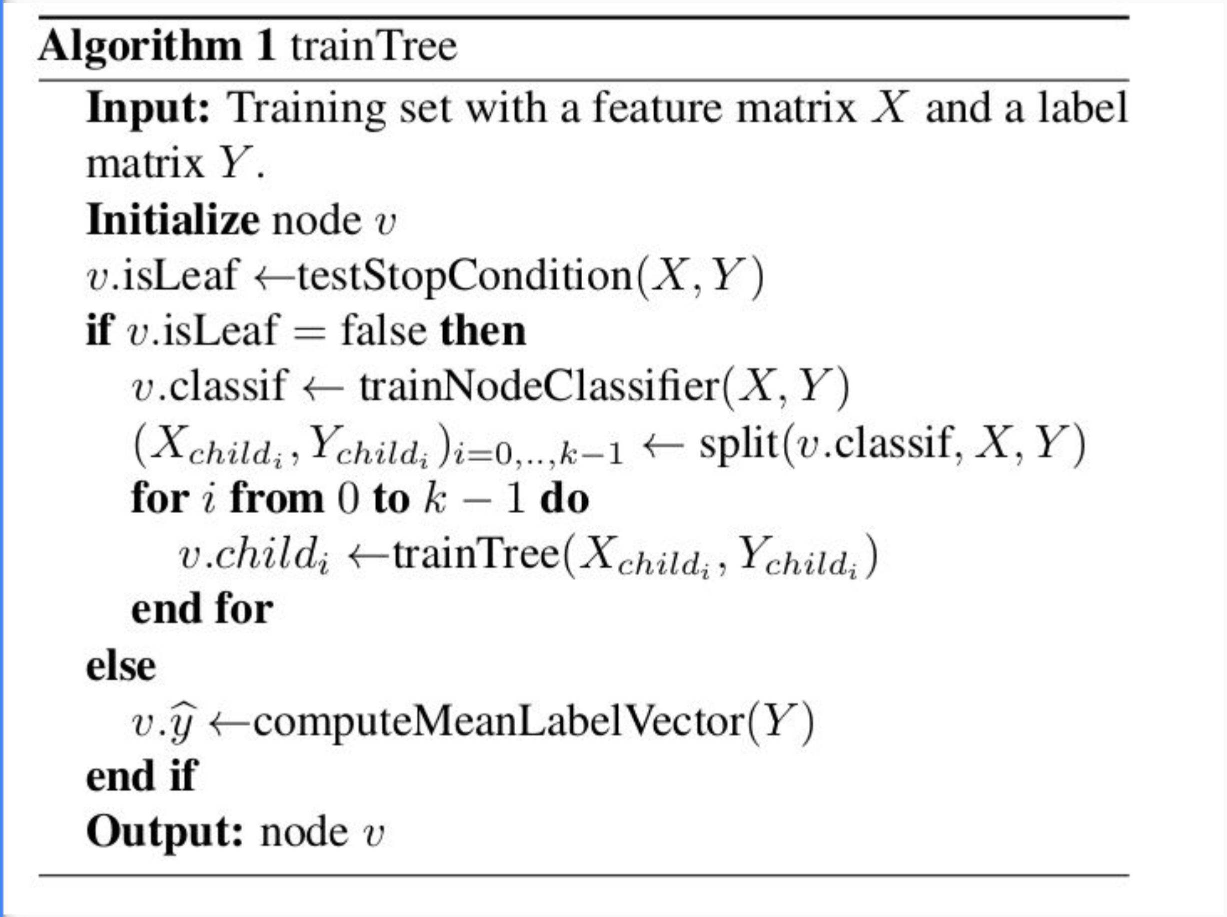
```
Input: Training set with a feature matrix $X$ and a label matrix $Y$.
Initialize node $v$
$v$.isLeaf ← testStopCondition($X, Y$)
if $v$.isLeaf = false then
    $v$.classif ← trainNodeClassifier($X, Y$)
    $(X_{child_i}, Y_{child_i})_{i=0,..,k-1}$ ← split($v$.classif, $X, Y$)
    for $i$ from $0$ to $k-1$ do
        $v.child_i$ ← trainTree($X_{child_i}, Y_{child_i}$)
    end for
else
    $v.\widehat{y}$ ← computeMeanLabelVector($Y$)
end if
Output: node $v$
```

Training the Decision Tree

If current node does not satisfy the leaf stop condition, we will split the data into k parts ,and pass this parts to the k children of the current node.

How do we divide the data into these k parts?

We want similar instances to be grouped together. We will use clustering to do this. Details in node training.

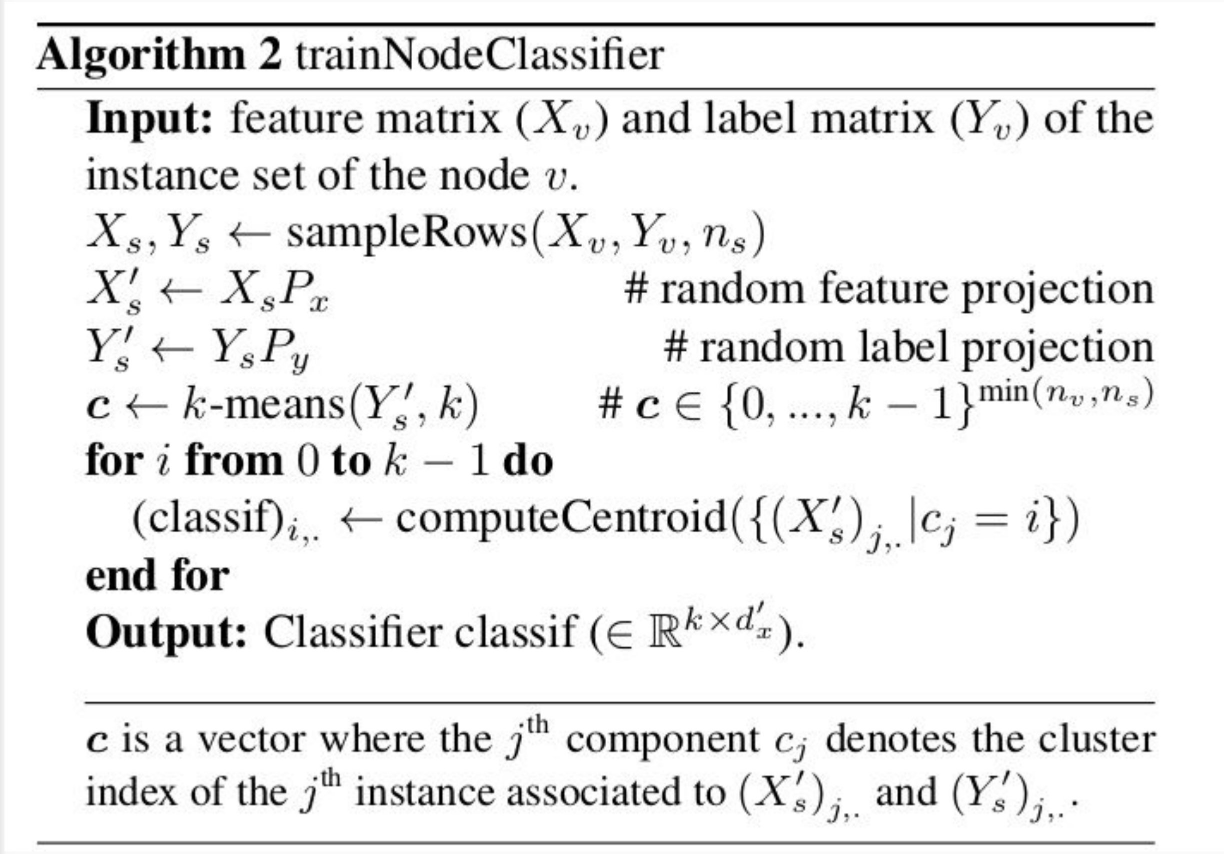


Algorithm 1 trainTree

```
Input: Training set with a feature matrix $X$ and a label matrix $Y$.
Initialize node $v$
$v$.isLeaf $\leftarrow$testStopCondition$(X, Y)$
if $v$.isLeaf $=$ false then
    $v$.classif $\leftarrow$ trainNodeClassifier$(X, Y)$
    $(X_{child_i}, Y_{child_i})_{i=0,..,k-1} \leftarrow$ split$(v$.classif$, X, Y)$
    for $i$ from $0$ to $k-1$ do
        $v.child_i \leftarrow$trainTree$(X_{child_i}, Y_{child_i})$
    end for
else
    $v.\widehat{y} \leftarrow$computeMeanLabelVector$(Y)$
end if
Output: node $v$
```

Node Training

- We wish to partition the instances into k clusters.
- We want the instances with similar labels to belong to the same clusters, hence we will use label matrix for clustering
- We will use spherical k-means clustering which is based on the cosine similarity of two data points as compared to euclidean distance in normal k-means clustering.



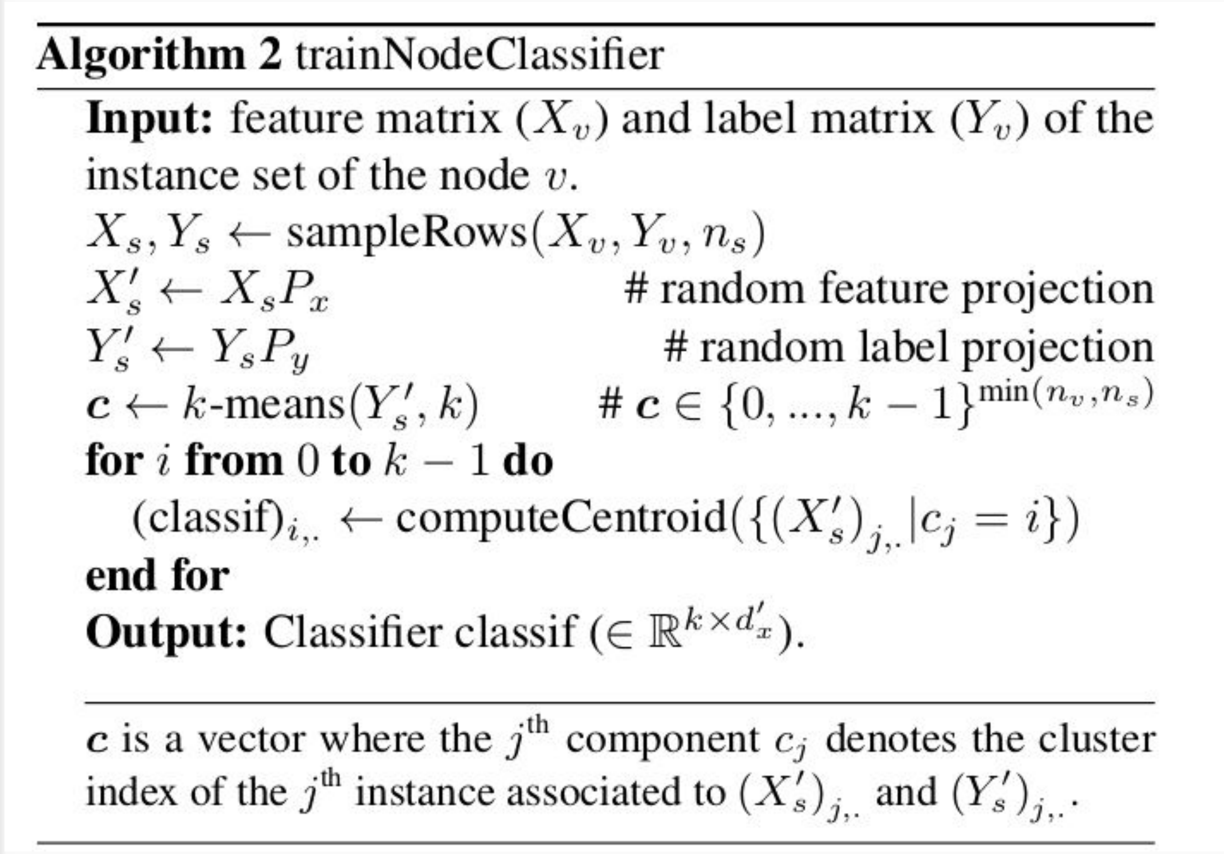
Algorithm 2 trainNodeClassifier

```
**Input:** feature matrix ($X_v$) and label matrix ($Y_v$) of the instance set of the node $v$.
$X_s, Y_s \leftarrow \text{sampleRows}(X_v, Y_v, n_s)$
$X'_s \leftarrow X_s P_x$                  # random feature projection
$Y'_s \leftarrow Y_s P_y$                  # random label projection
$\boldsymbol{c} \leftarrow k\text{-means}(Y'_s, k)$          # $\boldsymbol{c} \in \{0, \ldots, k-1\}^{\min(n_v, n_s)}$
**for** $i$ **from** $0$ **to** $k-1$ **do**
    $(\text{classif})_{i,.} \leftarrow \text{computeCentroid}(\{(X'_s)_{j,.} | c_j = i\})$
**end for**
**Output:** Classifier classif ($\in \mathbb{R}^{k \times d'_x}$).
```

$\boldsymbol{c}$ is a vector where the j^{th} component c_j denotes the cluster index of the j^{th} instance associated to $(X'_s)_{j,.}$ and $(Y'_s)_{j,.}$.

Node Training

- Assign a class number between 0 and k-1 to each instance indicating the cluster number this instance belong to.
- Build a classifier at current node, which will predict to which children an unlabelled instance will go.
- Each child will have a Feature indicator classif$_i$, an instance will go the child having feature indicator most similar to instance's features (cosine similarity).



Algorithm 2 trainNodeClassifier

```
**Input:** feature matrix ($X_v$) and label matrix ($Y_v$) of the instance set of the node $v$.
$X_s, Y_s \leftarrow \text{sampleRows}(X_v, Y_v, n_s)$
$X'_s \leftarrow X_s P_x$                      # random feature projection
$Y'_s \leftarrow Y_s P_y$                      # random label projection
$\mathbf{c} \leftarrow k\text{-means}(Y'_s, k)$          # $\mathbf{c} \in \{0, ..., k-1\}^{\min(n_v, n_s)}$
for $i$ from $0$ to $k-1$ do
    $(\text{classif})_{i,.} \leftarrow \text{computeCentroid}(\{(X'_s)_{j,.} | c_j = i\})$
end for
**Output:** Classifier classif ($\in \mathbb{R}^{k \times d'_x}$).
```

$\mathbf{c}$ is a vector where the j^{th} component c_j denotes the cluster index of the j^{th} instance associated to $(X'_s)_{j,.}$ and $(Y'_s)_{j,.}$.

Node Training

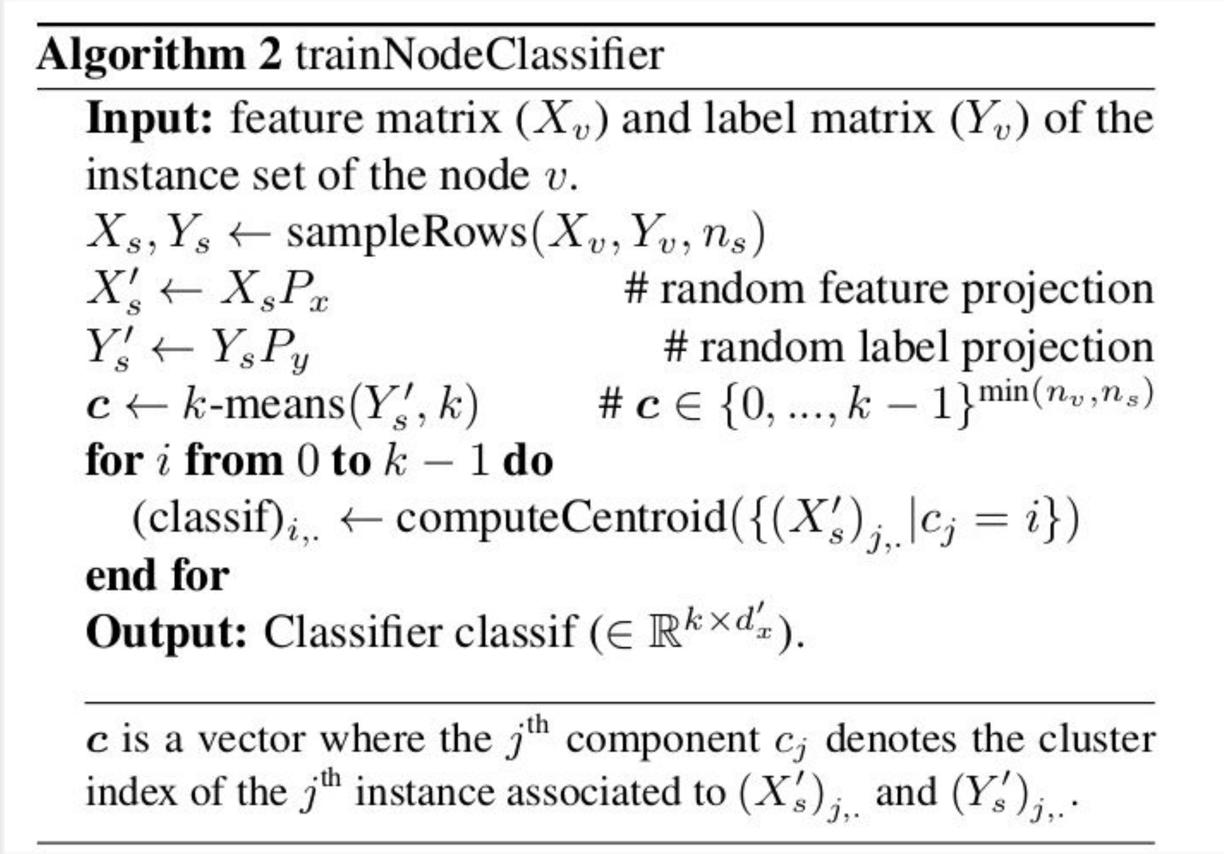
Working on the original data will incur huge computational complexity.

We will randomly select some samples of the original data for building the classifier.

We will also reduce features and labels to a lower dimensional subspaces to reduce the computation time.

P_x => Feature projection matrix

P_y => Label projection matrix



Algorithm 2 trainNodeClassifier

```
Input: feature matrix ($X_v$) and label matrix ($Y_v$) of the instance set of the node $v$.
$X_s, Y_s \leftarrow$ sampleRows$(X_v, Y_v, n_s)$
$X'_s \leftarrow X_s P_x$                    # random feature projection
$Y'_s \leftarrow Y_s P_y$                    # random label projection
$\mathbf{c} \leftarrow k$-means$(Y'_s, k)$     # $\mathbf{c} \in \{0, ..., k-1\}^{\min(n_v, n_s)}$
for $i$ from $0$ to $k-1$ do
    $(\text{classif})_{i,.} \leftarrow$ computeCentroid$(\{(X'_s)_{j,.} | c_j = i\})$
end for
Output: Classifier classif ($\in \mathbb{R}^{k \times d'_x}$).
```

$\mathbf{c}$ is a vector where the j^{th} component c_j denotes the cluster index of the j^{th} instance associated to $(X'_s)_{j,.}$ and $(Y'_s)_{j,.}$.

Sampling and Projection

- Sampling: Some samples will be selected randomly from the given set of samples at the current node for training
- Projection:
  - We have used sparse orthogonal projection (Weinberger et al., 2009) for projection. (Hashing trick)
  - It helps us to maintain sparsity and is much faster.
  - Also, we don’t need large storage space as only seed is required to generate projection matrices.

Projection

Projection Implementation:

- Consider a vector v : {i,val} (ith element of vector is val), We want to project this vector to projV
  - For {i,val} ∈ v, proj[Hash(i, seedElem) % projDim] += val*(2*(Hash(i,seedSign)%2) -1)
  - This guarantees that number of non zero elements in projected vector will be at most that of the original vector.

Diversity of projection matrices between nodes of same tree:

By observation it was found that, it is always better to use same seeds for projection across all nodes.

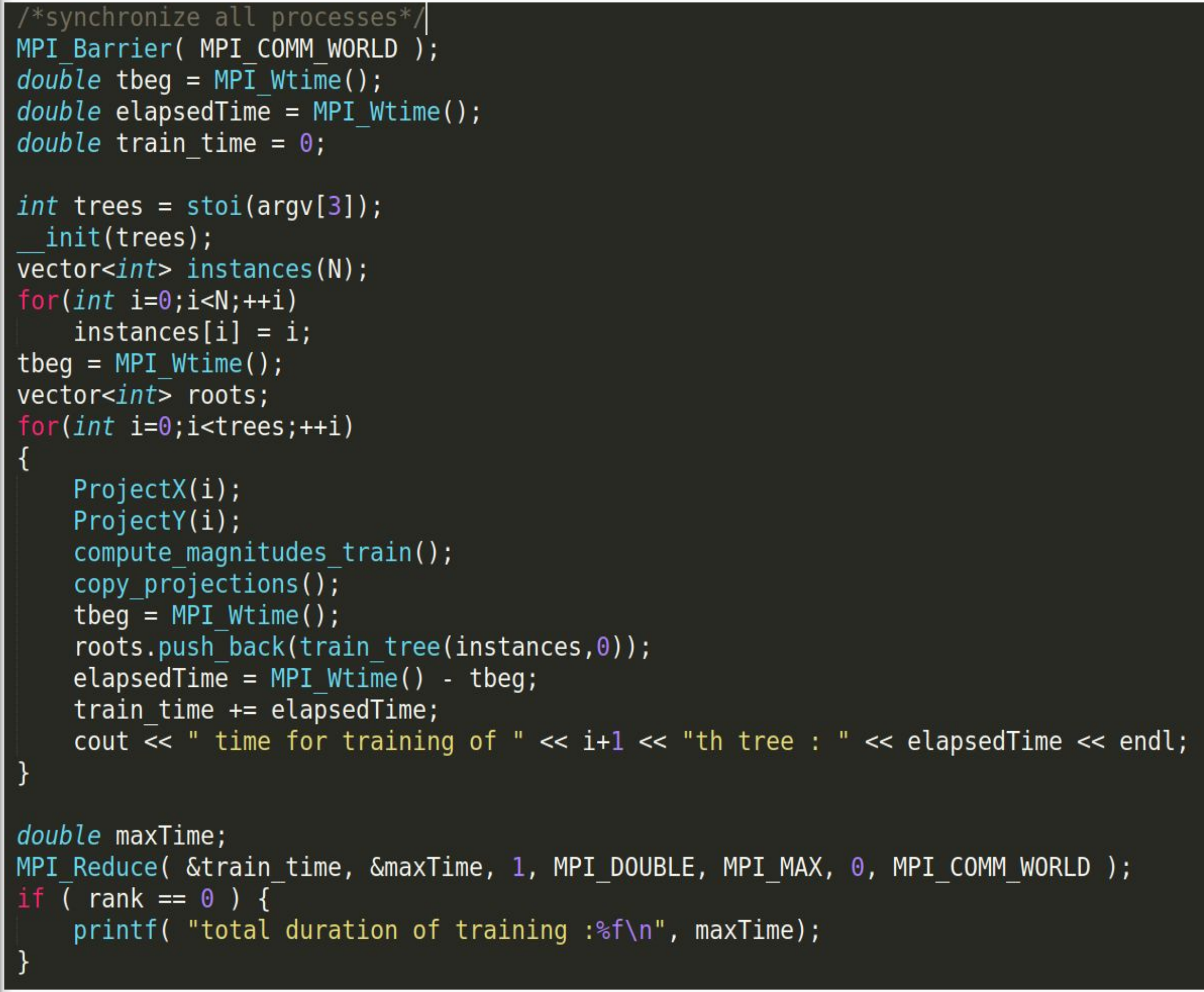
Testing

- For a test data , we start at root of each tree and on the basis of the node classifier at the node we move down into one of its children, we keep descending down until we reach a leaf and pick its meanlabel vector, now for all the trees in the forest we take means of all the vectors and classify the point to its most likely top 1, 3 and 5 labels and hence compute the precision @k scores for all the tests involved .

Parallelisation

Achieved through MPI :

- The first level of parallelisation simply involves allowing each of the process (let’s assume total p of them) to train a forest of t trees.
- As the training of p*t trees are all done, for each test data as each process decides the meanlabel for the test, the mean across all the processes are taken to finally classify the point using MPI_Reduce to decide the global mean vector at process 0 .

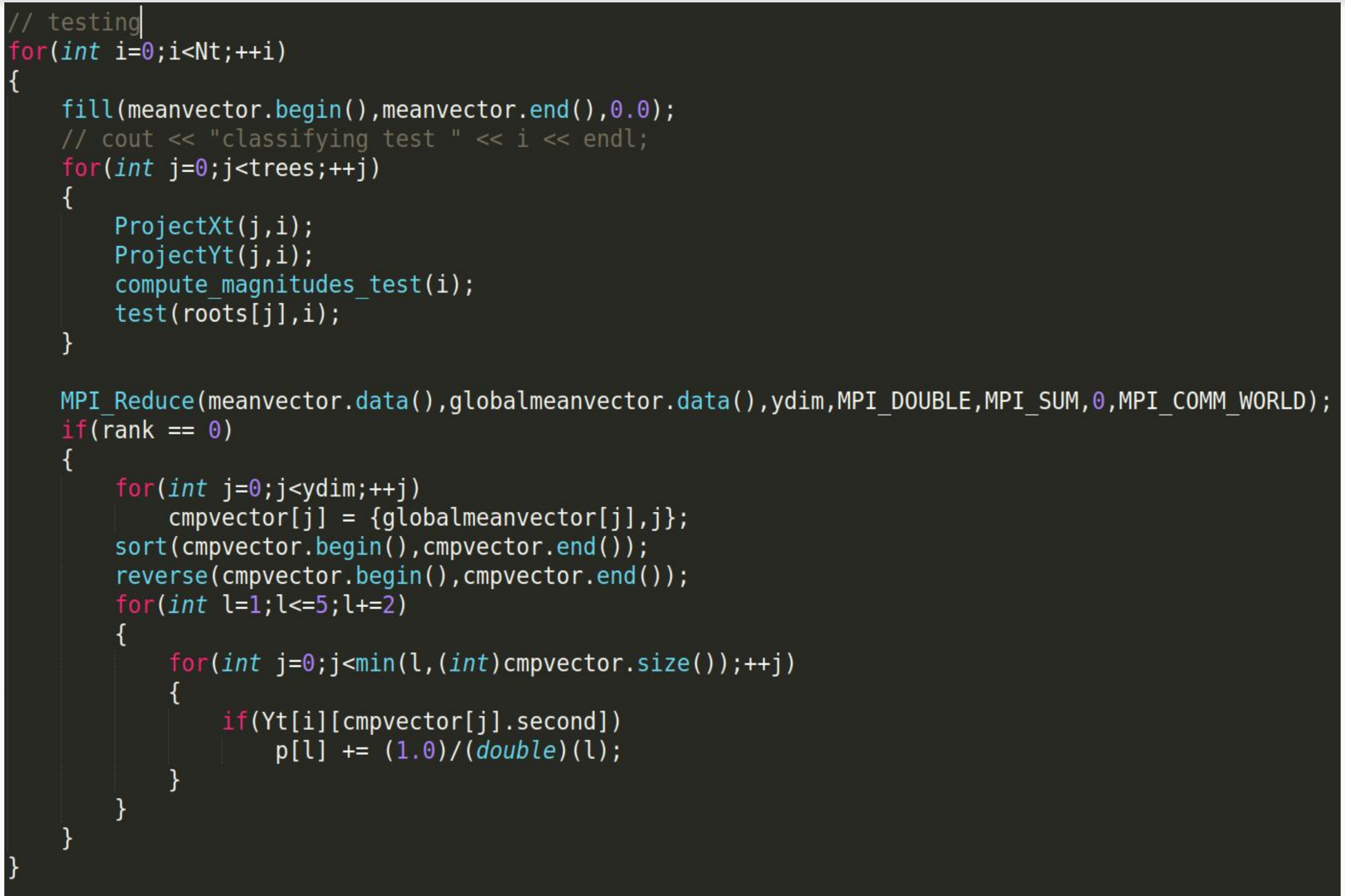


```
/*synchronize all processes*/
MPI_Barrier( MPI_COMM_WORLD );
double tbeg = MPI_Wtime();
double elapsedTime = MPI_Wtime();
double train_time = 0;

int trees = stoi(argv[3]);
__init(trees);
vector<int> instances(N);
for(int i=0;i<N;++i)
    instances[i] = i;
tbeg = MPI_Wtime();
vector<int> roots;
for(int i=0;i<trees;++i)
{
    ProjectX(i);
    ProjectY(i);
    compute_magnitudes_train();
    copy_projections();
    tbeg = MPI_Wtime();
    roots.push_back(train_tree(instances,0));
    elapsedTime = MPI_Wtime() - tbeg;
    train_time += elapsedTime;
    cout << " time for training of " << i+1 << "th tree : " << elapsedTime << endl;
}

double maxTime;
MPI_Reduce( &train_time, &maxTime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD );
if ( rank == 0 ) {
    printf( "total duration of training :%f\n", maxTime);
}
```

Parallelisation

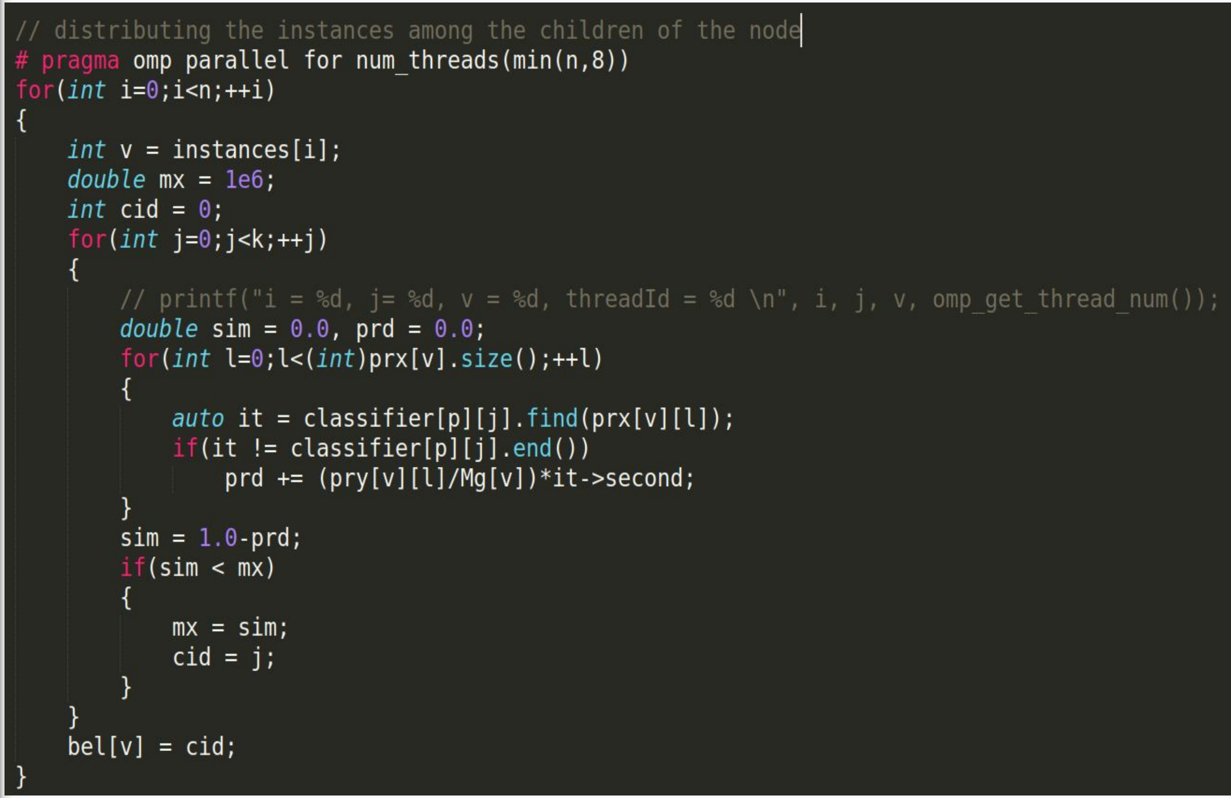


```
// testing
for(int i=0;i<Nt;++i)
{
    fill(meanvector.begin(),meanvector.end(),0.0);
    // cout << "classifying test " << i << endl;
    for(int j=0;j<trees;++j)
    {
        ProjectXt(j,i);
        ProjectYt(j,i);
        compute_magnitudes_test(i);
        test(roots[j],i);
    }

    MPI_Reduce(meanvector.data(),globalmeanvector.data(),ydim,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
    if(rank == 0)
    {
        for(int j=0;j<ydim;++j)
            cmpvector[j] = {globalmeanvector[j],j};
        sort(cmpvector.begin(),cmpvector.end());
        reverse(cmpvector.begin(),cmpvector.end());
        for(int l=1;l<=5;l+=2)
        {
            for(int j=0;j<min(l,(int)cmpvector.size());++j)
            {
                if(Yt[i][cmpvector[j].second])
                    p[l] += (1.0)/(double)(l);
            }
        }
    }
}
```

Parallelisation

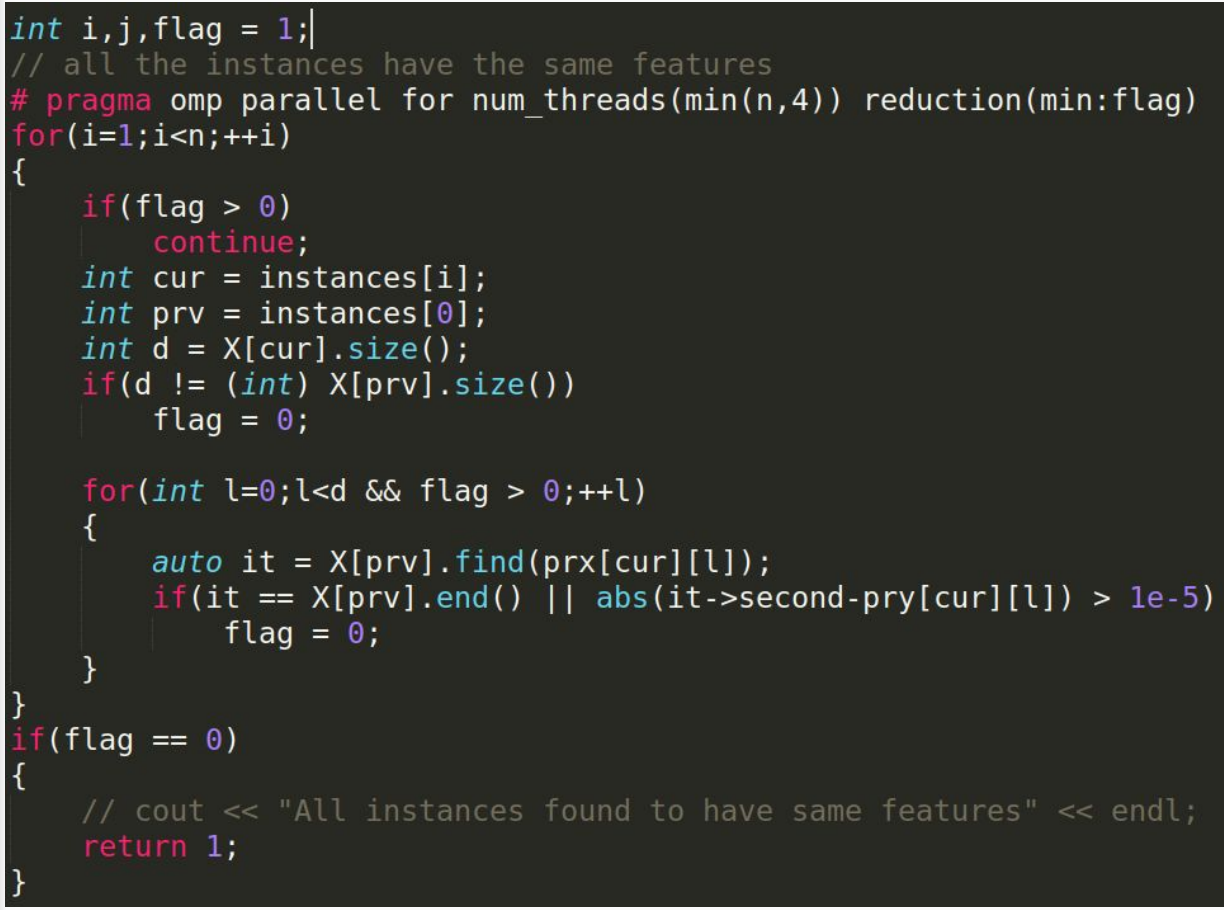
- The second level of parallelisation involves optimizing the heavy parts of the training algorithm using OpenMP threading .
- The heaviest part in algorithm is the data seggregation at each node according to the node classifier built . Note that each data can be independently mapped to a children, so parallelisation is possible over here .



```
// distributing the instances among the children of the node
# pragma omp parallel for num_threads(min(n,8))
for(int i=0;i<n;++i)
{
    int v = instances[i];
    double mx = 1e6;
    int cid = 0;
    for(int j=0;j<k;++j)
    {
        // printf("i = %d, j= %d, v = %d, threadId = %d \n", i, j, v, omp_get_thread_num());
        double sim = 0.0, prd = 0.0;
        for(int l=0;l<(int)prx[v].size();++l)
        {
            auto it = classifier[p][j].find(prx[v][l]);
            if(it != classifier[p][j].end())
                prd += (pry[v][l]/Mg[v])*it->second;
        }
        sim = 1.0-prd;
        if(sim < mx)
        {
            mx = sim;
            cid = j;
        }
    }
    bel[v] = cid;
}
```

Parallelisation

- For leaf testing, we need to look through all the instance vectors to check if they are the same or not at leaves, this is again easily parallelized , each being an independent check .
- Apart from this, the projection of the feature and label matrices come under the domain of generalized matrix vector multiplication and hence any fast known way can be employed here .
- We did not extend to further parallelization since with added threading for more functions led to slow-downs .



```
int i,j,flag = 1;
// all the instances have the same features
# pragma omp parallel for num_threads(min(n,4)) reduction(min:flag)
for(i=1;i<n;++i)
{
    if(flag > 0)
        continue;
    int cur = instances[i];
    int prv = instances[0];
    int d = X[cur].size();
    if(d != (int) X[prv].size())
        flag = 0;

    for(int l=0;l<d && flag > 0;++l)
    {
        auto it = X[prv].find(prx[cur][l]);
        if(it == X[prv].end() || abs(it->second-pry[cur][l]) > 1e-5)
            flag = 0;
    }
}
if(flag == 0)
{
    // cout << "All instances found to have same features" << endl;
    return 1;
}
```

Benchmarking

- With the available resources we have check the performance of our code against the popular datasets available at

  http://manikvarma.org/downloads/XC/XMLRepository.html
- We have tried against Mediamill, Bibtex, Delicious, EURLex-4K and Wiki10-31K.

Bibtex [performed on 10 splits of data]

6 processes 1 tree

- total duration of training 77.230885
- p@1 : 56.7356
- p@3 : 34.0928
- p@5 : 24.8549

5 processes 10 trees

- total duration of training 800.374705
- p@1 : 60.9185
- p@3 : 36.9768
- p@5 : 27.0815

Paper’s results for 50 trees :

- p@1 : 65.15
- p@2 : 39.83
- p@3 : 28.99

Delicious [performed on 10 splits of data]

1 process 1 tree

- total duration of training 477.962813
- p@1 : 47.9498
- p@3 : 47.3354
- p@5 : 45.3344

4 processes 1 tree

- total duration of training 697.001516
- p@1 : 62.9137
- p@3 : 58.1392
- p@5 : 54.0006

Paper achieves the p-scores on 50 trees :

- p@1 : 70.26
- p@3 : 63.98
- p@5 : 59.00

Mediamill [performed on 10 splits of data]

1 process, 1 tree :

- total duration of training 428.851507
- p@1 : 62.6065
- p@3 : 55.4827
- p@5 : 47.3991

4 processes 1 tree each, total 4 trees :

- total duration of training 575.464897
- p@1 : 84.6732
- p@3 : 68.5922
- p@5 : 54.8496

Paper achieves the p-scores on 50 trees :

- p@1 : 85.86
- p@3 : 69.01
- p@5 : 54.65

Wiki10-31K

For 1 process and 1 tree :

- Duration of training : 112.4304
- p@1 : 76.3755
- p@3 : 62.3539
- p@5 : 52.0345

For 2 process , 2 trees each, total 4 trees :

- Duration of training : 211.2771
- p@1 : 80.8495
- p@2 : 67.901
- p@3 : 57.9353

For 1 process, 5 trees :

- Duration of training : 516.3119
- p@1 : 81.0308
- p@3 : 69.216
- p@5 : 58.8815

Paper achieves the following p-scores on 50 trees :

- p@1 : 85.19
- p@3 : 73.17
- p@5 : 63.27

Eurlex-4K

1 process, 1 tree :

- duration for training : 39.8519
- p@1 : 52.9273
- p@3 : 47.029
- p@5 : 39.3437

1 process, 1 tree, parallelised by openmp :

- duration for training : 21.2891
- p@1 : 56.5503
- p@3 : 49.068
- p@5 : 40.7193

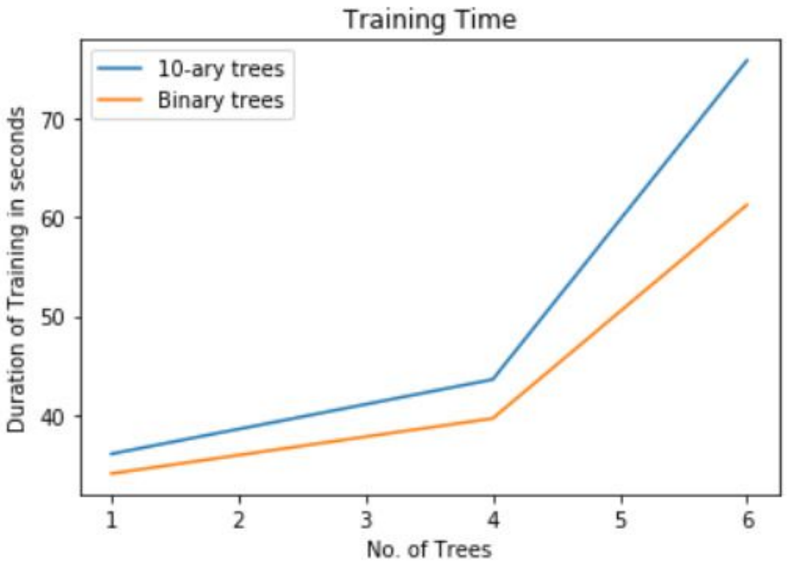
10 process, 5 trees each, total 50 trees :

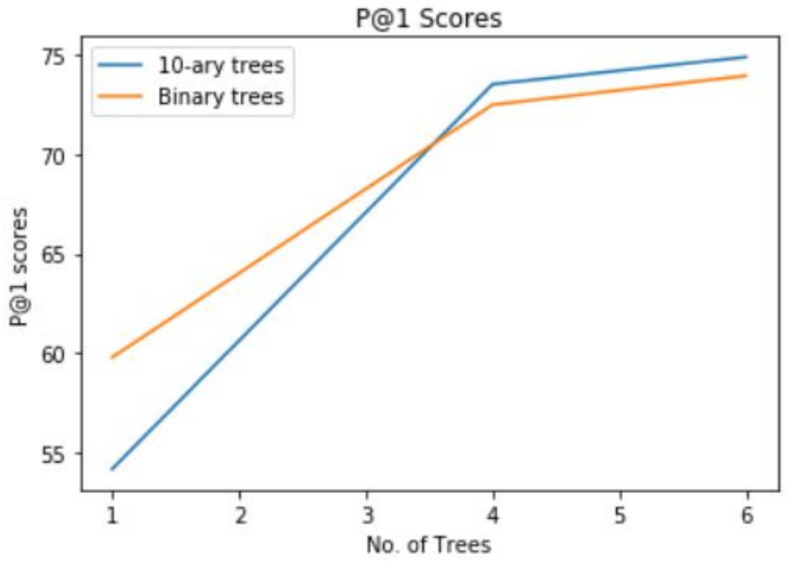
- p@1: 78.44
- p@3: 64.61
- p@5: 53.50

Results obtained by paper for 50 trees :

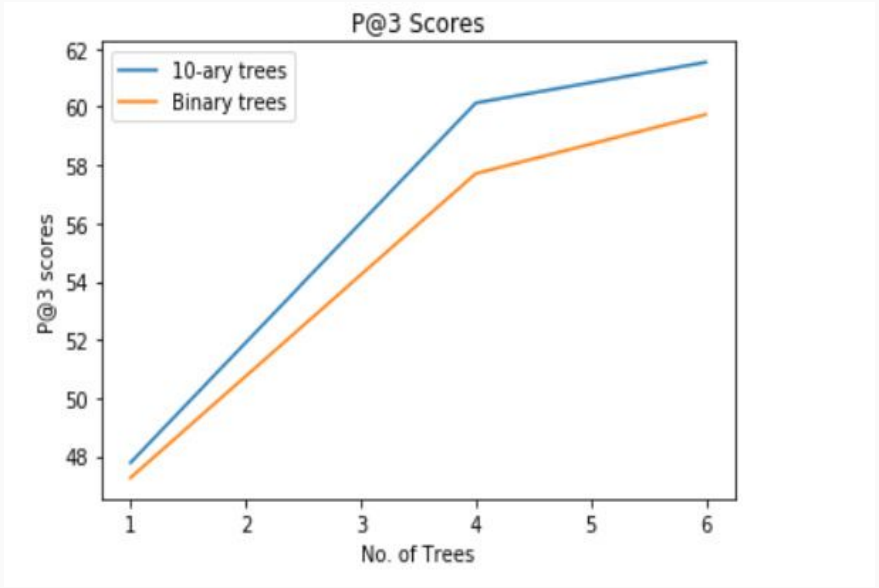
- p@1: 78.81
- p@3: 65.21
- p@5: 53.71

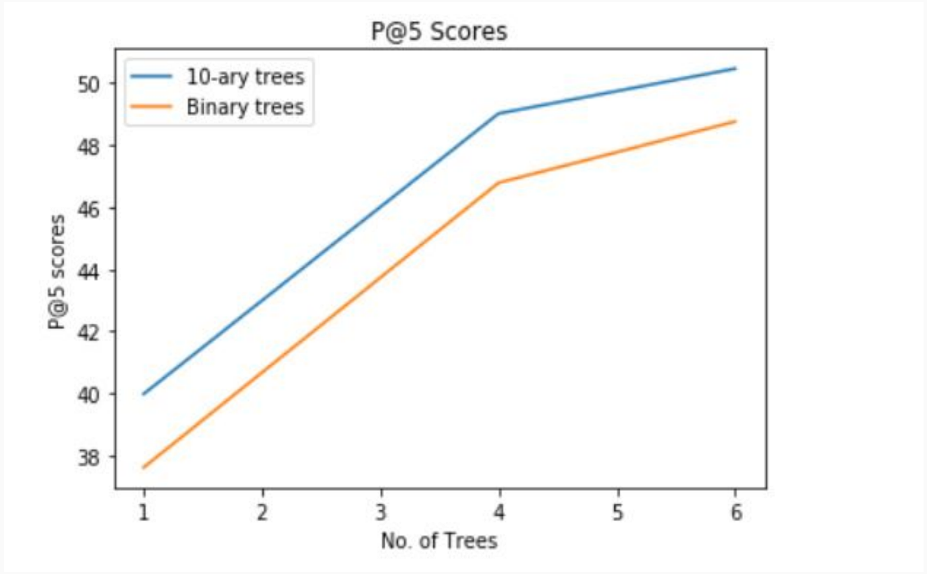
Comparision on Eurlex-4K





Comparison on Eurlex-4K





Observations on performances

- Compared to the sequential code, parallelisation through MPI alone can give ~3 times speed up .
- Compared to the sequential code, parallelisation through OpenMP threading give ~2 times speed up.
- But together OpenMP threading with MPI parallelisation through processes gives around only 5-10% speedup than what's already achieved by MPI parallelisation .

Improvements

- If we aim to speed up further just on MPI parallelisation, OpenMP threading doesn’t seem to be the perfect way, rather one can use Parallel STL

“ Parallel STL is an implementation of the C++ standard library algorithms. Parallel STL offers a portable implementation of threaded and vectorized execution of standard C++ algorithms, optimized and validated for Intel(R) 64 processors. For sequential execution, it relies on an available implementation of the C++ standard library. “

Thank You