IPSC-Project

Final Presentation
(Craft-ML)

Anupam Majumder (2019201002)
Tom Sabu (2019201005)
Shraddha(2019201001)

Outline:

- Benchmarks in code:
    - Loading data(DataSet considered : Eurlex 4K).
    - Projection of data.
    - Constructing Forest from Train data.
        - Making clusters from labels using KMeans++(for tree construction).
    - Predicting labels for Test data.
- Challenges:
    - In storing length of data.
    - Tree Traversal.
    - Memory management done.
    - **Parallelized different modules in code.**
    - Performance Analysis.

1.Loading Data: (Eurlex-4k)

- Format of Data in Text File:

  label1,label2,...labelk ft1:ft1_val ft2:ft2_val ft3:ft3_val .. ftd:ftd_val

- Format of Data in Code:

  Converted into 4 arrays:

  - keysX (containing indexes of features i.e. ft1, ft2, ft3.... ftd)
  - valuesX (containing values of features i.e. ft1_val, ft2_val, ft3_val… ftd_val)
  - keysY (containing the labels i.e. label1, label2….. labelk)
  - valuesY (containing “1” corresponding to each label)

Data Structures Used :

(i). **Structure** used for reading data:

```
struct Args_device
{
        float** valuesX;
        float** valuesY;
        int** keyX;
        int** keyY;
        int* useInst;
        int dimXProj;
        int dimYProj;
        int sparsity;
        int seedX;
        int seedY;
        int minInst;
};
```

Why not Class??

(ii).Structure of a Node in Tree:

```
struct Node
{
        bool Leaf;
        float labels[4000];
        float center_valuesX[10][101];
        int center_indexesX[10][101];
        struct Node* children[10];
};
```

(iii).Structure used for Clusterization(in KMeans++):

```
struct cluster
{
        float clusters[10][2000];
        int chosenClus[10];
        int indexBelong[15600];
};
```

2.Projection of Data:

(i) Calculate **Hash32** on data:

- Function definition : **__device__ uint32_t MurmurHash2 (const void * key, int len, uint32_t seed)** where
    - key=’azv’+data_value
    - len=length of key
    - seed=seed is a number (randomly generated) used to calculate hash.

(ii) **Reduce dimensionality** to a certain value(namely dimXproj,dimYproj in code):

```
for(int i = 1;i<=xKey[0];i++)
{
        currentIndex = getIndex(xKey[i], d_args->seedX, d_args->dimXProj);
        currentSign = getSign(xKey[i], d_args->seedX);
        projectedX[currentIndex] = projectedX[currentIndex] + currentSign * xValues[i];
}
```

(iii) **Normalise** data after reduction:

```
norm += projectedX[i]*projectedX[i];
…..
for(int i = 0;i<d_args->dimXProj;i++){
        if(projectedX[i] !=0) {
                currIndexes[k] = i;
                currValues[k] = projectedX[i]/norm;
                k++;
        }
}
```

3.Forest Construction:

(i) Forming clusters on **label** data:

(a) Calculate Projected values for Label Data:(Same as in “Feature Data”)

- ○ Hash32->Dimensionality Reduction->Normalise Data.

(b) Performing **KMeans++** to create clusters of labels.

```
for(int i = 1; i < 10;i++)
{       for(int j = 0;j<nbInst;j++)
        {       float min_=FLT_MAX;
                for(int k=0;k<i;k++)
                {       float dist=cosineDistance(dataValues[j],dataIndexes[j],cl.clusters[k]);
                        if(dist<min_)
                                min_=dist;
                }
                currentDist = min_;
                probaCum[j+1] = probaCum[j] + currentDist*currentDist;
        }
```

contd...

```
    .....
    currentRandom =((float)rand()/RAND_MAX)*probaCum[nbInst];
    index = getClusterIndex(currentRandom,probaCum,0,nbInst);
    cl.chosenClus[i] = index;
    float *currentValues = dataValues[index];
    int *currentIndexes = dataIndexes[index];
    for(int k = 1;k<=currentIndexes[0];k++)
    {
        cl.clusters[i][currentIndexes[k]] = currentValues[k];
    }
}
```

Binary Search is used here for **Faster Computations.**

➢ Clusters are made successfully now!

(ii) Use these clusters to clusterize feature-data(X):

```
float childrenInstancesCurrent [10][dimXProj];
for(int i=1;i<=reservoirSize;i++)
{
    indexCurrentClust = cl.indexBelong[i];
    for(int j=0;j<dimXProj;j++)
        currentX[j]=0;
    for(int j=1;j<=keyX[reservoirIndex[i]][0];j++)
    {
        currentX[keyX[reservoirIndex[i]][j]]= valuesX[reservoirIndex[i]][j];
    }
```

contd...

```
  ……..
  nbInst[indexCurrentClust]++;
  for(int j = 0;j<dimXProj;j++)
  {
      childrenInstancesCurrent[indexCurrentClust][j] = childrenInstancesCurrent[indexCurrentClust][j] + currentX[j];
   }
}
```

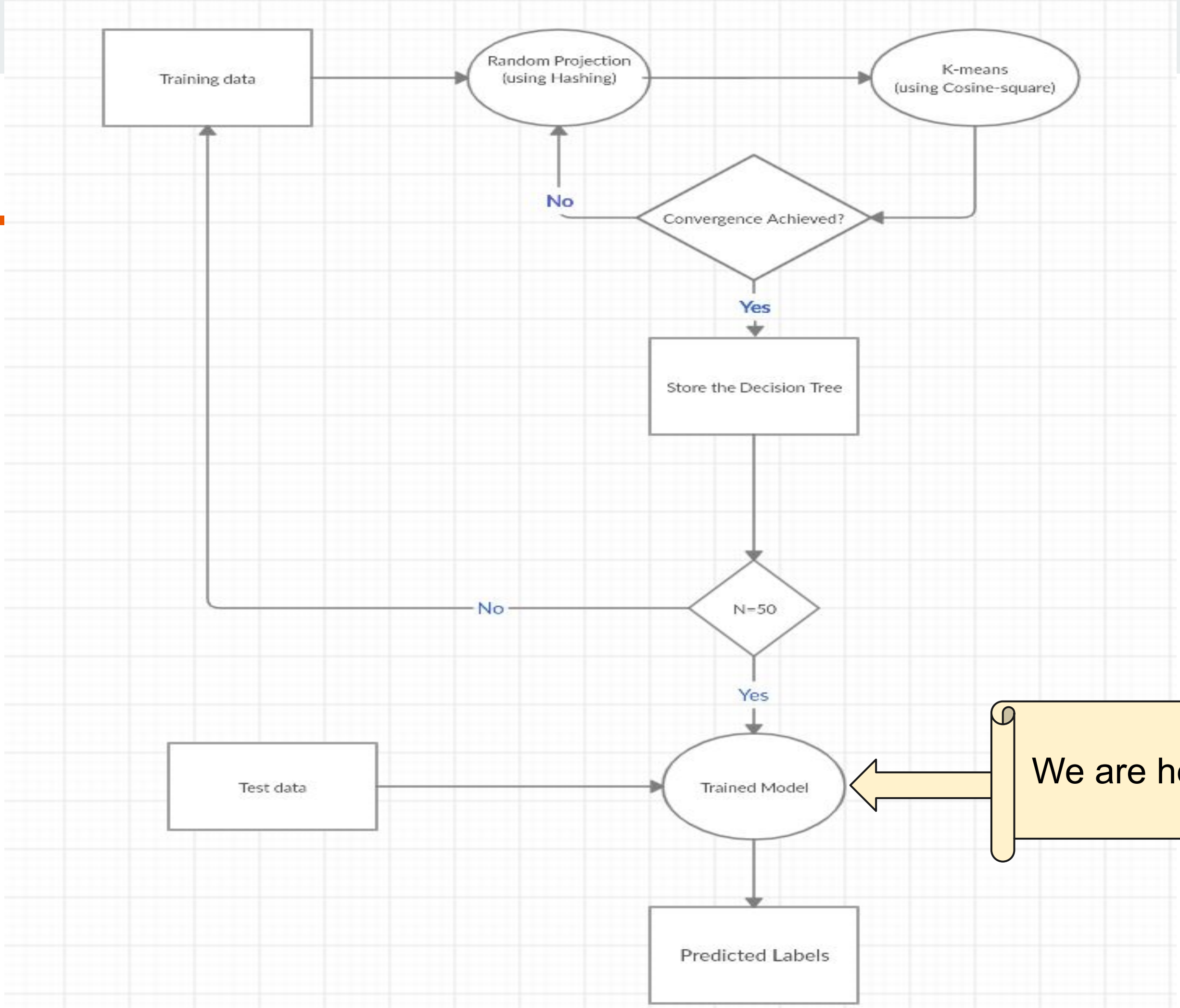
(iv)Conditions for leaf nodes:

- All features are same
- All labels are same
- No of instances is less than minInst(100).

```
if(queue_useInst[front][0]<d_args->minInst || checkSame(newValuesX,Y_val,queue_useInst[front]))
{
        .....
}
```

What does it store?

- Average of cumulative sum of label values.



We are here now!

4.Predicting the labels:

- Preprocess test data .
  - Hash32->Dimensionality Reduction->Normalise Data.

- Run it on trained model.
  - Using Cosine Distance, traversed the tree.

- Predict labels.
  - For each instances in the test data set we predicted using random forest classifier and took the top 5 labels.

Challenges...

1.Length of Data Handled Separately:

- Why?
    - Sparse data (can’t handle in static data structure (fixed length)).
    - Could not use STL in order to support parallelization in CUDA.

- How?
    - First value in each row represents **length** of row.
    - Each row size incremented by one.

2.Tree Traversal:

- **BFS using queue** (of nodes) because CUDA kernel does not support recursion.

```
int *queue_useInst[1000];
struct Node *queue_Node[1000];
int front=0,rear=0;
```

3.Memory Management:

- All temporarily allocated memory using ‘new’ is deleted explicitly using ‘delete’ which **provides visible difference** while execution.

  For instance:

  ```
  float *currentValues = new float[(int)dataValues[index][0]+1];
  int *currentIndexes = new int[dataIndexes[index][0]+1];
  ….
  delete currentValues;
  delete currentIndexes;
  ```

4. Parallelizing the Code:

1.Using **Threads.**

```
th1[i] = thread(buildTree, args, seedX, seedY, tree_no, root[i]);
```

5.Performance Analysis:

- Time Consumed(for 20 trees):

	Time(in sec) in C++	Time(in sec) in Java
1.Serial Code	73.24	-
2.Parallelized Code	19.69	21.23

- Performance Measured:

	Precision(1label)	Precision(3labels)	Precision(5 labels)
1. Java Code	73.09	56.31	45.68
2. C++ Code	72.49	56.17	45.82

Thank You