Graph Methods for Particle Tracking

Varun Sreenivasan, University of Wisconsin-Madison

Mentor: Daniel Murnane, Lawrence Berkeley National Laboratory

Particle Track Reconstruction

- In experiments such as ATLAS and CMS at the High Luminosity Large Hadron Collider (HL-LHC), giant particle detectors will collect measurements from 200 particle interactions per collision event on average.
- Reconstruction of charged particle trajectories in high granularity tracking detectors is a critical component of the data analysis pipeline in HEP.
- The Exa.Trkx project has been investigating machine learning solutions to solve this problem, namely Graph Neural Networks.

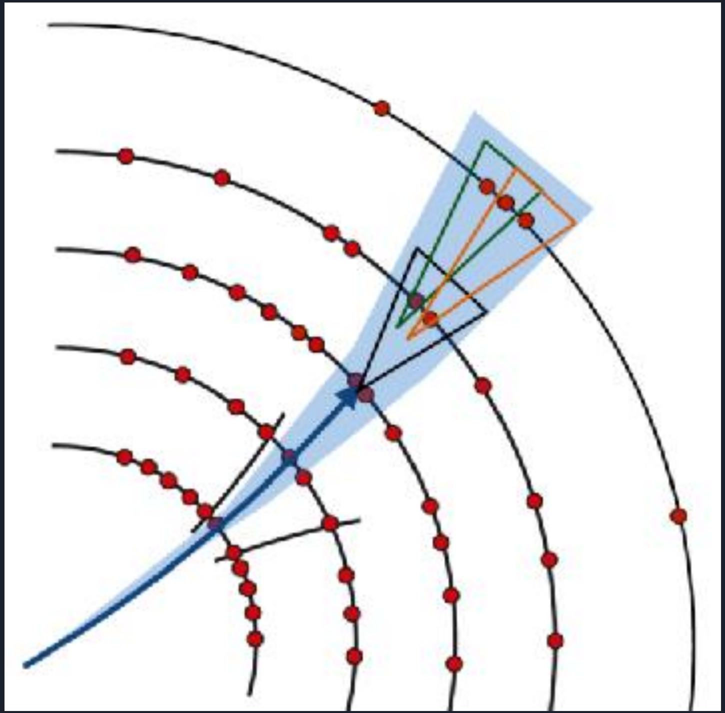


Figure: Track Reconstruction

Graph Neural Networks

- Graphs are constructed from the point cloud of hits in each event.
- Edges are drawn between hits that may come from the same particle according to a heuristic.
- The GNN model is then trained to classify the graph edges as real or fake, giving a pure and efficient sample of track segments which can be used to construct full track candidates.

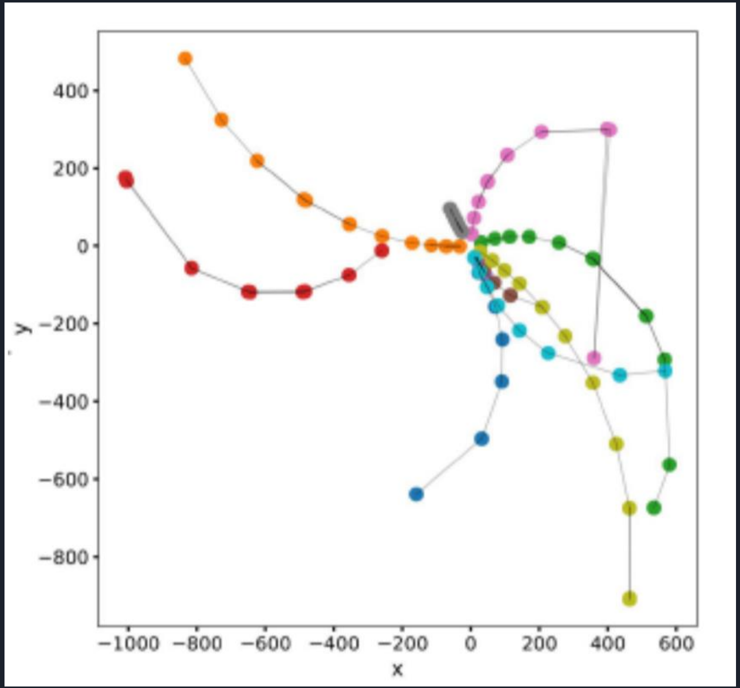


Figure: Graph Construction

TrackML CodaLab Dataset

- 9000 Events
- 100,000 hits from around 10,000 particles
- Includes noise

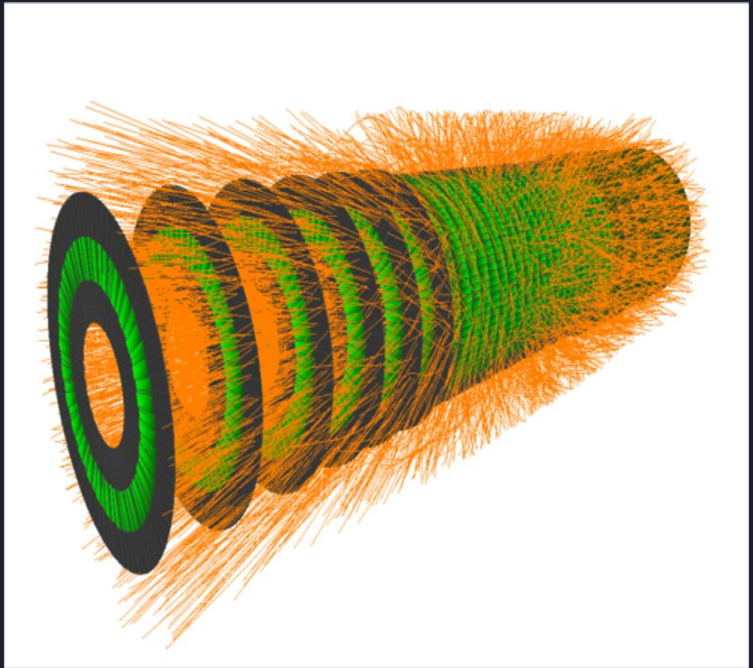


Figure: Simulated HL-LHC collision event

TrackML Pipeline: Processing

- Calculate cell information (ci) features that provide useful information about the direction of the charged particle.

- Construct truth graphs (true edges connecting hits from the same particle) to be used in training.

- Generate events to be used for the embedding stage.

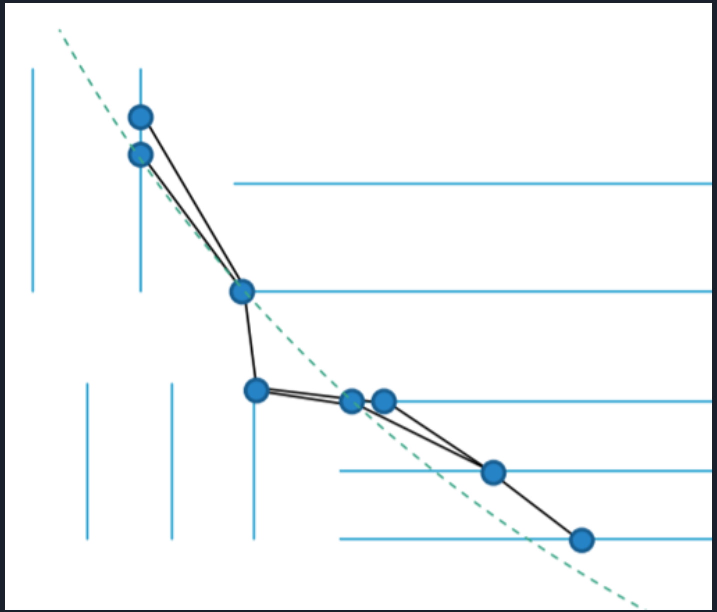


Figure: Truth Graph

TrackML Pipeline: Embedding

- Learn a distance metric for hit measurements embedded into a new euclidean space where d is low enough that the embedded space is not too sparse.

- Use Nearest-Neighbors algorithm to create predicted graph

- Graphs constructed are then passed to the filtering stage.

TrackML Pipeline: Embedding

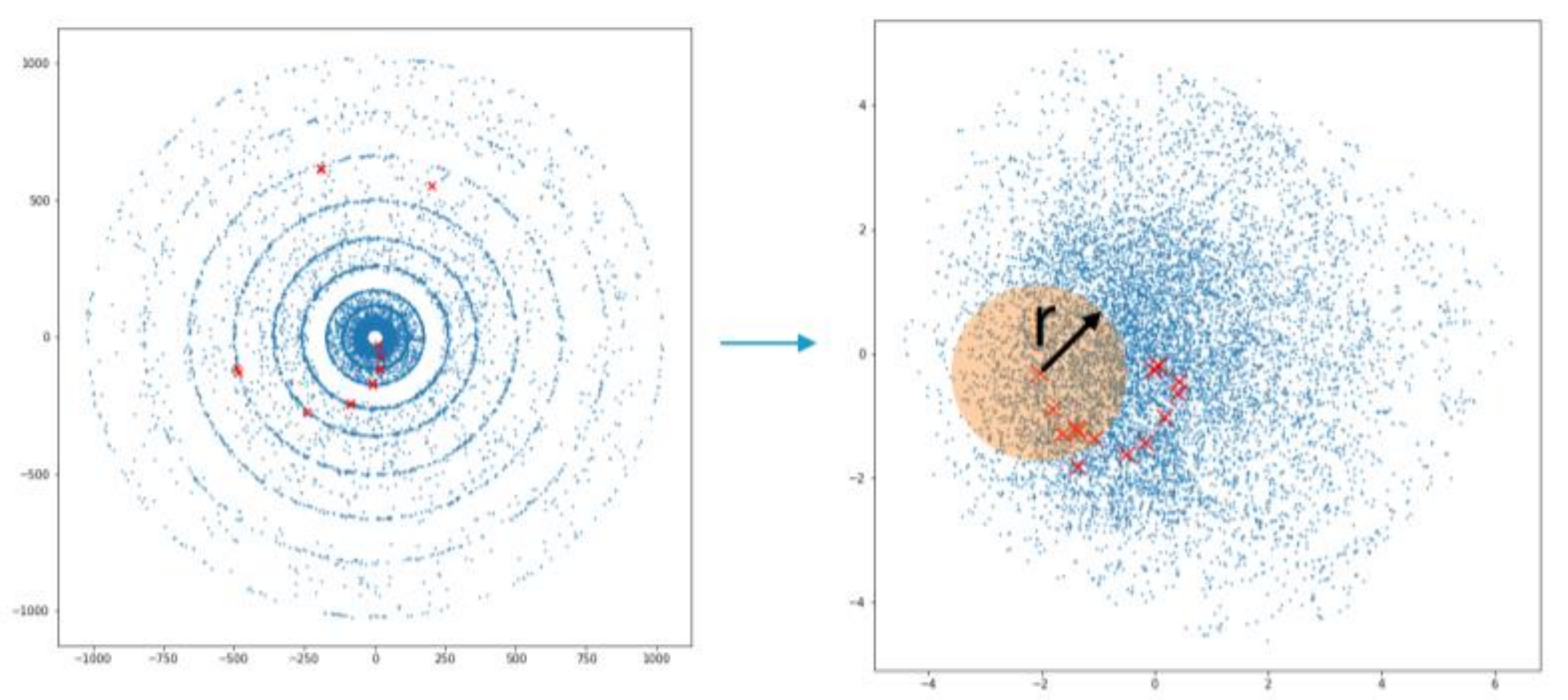


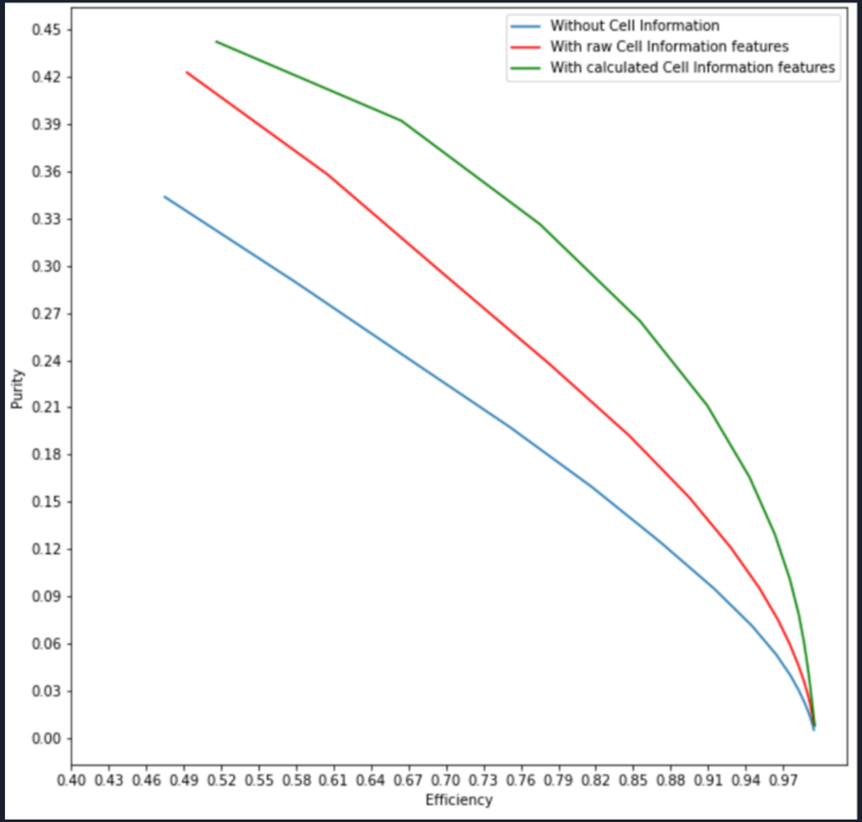
Figure: Embedding

Processing: Cell Information (CI) features

- Cell Count: number of channels in a cluster
- Cell Val: charge deposited
- lx, ly, lz: the vector of charge deposited through the module in local coordinates
- leta, lphi: vector in local cylindrical coordinates
- geta, gphi: vector in global cylindrical coordinates

Processing: Investigating importance of CI

- Understand the importance of the cell information features while training an embedding model.
- Compare three trained models: one without cell information, one with raw cell information, and the other with calculated cell information.
- Result demonstrated through an efficiency-purity graph where both metrics vary as radius changes.



Embedding

- Update build_edges function by integrating FRNN to replace Faiss.
- Determine the appropriate loss function.
- Study the impact of weighting edges.
- Perform Hyperparameter scanning to obtain optimal results
- Train MLP

Build Edges

This function calls the appropriate Nearest Neighbor algorithm to generate edges and build the graph.

Input

1) Query points: the set of hits for which we want to query the database of points and generate edges for.

1) Database points: the set of all hits in the event.

1) r_max: the search radius within which we want to generate edges.

1) K_max: the number of neighbors we want to explore in the search radius

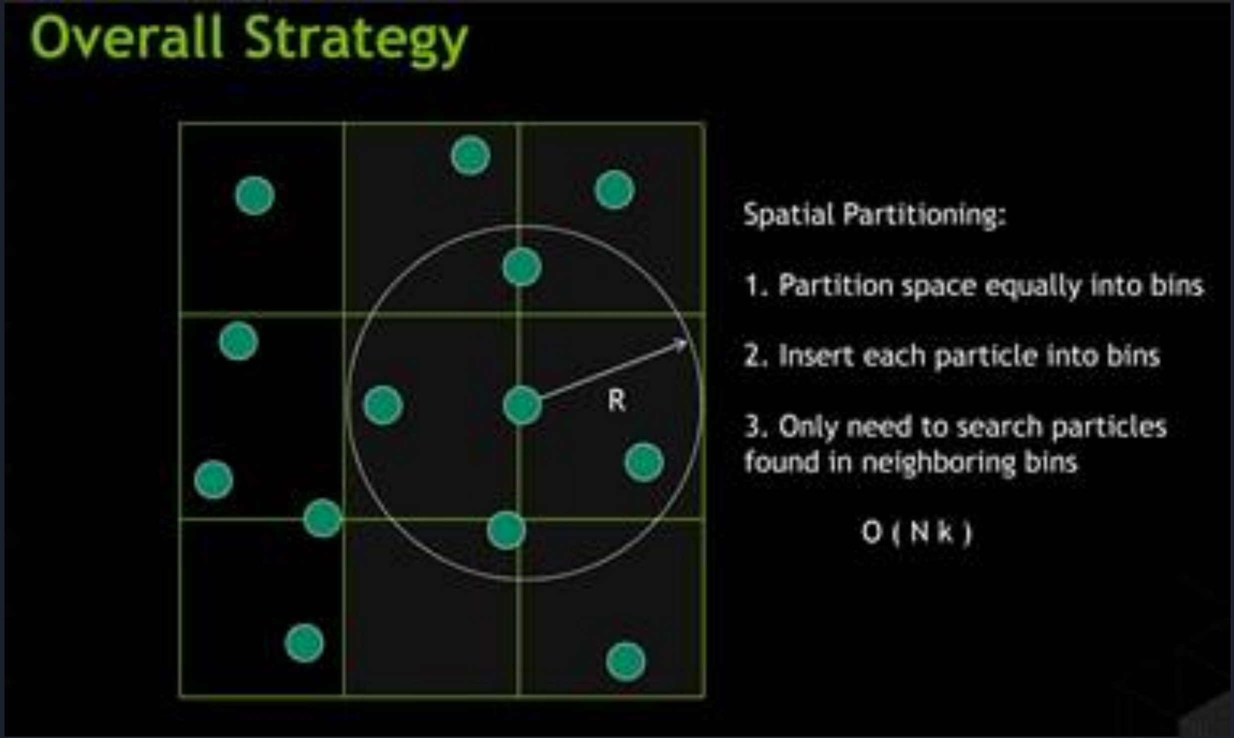
Output

1) The graph

Fixed Radius Nearest Neighbors

- https://github.com/lxxue/FRNN
- A Fixed Radius Nearest Neighbors search implemented on CUDA.
- Provides significant speedup over Facebook's Faiss and Pytorch3D KNN without sacrificing efficiency and purity.
- Integrated into the function build_edges to find the neighbors for query points from points in the database. This is integral for performing hard negative mining (adding negative samples to the training set) while training.
- Performed a couple of tweaks to further speedup the algorithm. Changes include not performing KNN and sorting.
- Benchmarking results show that the updated FRNN is over 10x faster than Faiss.

Glimpse into the FRNN Algorithm

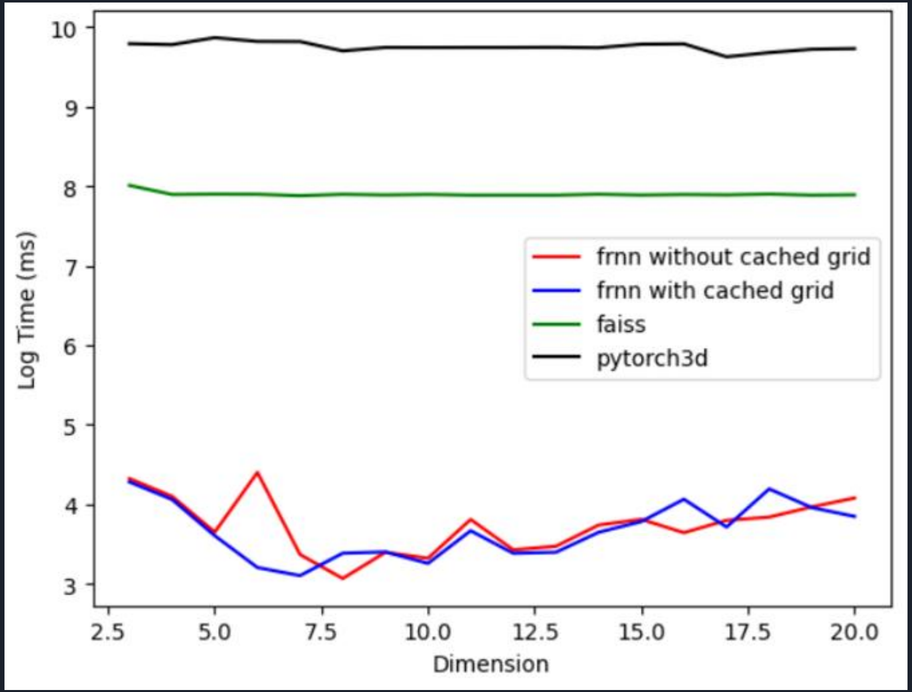


Benchmarking Sample (Time vs D)

N = 100000

K = 1500

r = 1.0



Determining Max K for FRNN

- Get an accurate estimate of the number of neighbors to explore in the specified radius.
- We want this value because this is a required input for the FRNN algorithm.
- To achieve high efficiency, it is important to have a Max-K value that is greater than or equal to the number of neighbors for a point in the densest region.
- Can’t arbitrarily set to a high value due to memory constraints.
- Explored KD-Tree and DBSCAN algorithms to do this. Finally, settled on using the FRNN grid to determine the Max-K value.

KD-Tree

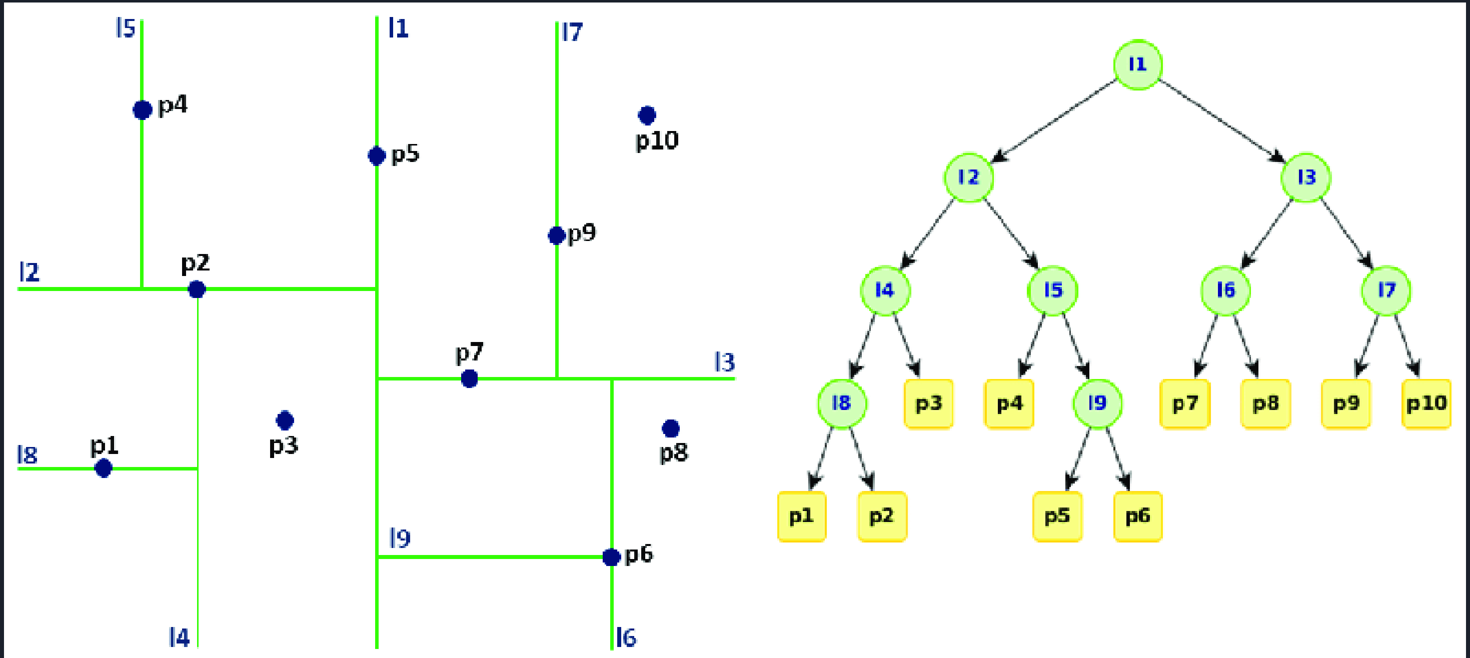


Figure: KD-Tree

DBSCAN

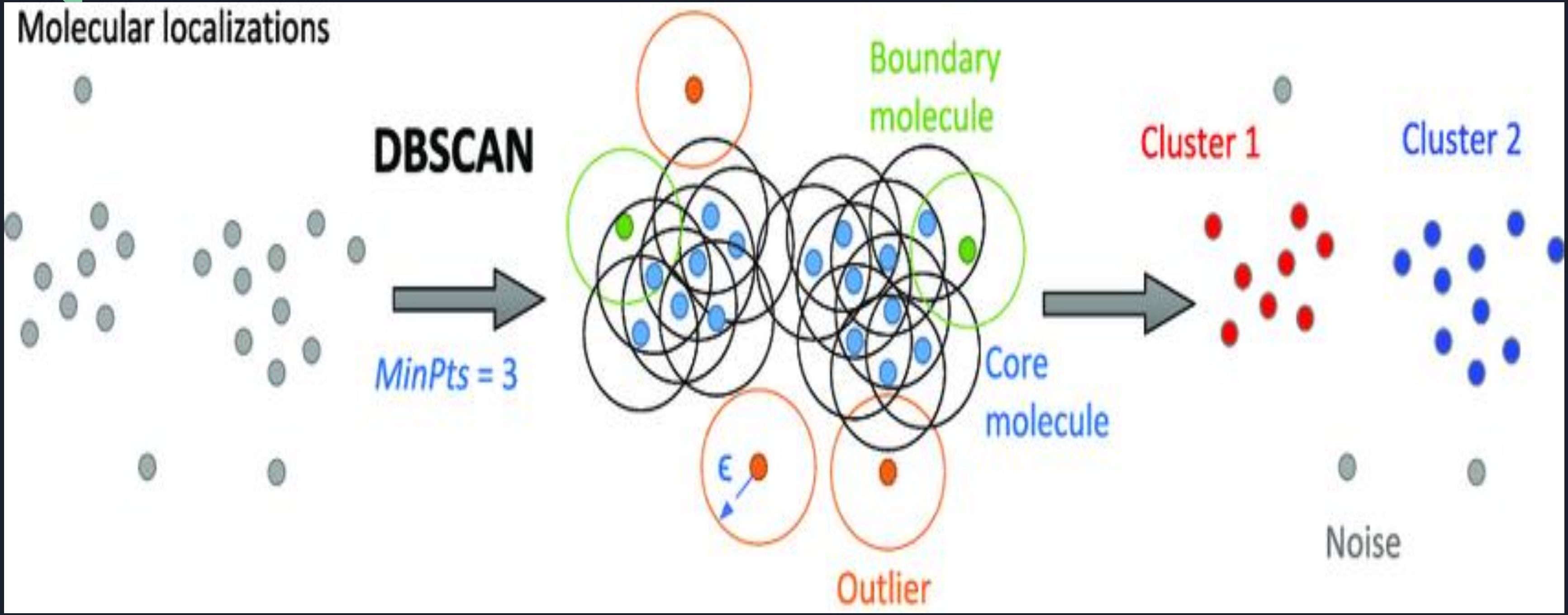


Figure: DBSCAN

FRNN Grid Solution

- Use Grid Count to determine the densest cell.
- Identify points belonging to the densest cell.
- Run nearest neighbors algorithm on each of the points in this cell.
- The Max-K value is the maximum number of edges generated for a specific point

Determining the appropriate loss function

- For positive examples (true edges), we want to ensure to minimum distance between embedded hits. So penalization increases as distance between these hits increases.

- For negative examples (false edges), we want to maximize the distance between embedded hits. So penalization increases as distance between these hits decreases.

- We explore two loss functions: Binary Cross Entropy loss and Hinge Embedding loss.

Determining the appropriate loss function

Hinge Embedding Loss:

Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1). This is usually used for measuring whether two inputs are similar or dissimilar, e.g. using the L1 pairwise distance as x, and is typically used for learning nonlinear embeddings or semi-supervised learning.

The loss function for n-th sample in the mini-batch is

$$l_n = \begin{cases} x_n, & \text{if } y_n = 1, \\ \max\{0, \Delta - x_n\}, & \text{if } y_n = -1, \end{cases}$$

and the total loss functions is

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'.} \end{cases}$$

where $L = \{l_1, \dots, l_N\}^\top$.

Determining the appropriate loss function

Binary Cross Entropy Loss

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \operatorname{mean}(L), & \text{if reduction} = \text{`mean';} \\ \operatorname{sum}(L), & \text{if reduction} = \text{`sum'.} \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets y should be numbers between 0 and 1.

Result

- The best loss function was the Hinge Embedding Loss without application of square root.

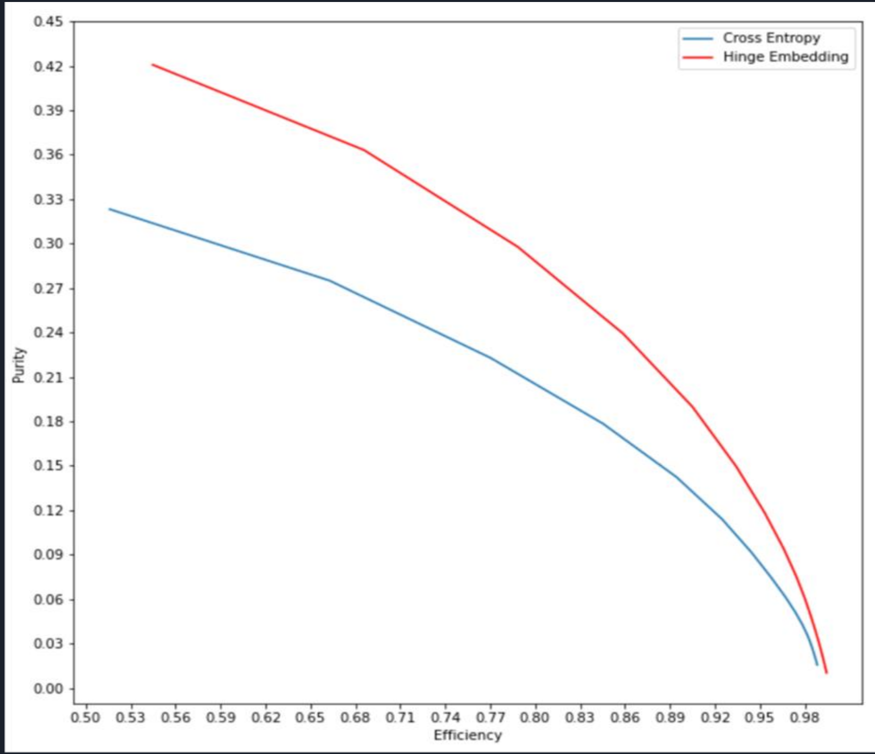
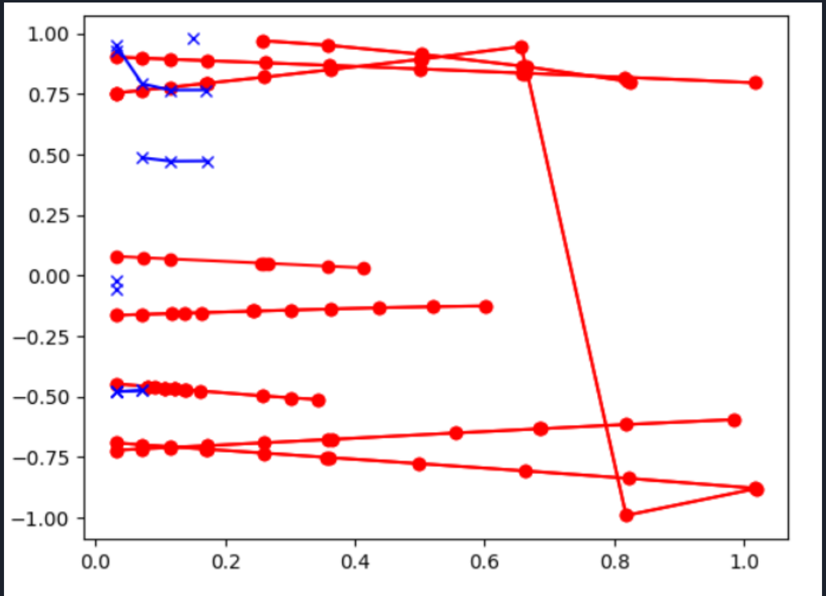


Figure: Loss Function Evaluation

Weighting

- The weighting functionality present in the embedding procedure is used to prioritize more important tracks.
- For the evaluation, the vanilla efficiency and purity metrics are replaced with weighted efficiency and purity.
- High weight tracks include more hits and are longer than low weight tracks.

Red: High Weight, Blue: Low Weights



Weighted Efficiency and Purity

Weighted Efficiency = sum (y * pred_weights) / sum (true_weights)

Weighted Purity = sum (y * pred_weights) / sum (pred_weights)

true_weights = ½ * (S_i + S_j)

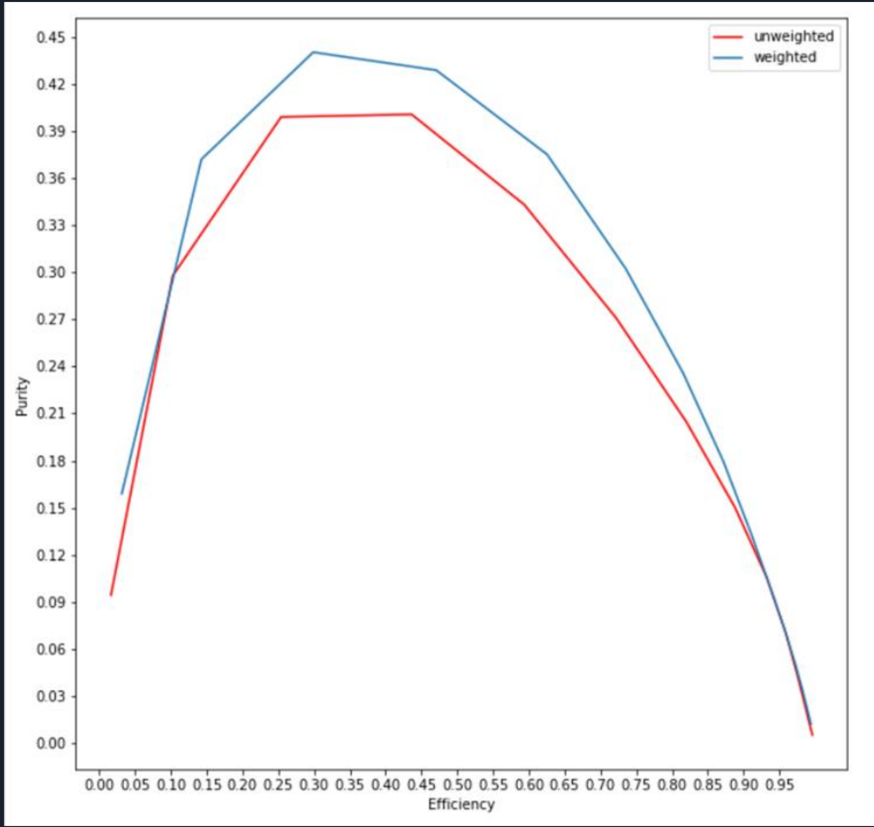
S_k: Spacepoint k weight

pred_weights = ½ * (S_i + S_j), if y_{ij} = 1

= 1 if y_{ij} = 0

Weighted Model vs Unweighted Model

- Comparison of models trained with weighting and without weighting.
- Analysis over weighted efficiency and Weighted purity.
- Model trained with weighting performs better across the board.



Hyperparameter Scanning

- Study the impact of varying parameters such as number of hidden layers, number of hidden nodes, etc.

- Integrated TrainTrack library to iterate over multiple hyperparameters and generate models in a serial and trackable way.

- Doing this helps us obtain the optimal hyperparameters to create the best model possible.

- TrainTrack: https://github.com/murnanedaniel/train-track

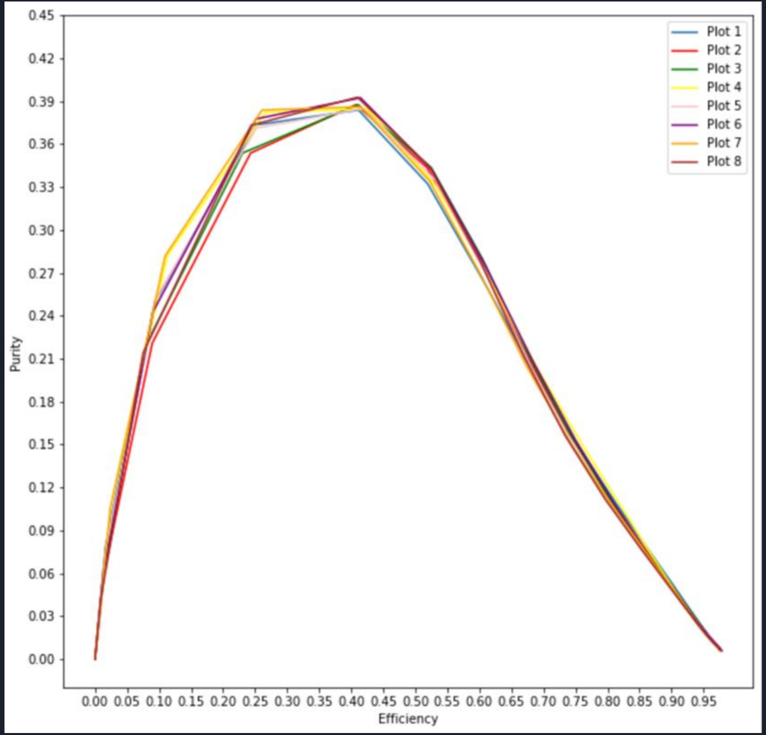


Figure: Scan Plot

MLP Architecture

- Input Channels: 12
- Hidden Layers: 6
- Hidden Nodes: 1024
- Embedding Dimension: 8
- Activation Function: Tanh - applied after every linear transformation