Distance Metrics for KNN

Algorithmic variations of KNN that modify how distance is used or how neighbors are found, especially for efficiency or improved accuracy:

1. **Weighted KNN:**
    - **Idea:** Instead of giving all K neighbors an equal vote (in classification) or equal contribution (in regression), this variant assigns **weights** to the neighbors based on their distance.
    - **How Distance is Used:** Closer neighbors get higher weights, meaning they have more influence on the final prediction. A common weighting scheme is the inverse of the distance (e.g., weight = 1 / distance).
    - **Benefit:** Reduces the impact of potentially noisy or less relevant neighbors that happen to be within the K set but are relatively far away compared to closer neighbors.
2. **KNN with Optimized Search Structures (KD-Trees / Ball Trees):**
    - **Idea:** Standard KNN requires calculating the distance from the query point to *every single point* in the training set, which is very slow for large datasets (brute-force search). These variants use tree-based data structures to organize the training data.
    - **How Distance is Used:** The tree structures (KD-Trees for lower dimensions, Ball Trees often better for higher dimensions) allow the algorithm to quickly prune large portions of the search space, drastically reducing the number of actual distance calculations needed to find the true K nearest neighbors according to the chosen metric (e.g., Euclidean).
    - **Benefit:** Significantly speeds up the prediction phase (neighbor search) without changing the final result (still finds the exact K nearest neighbors based on the chosen distance metric).

KD-Trees (K-Dimensional Trees)

1. **What it is:** A binary tree structure that recursively partitions the k-dimensional space using axis-aligned hyperplanes. Think of it as repeatedly splitting the data space into two halves along one dimension at a time.

2. **Construction:**
    - **Select Axis:** Choose a dimension (axis) to split along. Common strategies include cycling through dimensions (x, y, z, x, y, z, ...) at each level of the tree or choosing the dimension with the highest variance among the points currently being considered.
    - **Find Median:** Find the median point along the selected axis within the current subset of data points.

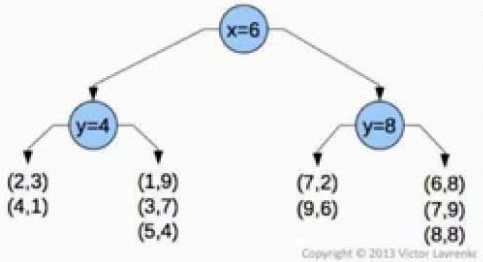
    - **Partition:** Divide the current subset of points into two groups: those whose coordinate value along the chosen axis is less than the median's value, and those whose value is greater than or equal to the median's value.
    - **Create Node:** The median point becomes the current node in the tree. Store the splitting dimension and the median value at this node.
    - **Recurse:** Recursively apply steps 1-4 to the left subset (points less than the median) and the right subset (points greater than or equal to the median), creating the left and right children of the current node. Stop when a node contains fewer than a specified number of points (leaf size).
3. **Structure:** Each internal node represents a split along a specific dimension at a specific value (the median). The leaves of the tree contain the actual data points (or small buckets of points). The tree effectively divides the entire space into smaller rectangular regions.

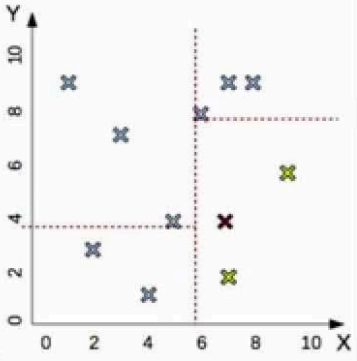
4. **Querying (Finding Nearest Neighbors):**
    - **Traversal Down:** Start at the root. Compare the query point's coordinate along the node's splitting dimension with the node's median value. Move down to the left child if the query point's value is smaller, or to the right child if it's larger/equal. Repeat until a leaf node is reached.
    - **Initial Check & Distance:** Calculate distances from the query point to all points in this leaf node. Keep track of the K closest points found so far and the distance to the farthest of these (`d_k`).
    - **Backtracking & Pruning:** Move back up the tree. At each node visited during backtracking:
        - Check the *other* subtree (the one not initially visited on the way down).
        - **Pruning Condition:** Calculate the shortest distance from the query point to the *hyperplane* defined by the current node's split. If this distance is *less than* `d_k` (the distance to the current Kth nearest neighbor), then the other subtree *could potentially* contain closer points.
        - If the other branch might contain closer points, traverse down that branch, updating the list of K nearest neighbors and `d_k` if necessary.
        - If the distance to the splitting hyperplane is greater than or equal to `d_k`, the entire other branch can be safely **pruned** (ignored), as no point within it can be closer than the Kth neighbor already found.
    - Continue backtracking until the root is processed. The final list contains the K nearest neighbors.
5. **Pros & Cons:**

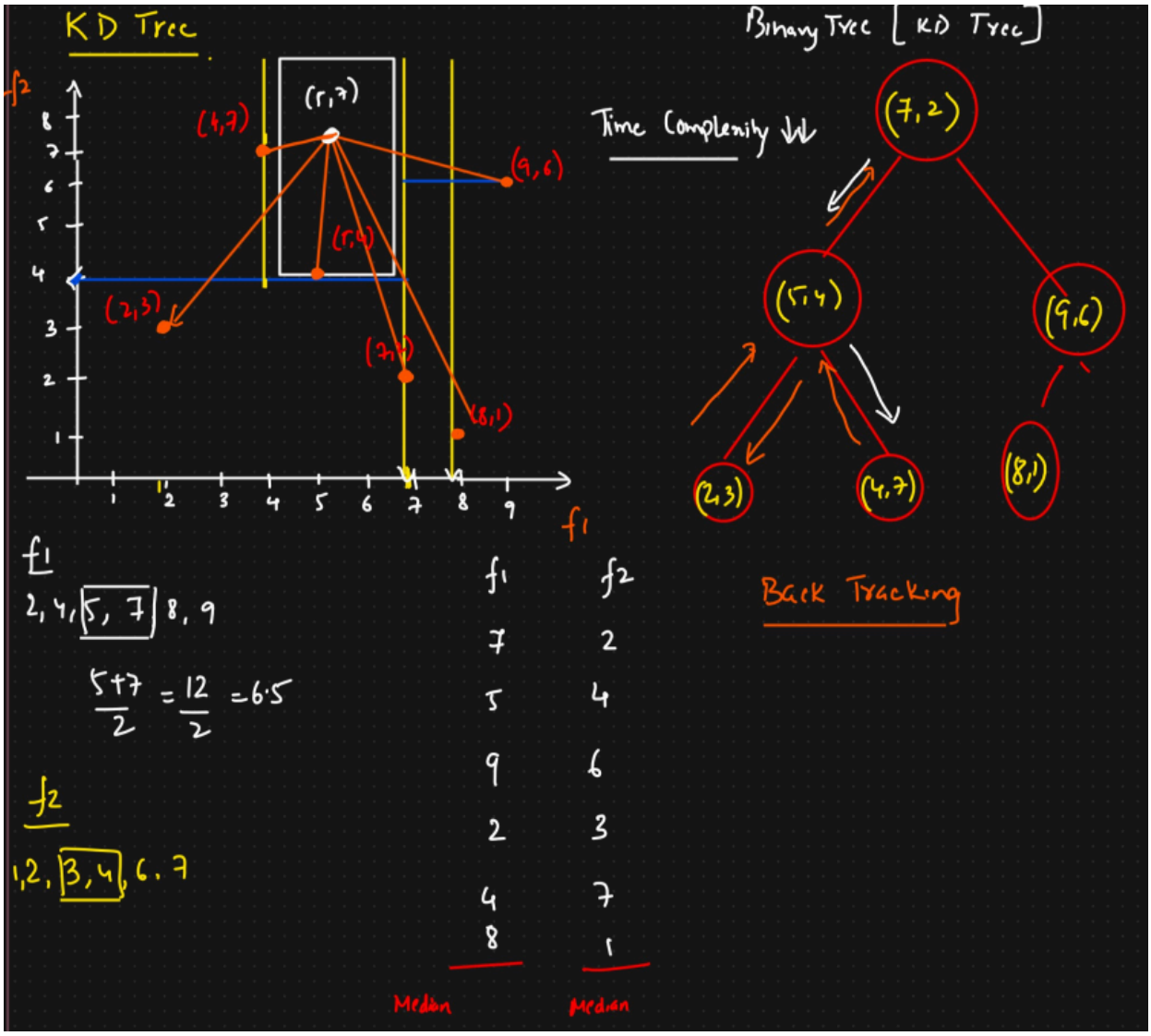
    - **Pros:** Significantly faster than brute-force search, especially in low to moderate dimensions (e.g., up to ~20 dimensions). Relatively simple concept.
    - **Cons:** Performance degrades rapidly as dimensionality increases (the "curse of dimensionality"). Axis-aligned splits become inefficient, and most branches might need to be explored, approaching brute-force performance. Can be sensitive to the data distribution.

K-D tree example

- Building a K-D tree from training data:
  - {(1,9), (2,3), (4,1), (3,7), (5,4), (6,8), (7,2), (8,8), (7,9), (9,6)}
  - pick random dimension, find median, split data, repeat
- Find NNs for new point (7,4)
  - find region containing (7,4)
  - compare to all points in region







Ball Trees

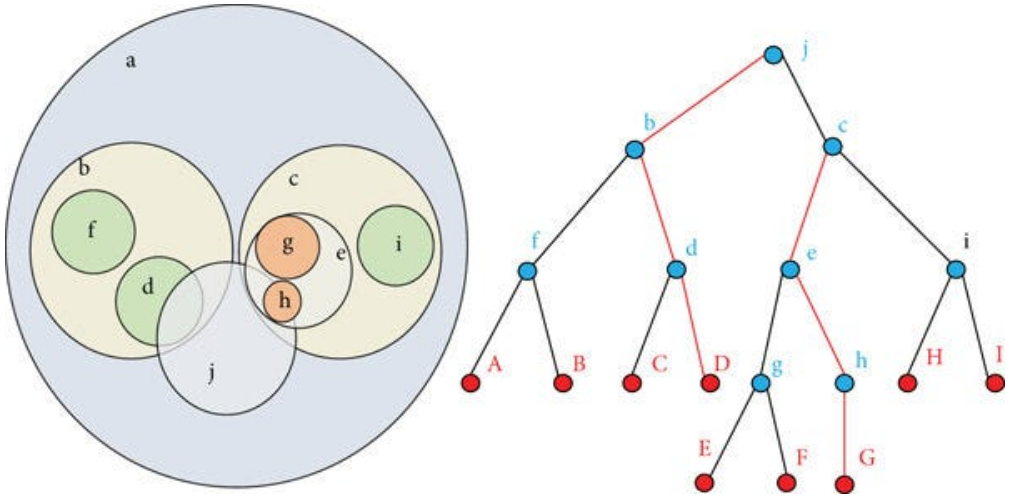
3. **What it is:** A tree structure that partitions data points into nested hyperspheres (or "balls"), rather than axis-aligned boxes.

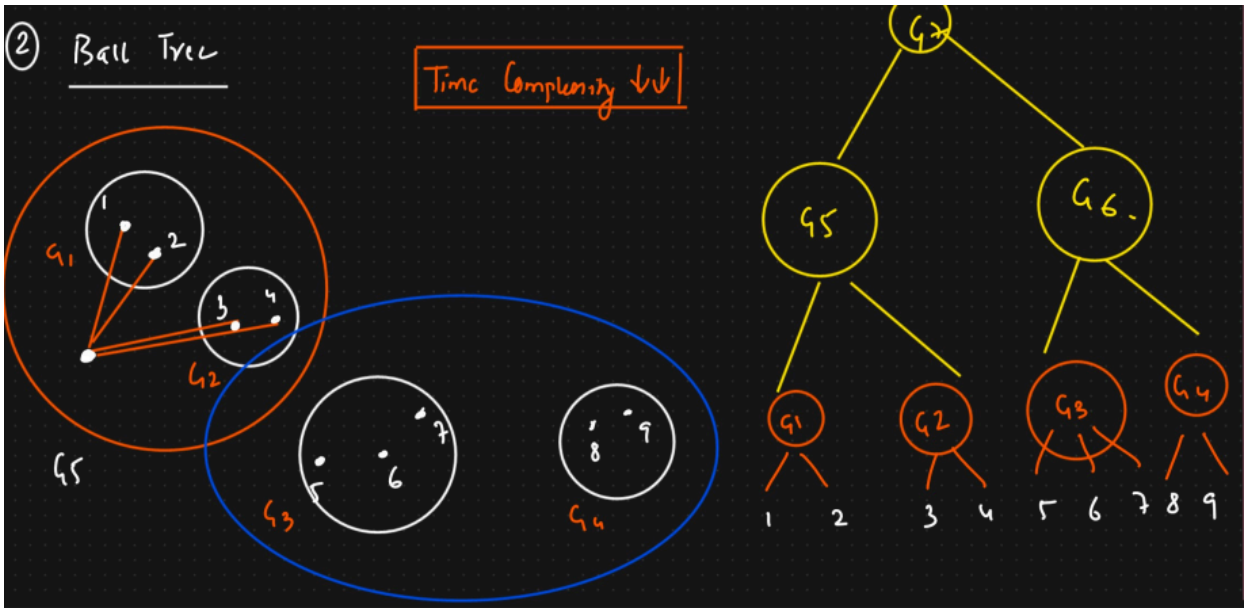
4. **Construction:**
    - **Root Ball:** Start with a single node representing a hypersphere that tightly bounds *all* data points in the training set. This ball is defined by a center point and a radius.
    - **Partition:** Choose a method to split the points within the current ball into two disjoint sets. Common methods include:
        i. Selecting the two points furthest apart within the ball and assigning other points to the closer of these two "pivot" points.
        ii. Performing a quick clustering (like k-means with k=2) to divide the points.

    - **Create Children:** Create two child nodes. For each child node, calculate the center and radius of a new, smaller hypersphere that tightly bounds only the points assigned to that child.
    - **Recurse:** Recursively apply steps 2-3 to the child nodes until the number of points within a ball falls below a specified leaf size.
5. **Structure:** Each node in the tree represents a hypersphere (defined by center and radius) containing all the data points in its subtree. Leaf nodes contain the actual data points.

6. **Querying (Finding Nearest Neighbors):**
    - **Traversal Down:** Start at the root. At each node, calculate the distance from the query point to the *centers* of its two child balls. Prioritize descending into the child ball whose center is closer to the query point.
    - **Initial Check & Distance:** When a leaf node is reached, calculate the distances from the query point to all points within that leaf. Maintain a list of the K closest points found so far and the distance to the farthest one (`d_k`).
    - **Backtracking & Pruning:** Move back up the tree. At each node visited during backtracking:
        i. Consider the *other* child ball (the one not initially chosen or fully explored yet).
        ii. **Pruning Condition:** Calculate the shortest distance from the query point to the *surface* of this other ball (distance to center minus radius). If this distance is *less than* `d_k`, then the other ball *might* contain points closer than the current Kth neighbor, and it must be explored.
        iii. If the distance to the surface of the other ball is greater than or equal to `d_k`, then the entire subtree rooted at that ball can be **pruned**, as no point within it can be closer than the Kth neighbor found so far. This relies on the triangle inequality property.
    - Continue backtracking and exploring promising branches until the root is processed.
7. **Pros & Cons:**

    - **Pros:** Generally performs better than KD-Trees in higher dimensions because the hypersphere partitioning can be more effective for clustered data, regardless of axis alignment. Less sensitive to data distribution than KD-trees.
    - **Cons:** Construction and querying can involve slightly more complex calculations (dealing with sphere boundaries). May have slightly higher overhead than KD-Trees in very low dimensions.





8. **Approximate Nearest Neighbors (ANN):**
    - **Idea:** For extremely large datasets or very high dimensions, even tree-based searches can become slow. ANN techniques prioritize speed over finding the *exact* K nearest neighbors.
    - **How Distance is Used:** Methods like Locality-Sensitive Hashing (LSH) group similar points together probabilistically. When searching for neighbors, the algorithm only checks points within the same "hash bucket(s)" as the query point, which is much faster. It finds points that are *likely* to be among the nearest neighbors.
    - **Benefit:** Much faster search, especially in high dimensions, at the cost of potentially missing some of the true nearest neighbors (hence, "approximate").