The K-Nearest Neighbors Algorithm

5th February 2026

Computerpraktikum Teil 2

Presentation Overview

- **Problem Statement**
- **Code Structure**
- **Ball Tree**
    - Constructed Recursively
    - Constructed Iteratively
- **Speed Ups**
- **Error Reduction Strategies**
- **References**

Problem Statement

Problem Statement

- Implement a method for **binary classification** in Python
- Given labeled data

$$D = \{(y_i, x_i)\}_{i=1}^n,\ y_i \in \{-1, 1\},\ x_i \in [-1, 1]^d$$

- Learn a classifier $f_D : [-1, 1]^d \to \{-1, 1\}$
- Goal: **minimize the misclassification rate** on unseen test data D'
- Approach: k-**nearest neighbors** with cross validation to select k^*

Code Structure

classify.py

- **Setup**
  - Import standard libraries
  - Custom `BallTree` class for nearest neighbor search
- **Data Loading**
  - `load_data`: reads CSV files and stores samples as (y, x) pairs (label and feature vector)
- **Cross Validation**
  - `run_cross_validation`: performs l-fold cross validation
  - Evaluates $k = 1, \ldots, K_{\max}$ using cumulative majority voting
  - Computes fold-wise and averaged validation error rates
- **Training**
  - Selection of optimal k^* based on minimal validation error
  - Construction of an ensemble of BallTrees
- **Testing & Output**
  - Ensemble voting for final predictions on the test set
  - Writes predictions, logs, and error curves to disk
  - Optional visualization for 2D datasets

ball_tree.py

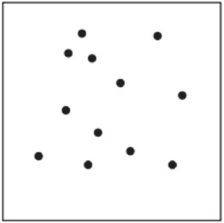
- **Tree Construction**
  - Data is recursively partitioned into hyperspherical regions (balls)
  - Each node stores a center and radius enclosing its data
  - Leaf nodes store the actual data points
  - Tree is built iteratively using a stack (no recursion)
- **Distance Optimization**
  - Precomputation of point norms $\|x\|^2$ for fast distance evaluation
  - Uses the identity $\|x - y\|^2 = \|x\|^2 - 2\langle x, y \rangle + \|y\|^2$
- **Query Algorithm**
  - Best-first search using a priority queue ordered by distance to node centers
  - Local and global pruning based on node radius and current worst neighbor
  - Early termination when no closer neighbors are possible

Speed Ups

Error Reduction Strategies

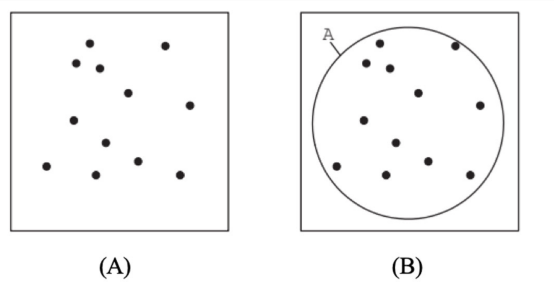
Ball Tree

Ball Tree: Konzept



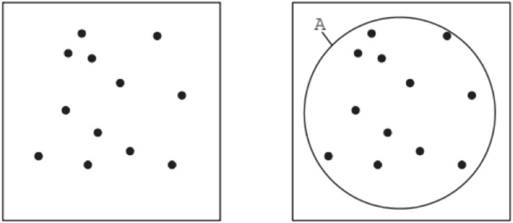
(A)

Ball Tree: Konzept



An jedem Knoten:

Ball Tree: Konzept



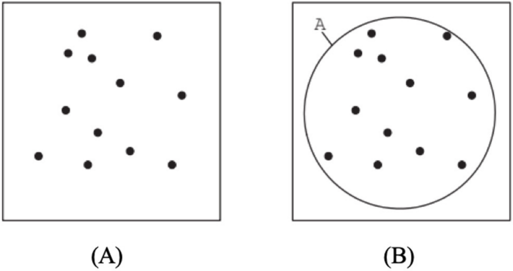
(A)

(B)

An jedem Knoten:

1. Mittelpunkt, Radius berechnen

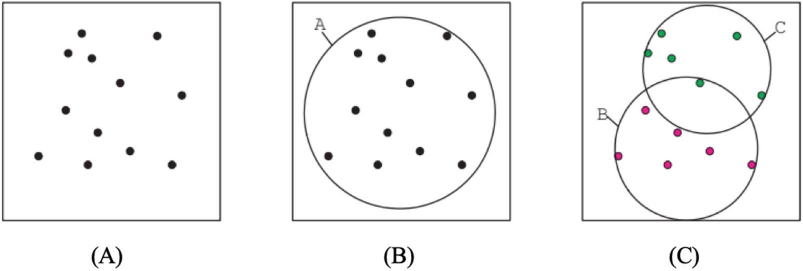
Ball Tree: Konzept



An jedem Knoten:

1. Mittelpunkt, Radius berechnen
2. Zwei entfernte Punkte wählen

Ball Tree: Konzept

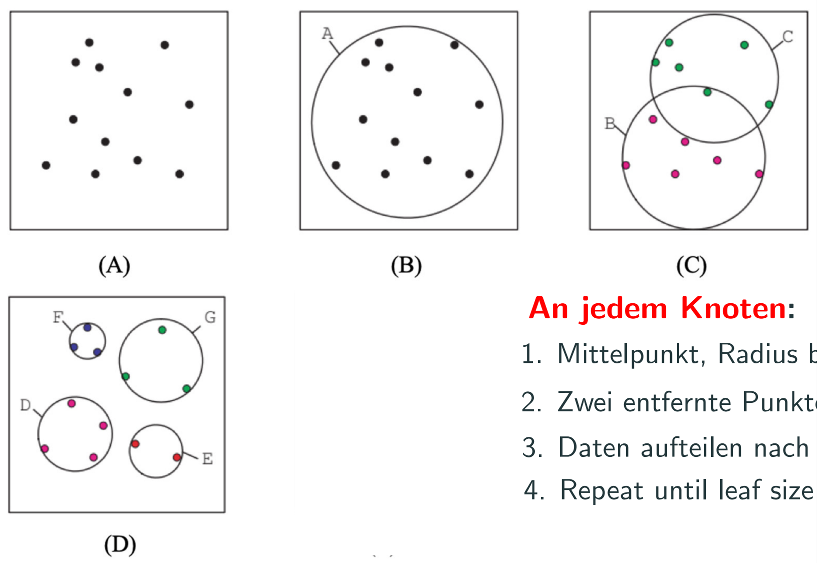


(A) (B) (C)

An jedem Knoten:

1. Mittelpunkt, Radius berechnen
2. Zwei entfernte Punkte wählen
3. Daten aufteilen nach Distanz

Ball Tree: Konzept



(A)

(B)

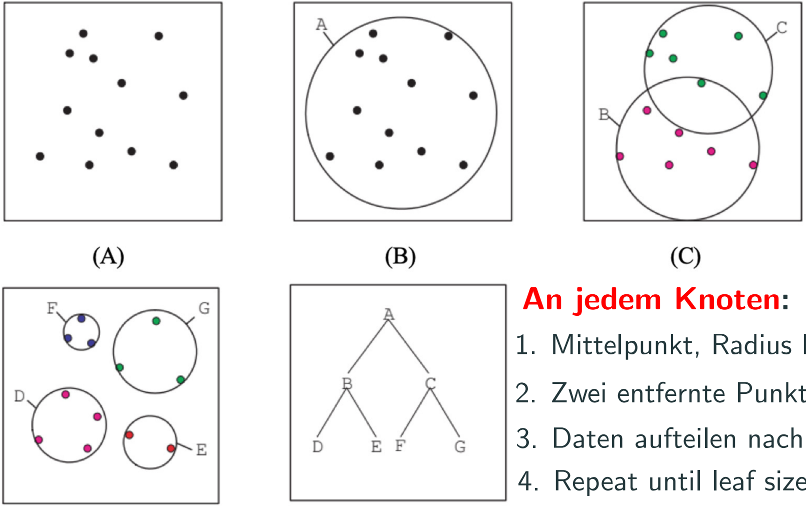
(C)

(D)

An jedem Knoten:

1. Mittelpunkt, Radius berechnen
2. Zwei entfernte Punkte wählen
3. Daten aufteilen nach Distanz
4. Repeat until leaf size reached

Ball Tree: Konzept



(A) (B) (C)

(D) (E)

An jedem Knoten:

1. Mittelpunkt, Radius berechnen
2. Zwei entfernte Punkte wählen
3. Daten aufteilen nach Distanz
4. Repeat until leaf size reached

Leaf nodes store data points

Ball Tree: Rekursiv

```
def BallTree:

    def __init__(self, data, leaf_size=1):
        self.leaf_size = leaf_size
        self.points = data
        self.left = None
        self.right = None
        self.center = self._computer_center(data)
        self.radius = self._compute_radius(data, self.center)
```

Ball Tree: Rekursiv

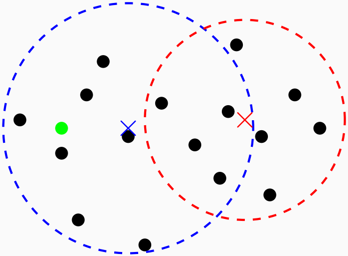
```
def BallTree:

    def __init__(self, data, leaf_size=1):
        self.leaf_size = leaf_size
        self.points = data
        self.left = None
        self.right = None
        self.center = self._computer_center(data)
        self.radius = self._compute_radius(data, self.center)

        if len(data) > self.leaf_size:
            self._split()
```

Ball Tree: Rekursiv

```
def BallTree:

    def __init__(self, data, leaf_size=1):
        self.leaf_size = leaf_size
        self.points = data
        self.left = None
        self.right = None
        self.center = self._computer_center(data)
        self.radius = self._compute_radius(data, self.center)

        if len(data) > self.leaf_size:
            self._split()

    def _split(self):

        ... # Split points

        self.left = BallTree(left_points, self.leaf_size)
```

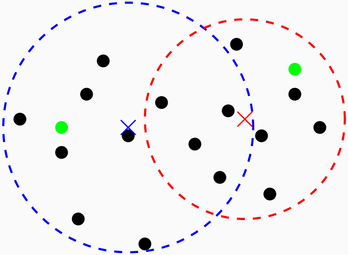
Ball Tree: Rekursiv

```
def BallTree:

    ...
```



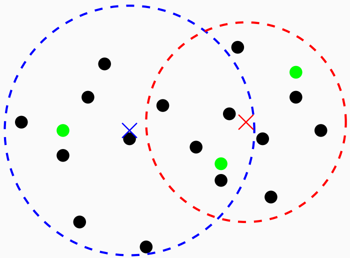
Ball Tree: Rekursiv

```
def BallTree:

    ...
```

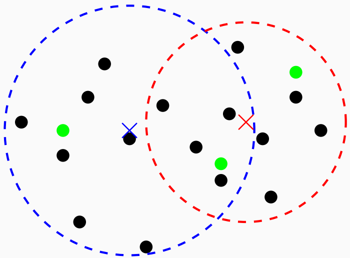


Ball Tree: Rekursiv

```
def BallTree:

    ...
```

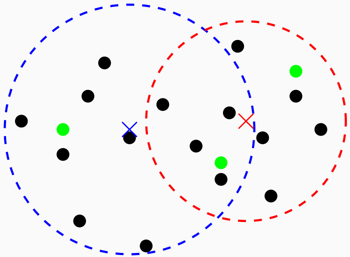


Ball Tree: Rekursiv



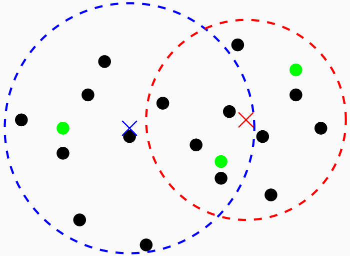
```
def BallTree:

    ...

    def query(self, x, k, knn=[]):
        lower_bound = max(0.0, distance(x, self.center) - self.radius)
```

Ball Tree: Rekursiv



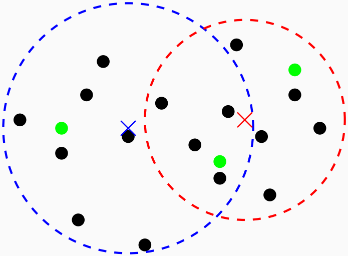
```
def BallTree:

    ...

    def query(self, x, k, knn=[]):
        lower_bound = max(0.0, distance(x, self.center) - self.radius)
        if len(knn) == k and lower_bound > knn[0][0]:
            return
```

Ball Tree: Rekursiv



```
def BallTree:

    ...

    def query(self, x, k, knn=[]):
        lower_bound = max(0.0, distance(x, self.center) - self.radius)
        if len(knn) == k and lower_bound > knn[0][0]:
            return

        if self.left is None and self.right is None:

            ... # Pointsearch in leaf node
```

Ball Tree: Rekursiv



```
def BallTree:

    ...

    def query(self, x, k, knn=[]):
        lower_bound = max(0.0, distance(x, self.center) - self.radius)
        if len(knn) == k and lower_bound > knn[0][0]:
            return

        if self.left is None and self.right is None:

            ... # Pointsearch in leaf node

        if distance(x, self.left.center) < distance(x, self.right.cente
            self.left.query(x, k, knn)
```

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized).
   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized).
   Zum Beispiel, anstatt von list comprehension:

```
def vector_sum(vectors):
    return [sum(components) for components in zip(*vectors)]
```

2. **Avoid defining** often used functions.

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized).
   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

2. **Avoid defining** often used functions.
3. **Lokales Speichern** von oft verwendeten Variablen.

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized). ✓
   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

2. **Avoid defining** often used functions.
3. **Lokales Speichern** von oft verwendeten Variablen.

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized). ✓
   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

2. **Avoid defining** often used functions. X
3. **Lokales Speichern** von oft verwendeten Variablen.

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized). ✓
   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

2. **Avoid defining** often used functions. X
3. **Lokales Speichern** von oft verwendeten Variablen. X

Ball Tree: Rekursive Laufzeit

1. Verwenden von **built-in** Funktionen (C-optimized). ✓

   Zum Beispiel, anstatt von list comprehension:

   ```
   def vector_sum(vectors):
       return [sum(components) for components in zip(*vectors)]
   ```

2. **Avoid defining** often used functions. X
3. **Lokales Speichern** von oft verwendeten Variablen. X

Rekursive Methode nicht optimal !!!

References

References

author (XXXX). **“title”.** In.