The K-Nearest Neighbors Algorithm

5th February 2026

Team 2
Computerpraktikum Teil 2

Inhaltsverzeichnis

Problemstellung

Problemstellung

- Aufgabe: Methode zur **binären Klassifikation** in Python
- Gegeben: Datenpuntke mit Labels

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

- Lerne einen Klassifikator

$$f_D : [-1, +1]^d \to \{-1, +1\}$$

- Ziel: **Minimierung der Fehlklassifikationsrate** auf unbekannten Testdaten D'
- Ansatz: k-**nächste-Nachbarn** mit Kreuzvalidierung zur Auswahl von k^* mit Ball-Tree

Aufbau von `classify.py`

Laden von Datensätzen: Alte Version

```
try:
    import csv
    data = []
    with open(args.datasetname, 'r') as f:
        reader = csv.reader(f)
        for row in reader:
            if not row:
                continue
            try:
                label = int(row[0])
                features = [float(v) for v in
                    row[1:]]
                data.append((label, features))
            except ValueError:
                print("Warning: skipping
                    malformed row", row)
    print(f"Dataset loaded successfully. ...
```

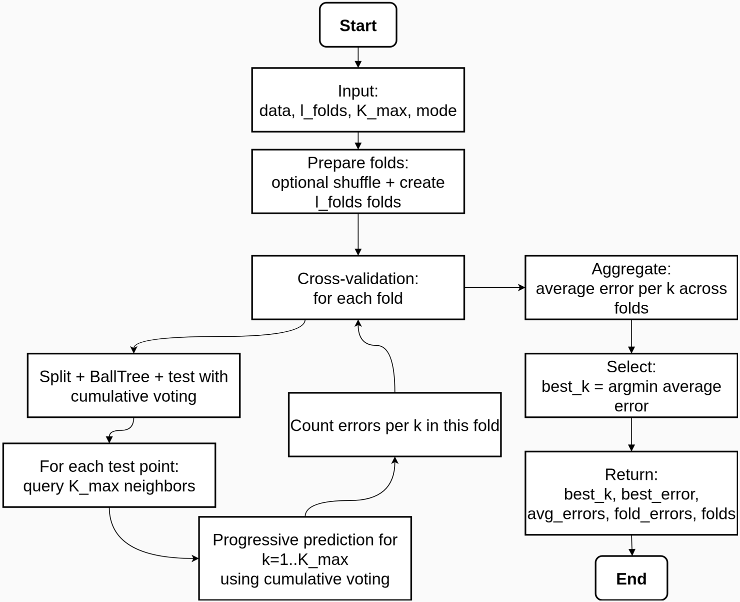
Laden von Datensätzen: Neue Version

```
def load_data(filename):
    data = []
    try:
        with open(filename, 'r') as f:
            for line in f:
                if not line.strip(): continue
                parts = line.split(',')
                data.append((float(parts[0]),
                   list(map(float, parts[1:]))))
    except Exception as e:
        print(f"Error while loading: {e}")
        sys.exit(1)
    return data
```

Erstellen vom Interface

```
usage: classify.py [-h] [-f l] [-k Kmax] [-d mode] [-n N] datasetname

KNN classification with l-fold cross validation.

The program determines the optimal k* ∈ {1, ..., kmax} using cross
 validation on the training data and then applies the resulting classifier
 f_D to the test data.

positional arguments:
  datasetname  Name of the dataset (without file extension).
               The following files are expected:
                 ../classification-data/<datasetname>.train.csv
                 ../classification-data/<datasetname>.test.csv

options:
  -h, --help   show this help message and exit
  -f l         Number of folds for cross validation (default: 5).
               The training data is split into l subsets D1, …, Dl.
  -k Kmax      Maximum value of k (default: 200).
               The set K = {1, 2, …, Kmax} is evaluated.
  -d mode      Mode for generating the folds (default: 0).
                 0: Random partitioning of the data
                 1: Deterministic partitioning as specified in the assignment:
                    D1 = (y1, x1), (yl+1, xl+1), (y2l+1, x2l+1), ...
                    D2 = (y2, x2), (yl+2, xl+2), (y2l+2, x2l+2), ...
                    ...
  -n N         Optional additional parameter.
               Uses only the first N training samples.
               This parameter does not affect the default behavior
               and is intended solely for testing or runtime experiments.
```

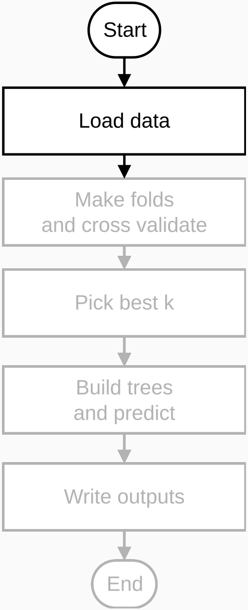
Die `run_cross_validation` Funktion



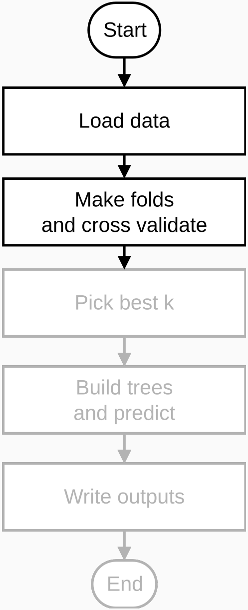
Die run_cross_validation Funktion

```python
def run_cross_validation(data, l_folds, K_max, mode):
    n = len(data)
    if mode != 1: random.shuffle(data)
    folds = [data[i::l_folds] for i in range(l_folds)]
    fold_errors = {k: [] for k in range(1, K_max + 1)}
    for i in range(l_folds):
        test_set = folds[i]
        train_set = []
        for j in range(l_folds):
            if i != j: train_set.extend(folds[j])

        tree = BallTree(train_set)
        current_fold_counts = {k: 0 for k in range(1, K_max + 1)}

        for y_true, x_test in test_set:
            neighbors = tree.query(x_test, K_max)
            current_sum = 0
            for idx, label in enumerate(neighbors):
                k = idx + 1
                current_sum += label
                y_pred = 1.0 if current_sum >= 0 else -1.0
                if y_pred != y_true:
                    current_fold_counts[k] += 1

        for k in range(1, K_max + 1):
            fold_errors[k].append(current_fold_counts[k] / len(test_set))
    avg_errors = {k: sum(fold_errors[k]) / l_folds for k in range(1, K_max + 1)}
    best_k = min(avg_errors.items(), key=lambda x: x[1])[0]
```

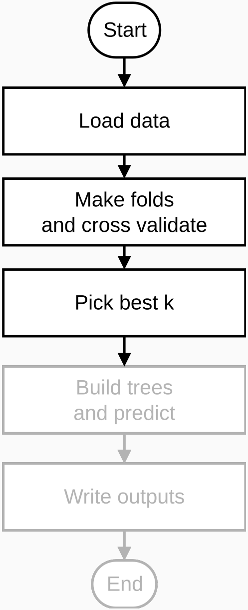
Die main Funktion



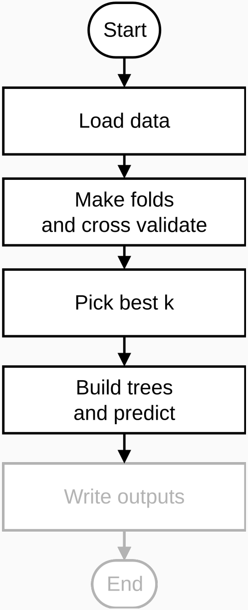
Die main Funktion



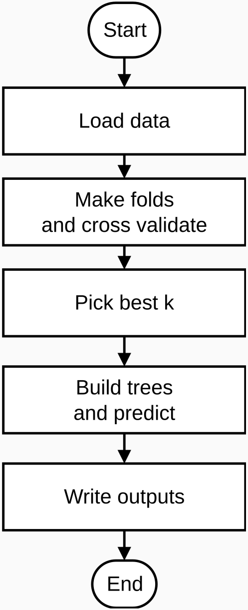
Die main Funktion



Die main Funktion

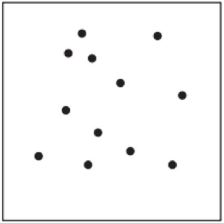


Die main Funktion



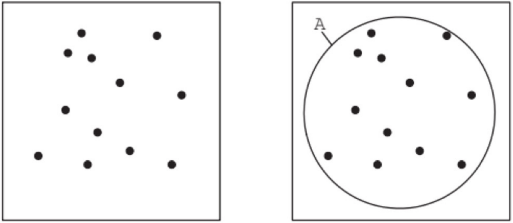
Aufbau von `ball_tree.py`

Ball Tree: Konzept



(A)

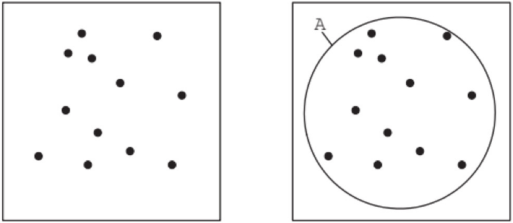
Ball Tree: Konzept



(A) (B)

An jedem Knoten:

Ball Tree: Konzept

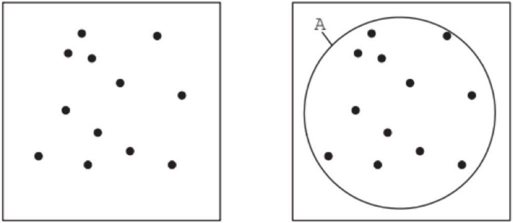


(A) (B)

An jedem Knoten:

1. Mittelpunkt, Radius berechnen

Ball Tree: Konzept

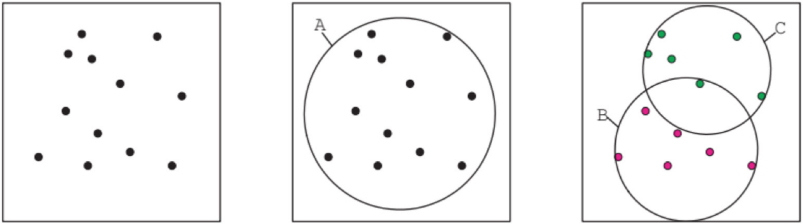


(A) (B)

An jedem Knoten:

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

Ball Tree: Konzept

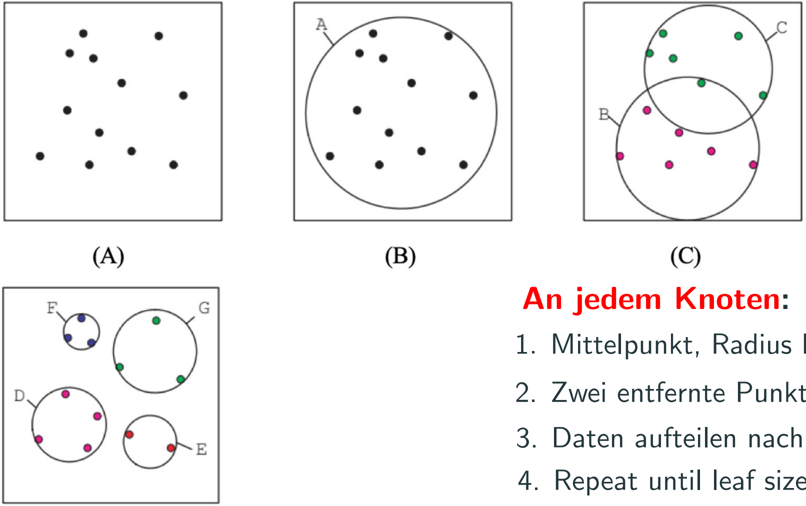


(A) (B) (C)

An jedem Knoten:

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

Ball Tree: Konzept

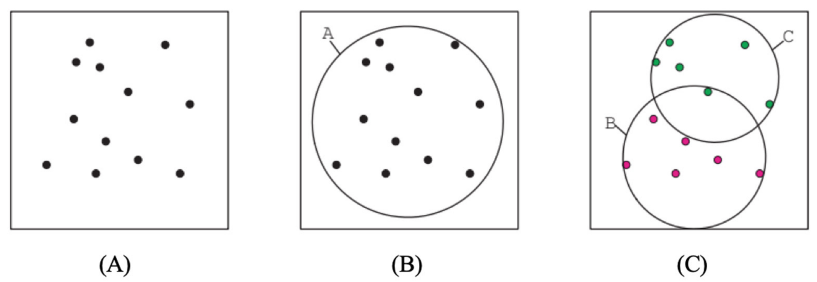


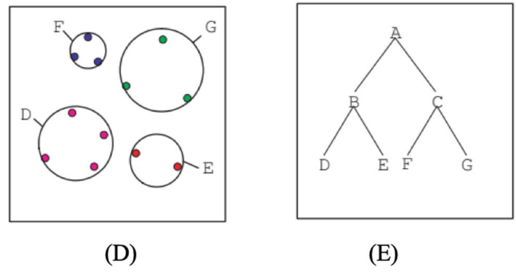
(A) (B) (C) (D)

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





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

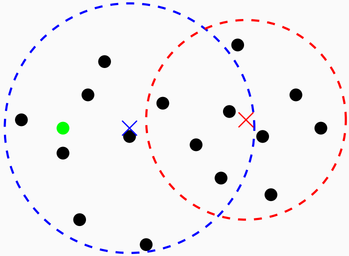
```python
def BallTree:

    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

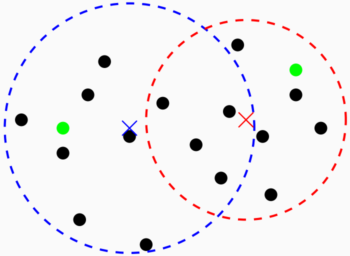
```
def BallTree:

    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)
        self.right = BallTree(right_points, self.leaf_size)
        self.points = None
```

Ball Tree: Rekursiv

```
def BallTree:
    ...
```

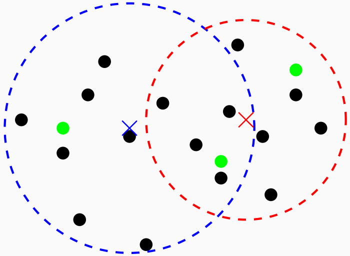


Ball Tree: Rekursiv



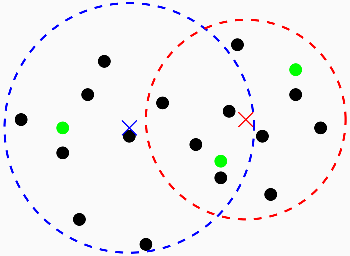
```
def BallTree:
    ...
```

Ball Tree: Rekursiv



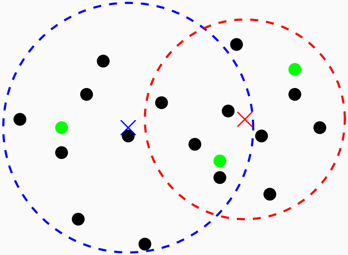
```python
def BallTree:
    ...
```

Ball Tree: Rekursiv



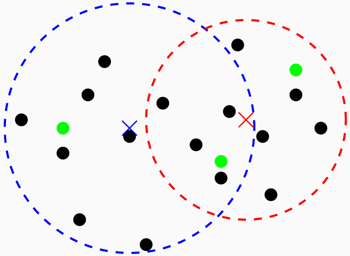
```
def BallTree:
    ...

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

Ball Tree: Rekursiv



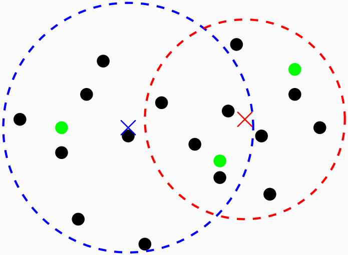
```
def BallTree:
    ...

    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.center):
            self.left.query(x, k, knn)
            self.right.query(x, k, knn)
        else:
            ...
```

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

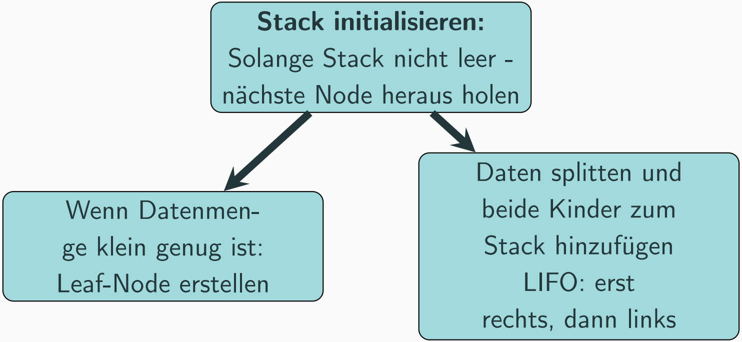
1. Verwenden von **built-in** Funktionen (C-optimized). ✓
   Zum Beispiel, anstatt von list comprehension:

```
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 !!!

Ball Tree: iterative Methode



Speed Ups

Speed Ups

1. built-in Funktionen $\Rightarrow$ C-optimiert
2. Best-First statt Depth-First $\Rightarrow$ intelligente Baum-Suche
3. Binomische Formel $\Rightarrow$ weniger Rechenoperationen
4. Norm-Vorberechnung $\Rightarrow$ Caching-Strategie
5. Global Pruning $\Rightarrow$ Frühe Terminierung

Speed Ups: Best-First statt Depth-First

- **Alt: Depth-First**
  - Stack-LIFO
  - Näheres Kind zuerst (lokal)
  - keine globale Priorisierung
- **Neu: Best-First**
  - Queue statt Stack
  - Sortieren die Queue bei jedem Loop nach der Distanz
  - Mit `reverse=True` und `.pop()` holen wir Node mit kleinster Distanz zum Query-Punkt ⇒ besuchen vielversprechendste Node als nächste
  - Beim Hinzufügen von Kindern speichern wir auch deren Distanz: Tupel `d-child`, `child-idx` erlaubt Sortierung

⇒ Wir besuchen deutliche weniger Nodes, weil weit entfernte Bereiche früher gepruned werden..

Speed Ups: Best-First statt Depth-First

```
def query(self, target, k):

    # Queue format: (distance_to_center, index)
    queue = [(d_root, 0)]

    while queue:
        # Sorting for best-first search (smallest distance first)
        queue.sort(key=lambda x: x[0], reverse=True)
        d_to_center, idx = queue.pop()
        node = self.nodes[idx]

        # Early termination (global stop)
        if len(neighbors) == k:
            # d_min_so_far is the square root of the worst distance
            if d_to_center > node['radius'] + math.sqrt(neighbors[-1][0]):
             break
        else:
            # Inspect child nodes and add them to the queue only if they can
            #potentially improve the result

            for child_idx in [node['left'], node['right']]:
                d_child = math.sqrt(max(0, target_norm_sq -
                2 * dot_child + child['center_norm_sq']))

                # Local pruning: add only if the ball is reachable
                if len(neighbors) < k or d_child <= child['radius']
                + math.sqrt(neighbors[-1][0]):
                    queue.append((d_child, child_idx))
```

Speed Ups: Binomische Formel

Alt: Naive Distanz

```
def query(target, k):
    # Fuer jeden Punkt:
    d_sq = sum(
        (t - c)**2
            for t, c in zip(target, coord)
    )
```

Kosten bei $d = 15$:

- 15 Subtraktionen
- 15 Quadrierungen
- 14 Additionen

44 Operationen

Neu: Binomische Formel

```
def __init__(self, data, leaf_size=40):
    # Norm-Vorberechnung
    self.point_norms[coords_tuple]
    = sum(c * c
    for c in coords)
def query(self, target, k):
    target_norm_sq = sum(t * t
    for t in target)
    # F r jeden Punkt:
    d_sq = target_norm_sq -
    2 * sum(t * c for t, c in
    zip(target, coord))
    + self.point_norms[c_tuple]
# 2d + 3 Operationen
```

Kosten bei $d = 15$:

33 Operationen

Fehlerreduktionsstrategien

Optimierung: Versuchte Methoden

- **stratified l-fold**
- additional distance metrics
- higher percision summation
- leaf size Variation

```
from collections import defaultdict
import random
def make_stratified_folds(data, l_folds, seed=42):
    rnd = random.Random(seed)
    buckets = defaultdict(list)
    for y, x in data:
        buckets[y].append((y, x))
    for y in buckets:
        rnd.shuffle(buckets[y])
    folds = [[] for _ in range(l_folds)]
    for y, items in buckets.items():
        for i, item in enumerate(items):
            folds[i % l_folds].append(item)
    return folds
```

Optimierung: Versuchte Methoden

- stratified l-fold
- **additional distance metrics**
- higher percision summation
- leaf size Variation

```
if self.metric == "l2":
  return sum((x - y) ** 2 for x, y in zip(a, b))
if self.metric == "l1":
  return sum(abs(x - y) for x, y in zip(a, b))
# linf
    return max(abs(x - y) for x, y in zip(a, b))
```

Optimierung: Versuchte Methoden

- stratified l-fold
- additional distance metrics
- **higher percision summation**
- **leaf size Variation**

```
math.fsum(...)
```

Fehlerreduktion: Plots

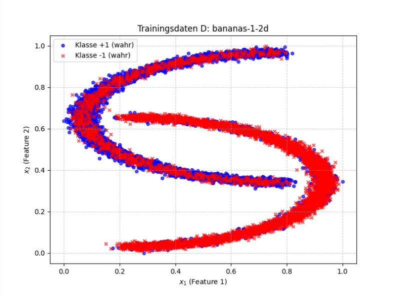


Abbildung 1: Alt: Sequentielles Plotten

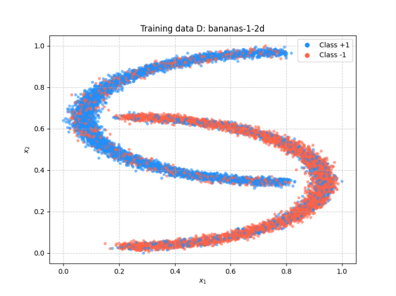


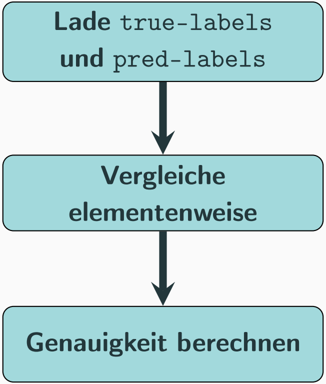
Abbildung 2: Neu: Zufälliges Plotten

- Klassen nacheinander geplottet
- Klasse -1 überdeckt +1

- `random.shuffle()` vor Plot
- Tatsächliche Überlappung der Klassen ⇒ „wahre“ Struktur

Fehlerreduktion: `check accuracy.py`

Zweck: Vergleiche `test.csv` mit `result.csv` und berechne Genauigkeit
Nutzen: Direkte Datenvalidierung



Lernerfolge

Lernerfolge: Technische Skills

1. **Kollaboration:**
   - GitHub
   - Code Reviews
2. **Software Engineering:**
   - Iterative Entwicklung: Rekursiv $\rightarrow$ Iterativ
   - Testing & Debugging
   - Dokumentation
3. **Datenstrukturen & Algorithmen**
   - BallTree
   - Komplexität
4. **Performance-Optimierung:**
   - Profiling: Bottleneck-Identifikation
   - Algorithmen:
   - Mathematik: Binomische Formel

Lernerfolge: Theoretisches Verständnis

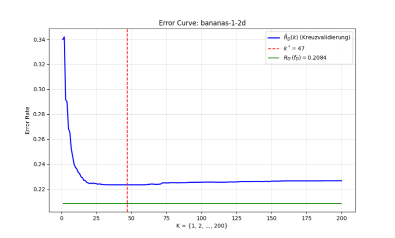


Abbildung 3: Fehlerkurve `bananas-1-2d`

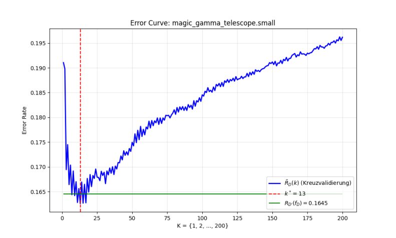


Abbildung 4: Fehlerkurve `magic gamma telescope.small`

Vielen Dank für Ihre Aufmerksamkeit!