K-Nearest Neighbours (KNN) for Simple Data

Zbigniew Dendzik

March 2026

Lecture Plan

- Introduction to supervised learning
- K-Nearest Neighbours algorithm
- Point2D class – 2D point representation
- Distance class – Euclidean distance
- PointSet class – training data storage
- KNN classifier implementation
- Testing and accuracy evaluation
- Extensions and comparison with advanced models

Introduction to Supervised Learning

Goal: Learn a mapping from input to output based on labeled examples.

Application examples:

- Image classification (cat / dog / bird)
- Handwritten digit recognition (0-9)
- Medical diagnosis (healthy / sick)
- Credit scoring (approve / reject)

Approaches:

- Simple: K-Nearest Neighbours
- Advanced: neural networks, deep learning, LLM

K-Nearest Neighbours Algorithm

KNN – one of the simplest supervised learning algorithms.

Main idea:

- Each example is a point in feature space
- Training data = collection of labeled points
- Classification: find K nearest neighbors
- Predict class by majority voting

Pros and cons:

+ Very simple implementation
+ No training phase
– Computation at classification time (lazy learning)
– Requires distance metric

Feature Space

Feature space – representation of data as points.

Example 1: 2D points

- Features: (x, y) coordinates
- Class labels: "A", "B"
- Distance: Euclidean

Example 2: Iris flowers

- Features: petal length, petal width, sepal length, sepal width
- Class labels: "setosa", "versicolor", "virginica"
- 4-dimensional feature space

KNN Algorithm – Step by Step

Training phase:

- Store all training examples
- No computation required

Classification phase:

1. Given a query point q (without label)
2. Compute distance from q to all training points
3. Sort points by distance
4. Select K nearest points
5. Count class labels among K neighbors
6. Return most frequent class (majority voting)

Distance Metric

Distance metric – measure of similarity between points.

Euclidean distance (2D):

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Euclidean distance (n-D):

$$d(p_1, p_2) = \sqrt{\sum_{i=1}^{n} (x_{1,i} - x_{2,i})^2}$$

Other metrics:

- Manhattan distance: $|x_1 - x_2| + |y_1 - y_2|$
- Minkowski distance: $(\sum |x_i - y_i|^p)^{1/p}$

System Architecture

System components:

1. **Data representation**
   - `Point2D` – labeled point in 2D space
   - `Distance` – distance computation

2. **Training set**
   - `PointSet` – collection of training points
   - Method: `nearestNeighbours(q)` – find K nearest

3. **Classifier**
   - `KNN` – K-Nearest Neighbours classifier
   - Training: store training set
   - Prediction: majority voting

Point2D Class

```java
class Point2D {
    private double x;
    private double y;
    private String label; // e.g. "A", "B"

    public Point2D(double x, double y, String label) {
        this.x = x;
        this.y = y;
        this.label = label;
    }

    public double getX() { return x; }
    public double getY() { return y; }
    public String getLabel() { return label; }

    public String toString() {
        return "Point2D(" + x + ", " + y + ", " +
    label + ")";
    }
}
```

Point2D Class – Explanation

Private fields:

- `x` – x-coordinate
- `y` – y-coordinate
- `label` – class label (category)

Public methods:

- `getX()` – returns x-coordinate
- `getY()` – returns y-coordinate
- `getLabel()` – returns class label
- `toString()` – string representation

Application:

- Storing training examples
- Representing query points
- Basic building block of training set

Distance Class

```
class Distance {
    public static double euclidean(Point2D a, Point2D
    b) {
        double dx = a.getX() - b.getX();
        double dy = a.getY() - b.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Explanation:

- Static method – can be called without creating object
- Computes Euclidean distance between two points
- Returns `double` value
- Application: distance computation in KNN

TestPoint2D – Example Program

```
1  public class TestPoint2D {
2      public static void main(String[] args) {
3          Point2D p1 = new Point2D(1.0, 2.0, "A");
4          Point2D p2 = new Point2D(2.0, 3.0, "A");
5          Point2D p3 = new Point2D(4.0, 5.0, "B");
6
7          System.out.println(p1);
8          System.out.println(p2);
9          System.out.println(p3);
10
11         double d12 = Distance.euclidean(p1, p2);
12         double d13 = Distance.euclidean(p1, p3);
13
14         System.out.println("d(p1, p2) = " + d12);
15         System.out.println("d(p1, p3) = " + d13);
16     }
17 }
```

PointSet Class – Structure

Purpose: Store training points and find nearest neighbors.

Fields:

- `List<Point2D> points` – list of training points

Methods:

- `add(Point2D p)` – add training point
- `getPoints()` – retrieve all points
- `nearestNeighbours(Point2D q)` – find nearest neighbors

Helper class:

- `PointDistance` – stores point with its distance to query
- Used for sorting by distance

PointSet Class – Fields and Constructor

```
1  import java.util.*;
2
3  class PointSet {
4      private List<Point2D> points;
5
6      public PointSet() {
7          points = new ArrayList<Point2D>();
8      }
9
10     public void add(Point2D p) {
11         points.add(p);
12     }
13
14     public List<Point2D> getPoints() {
15         return points;
16     }
17 }
```

PointSet – PointDistance Helper Class

```
1 static class PointDistance {
2     Point2D point;
3     double distance;
4
5     PointDistance(Point2D point, double distance) {
6         this.point = point;
7         this.distance = distance;
8     }
9 }
```

Explanation:

- Helper class for sorting
- Stores point with its distance to query
- Used in `nearestNeighbours` method

PointSet – nearestNeighbours Method (1)

```
1  public List<Point2D> nearestNeighbours(Point2D q) {
2      List<PointDistance> list =
3          new ArrayList<PointDistance>();
4
5      for (Point2D p : points) {
6          double d = Distance.euclidean(p, q);
7          list.add(new PointDistance(p, d));
8      }
9
10     // sorting and returning...
11 }
```

Step 1:

- Create list of `PointDistance` objects
- For each training point compute distance to query
- Store point with distance

PointSet – nearestNeighbours Method (2)

```
1  Collections.sort(list,
2      new Comparator<PointDistance>() {
3          public int compare(PointDistance a,
4                             PointDistance b) {
5              return Double.compare(a.distance,
6                                    b.distance);
7          }
8      });
9
10 List<Point2D> result = new ArrayList<Point2D>();
11 for (PointDistance pd : list) {
12     result.add(pd.point);
13 }
14 return result;
```

Step 2:

- Sort list by distance (ascending order)
- Extract points to result list
- Return sorted list

TestPointSet – Example Program (1)

```
1  import java.util.*;
2
3  public class TestPointSet {
4      public static void main(String[] args) {
5          PointSet set = new PointSet();
6          set.add(new Point2D(1, 2, "A"));
7          set.add(new Point2D(2, 1, "A"));
8          set.add(new Point2D(4, 5, "B"));
9          set.add(new Point2D(5, 4, "B"));
10         set.add(new Point2D(0, 0, "A"));
11
12         Point2D q = new Point2D(3, 3, "?");
13
14         // find neighbors...
15     }
16 }
```

TestPointSet – Example Program (2)

```
1  List<Point2D> neigh = set.nearestNeighbours(q);
2
3  System.out.println("Nearest neighbours of ("
4      + q.getX() + ", " + q.getY() + "):");
5
6  for (Point2D p : neigh) {
7      double d = Distance.euclidean(p, q);
8      System.out.println(p + " d=" + d);
9  }
10
11 int K = 3;
12 System.out.println("First " + K + " neighbours:");
13 for (int i = 0; i < K && i < neigh.size(); i++) {
14     System.out.println((i+1) + ": " + neigh.get(i));
15 }
```

KNN Class – Structure

Purpose: Classify query points using K-Nearest Neighbours.

Fields:

- `PointSet set` – training data
- `int K` – number of neighbors to consider

Methods:

- `KNN(PointSet set, int K)` – constructor
- `getK()` – returns K value
- `setK(int K)` – updates K value
- `classify(Point2D q)` – classifies query point

KNN Class – Fields and Constructor

```
1  import java.util.*;
2
3  class KNN {
4      private PointSet set;
5      private int K;
6
7      public KNN(PointSet set, int K) {
8          this.set = set;
9          this.K = K;
10     }
11
12     public int getK() {
13         return K;
14     }
15
16     public void setK(int K) {
17         this.K = K;
18     }
19 }
```

KNN – classify Method (1)

```
1  public String classify(Point2D q) {
2      List<Point2D> neigh =
3          set.nearestNeighbours(q);
4
5      Map<String, Integer> counts =
6          new HashMap<String, Integer>();
7
8      for (int i = 0; i < K && i < neigh.size(); i++) {
9          Point2D p = neigh.get(i);
10         String label = p.getLabel();
11         Integer c = counts.get(label);
12         if (c == null) c = 0;
13         counts.put(label, c + 1);
14     }
15
16     // find most frequent label...
17 }
```

KNN – classify Method (2)

```
String bestLabel = null;
int bestVotes = -1;

for (String label : counts.keySet()) {
    int votes = counts.get(label);
    if (votes > bestVotes) {
        bestVotes = votes;
        bestLabel = label;
    }
}

return bestLabel;
```

Classification process:

- Count votes for each class among K neighbors
- Return class with maximum votes (majority voting)
- In case of tie: first class encountered wins

Majority Voting

Example: Query point $q = (3, 3)$, $K = 5$

5 nearest neighbors:

- $(2.8, 3.1)$ – class A – distance 0.28
- $(3.2, 2.9)$ – class A – distance 0.28
- $(4.0, 4.0)$ – class B – distance 1.41
- $(2.0, 2.0)$ – class A – distance 1.41
- $(5.0, 5.0)$ – class B – distance 2.83

Vote counting:

- Class A: 3 votes
- Class B: 2 votes

Prediction: Class A (majority)

TestKNN – Example Program (1)

```
1  import java.util.*;
2
3  public class TestKNN {
4      public static void main(String[] args) {
5          PointSet set = new PointSet();
6          set.add(new Point2D(1, 2, "A"));
7          set.add(new Point2D(1, 1, "A"));
8          set.add(new Point2D(2, 1, "A"));
9          set.add(new Point2D(4, 5, "B"));
10         set.add(new Point2D(5, 4, "B"));
11         set.add(new Point2D(6, 5, "B"));
12
13         Point2D[] test = {
14             new Point2D(2, 2, "?"),
15             new Point2D(4, 4, "?"),
16             new Point2D(0, 0, "?")
17         };
18
19         // test for different K...
20     }
21 }
```

TestKNN – Example Program (2)

```
int[] Ks = {1, 3, 5};

for (int K : Ks) {
    KNN knn = new KNN(set, K);
    System.out.println("K=" + K);
    for (Point2D q : test) {
        String label = knn.classify(q);
        System.out.println("Point (" + q.getX()
            + ", " + q.getY()
            + ") -> " + label);
    }
}
```

Comparing K values:

- Different K may give different predictions
- Small K: sensitive to noise
- Large K: over-smoothing

Random Data Generation

DataGenerator class: Generate random clusters of points.

Methods:

- `randomPoint(cx, cy, spread, label)` – single random point
- `generateCluster(n, cx, cy, spread, label)` – cluster of n points

Parameters:

- `cx, cy` – center coordinates
- `spread` – standard deviation (Gaussian distribution)
- `label` – class label
- `n` – number of points

Application:

- Testing KNN on synthetic data
- Evaluating accuracy

DataGenerator Class

```
1  import java.util.*;
2
3  class DataGenerator {
4      private Random rand = new Random();
5
6      public Point2D randomPoint(double cx, double cy,
7                                 double spread,
8                                 String label) {
9          double x = cx + rand.nextGaussian() * spread;
10         double y = cy + rand.nextGaussian() * spread;
11         return new Point2D(x, y, label);
12     }
13
14     // generateCluster method...
15 }
```

DataGenerator – generateCluster Method

```
public List<Point2D> generateCluster(int n,
                                     double cx,
                                     double cy,
                                     double spread,
                                     String label) {
    List<Point2D> list = new ArrayList<Point2D>();
    for (int i = 0; i < n; i++) {
        list.add(randomPoint(cx, cy, spread, label));
    }
    return list;
}
```

Explanation:

- Generate `n` random points
- Points distributed around center `(cx, cy)`
- Gaussian distribution with standard deviation `spread`

Accuracy Evaluation

Accuracy metric:

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{total number of tests}}$$

Example:

- 100 test points
- 85 correct predictions
- Accuracy $= 85/100 = 0.85 = 85\%$

Procedure:

1. Split data into training and test sets (e.g. 70% / 30%)
2. Train classifier on training set
3. Test on test set
4. Count correct predictions

EvaluateKNN – Program (1)

```
1  import java.util.*;
2
3  public class EvaluateKNN {
4      public static void main(String[] args) {
5          DataGenerator gen = new DataGenerator();
6
7          List<Point2D> classA =
8              gen.generateCluster(50, 0.0, 0.0, 1.0, "A"
       );
9          List<Point2D> classB =
10             gen.generateCluster(50, 5.0, 5.0, 1.0, "B"
       );
11
12         List<Point2D> all = new ArrayList<Point2D>();
13         all.addAll(classA);
14         all.addAll(classB);
15
16         Collections.shuffle(all);
17
18         // split into train and test...
19     }
20 }
```

EvaluateKNN – Program (2)

```
1  int n = all.size();
2  int nTrain = (int)(0.7 * n);
3
4  PointSet train = new PointSet();
5  List<Point2D> test = new ArrayList<Point2D>();
6
7  for (int i = 0; i < n; i++) {
8      Point2D p = all.get(i);
9      if (i < nTrain)
10         train.add(p);
11     else
12         test.add(p);
13 }
14
15 // evaluate for different K...
```

Data split:

- 70% training
- 30% testing

EvaluateKNN – Program (3)

```
int[] Ks = {1, 3, 5, 7};

for (int K : Ks) {
    KNN knn = new KNN(train, K);
    int correct = 0;

    for (Point2D q : test) {
        String pred = knn.classify(q);
        if (pred.equals(q.getLabel()))
            correct++;
    }

    double accuracy =
        (double)correct / (double)test.size();
    System.out.println("K=" + K
        + " accuracy=" + accuracy);
}
```

Choosing K Value

Small K (e.g. K=1):

+ Sensitive to local patterns
– Sensitive to noise and outliers
– Overfitting risk

Large K (e.g. K=dataset size):

+ Smooth decision boundaries
– Over-smoothing
– May ignore local patterns
– Underfitting risk

Optimal K:

- Typically: $K = \sqrt{n}$ where n is training set size
- Use cross-validation to find best K
- Odd K preferred (avoid ties)

KNN Behavior – Examples

Well-separated clusters:

- High accuracy for any reasonable K
- Even K=1 works well

Overlapping clusters:

- Lower accuracy
- Optimal K more critical
- Larger K helps reduce noise impact

Imbalanced classes:

- Class A: 90 points, Class B: 10 points
- Large K biased toward majority class
- Small K better for minority class detection

Extensions – Weighted KNN

Problem: All K neighbors have equal vote.

Solution: Weight votes by inverse distance.

$$w_i = \frac{1}{d_i}$$

where d_i is distance to i-th neighbor.

Weighted voting:

- Closer neighbors have more influence
- Farther neighbors have less influence
- Better accuracy in many cases

Implementation:

- Modify `classify` method
- Sum weights instead of counting votes

Extensions – Normalization

Problem: Features on different scales.

Example:

- Feature 1: height (150-200 cm)
- Feature 2: weight (50-100 kg)
- Distance dominated by larger values

Solution: Normalize features to same scale.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

Result:

- All features in range [0, 1]
- Equal contribution to distance
- Better classification accuracy

Extensions – K-D Tree

Problem: Naive KNN requires computing distance to all points.

Complexity:

- $O(n)$ distance computations per query
- $O(n \log n)$ for sorting
- Slow for large datasets

Solution: K-D tree data structure.

K-D tree:

- Binary tree partitioning space
- Faster neighbor search: $O(\log n)$ average
- Preprocessing: $O(n \log n)$
- Worth it for multiple queries

KNN vs Advanced Models

Aspect	**KNN**	**Neural Networks**
Training	None	Extensive
Memory	Full dataset	Model parameters
Speed	Slow at test	Fast at test
Interpretability	High	Low
Complexity	Very low	Very high
Data requirement	Dozens	Thousands
Feature learning	No	Yes

Example:

- KNN: stores all points, distance-based decision
- Neural network: learns complex decision boundaries

When is KNN Useful?

Good applications:

1. Small to medium datasets
   - Dozens to thousands of examples
   - Low-dimensional feature space
2. Rapid prototyping
   - Quick baseline implementation
   - Testing idea before complex model
3. Interpretable results
   - Need to explain predictions
   - Show similar examples
4. Non-linear decision boundaries
   - Complex class shapes
   - No parametric assumptions

Limitations of KNN

Challenges:

1. Curse of dimensionality
   - High-dimensional spaces: all points equally distant
   - Distance metric becomes meaningless
2. Computational cost
   - $O(n)$ per query for naive implementation
   - Large memory requirement
3. Sensitive to irrelevant features
   - Noise features degrade performance
   - Feature selection crucial
4. Imbalanced classes
   - Majority class dominates
   - Need weighted voting or resampling

Modern Approaches

Evolution of classification methods:

1. **Classical ML (1950s-2010):**
   - KNN, decision trees, SVM, random forests
   - Hand-crafted features
   - Small to medium datasets

2. **Deep learning (2012-2020):**
   - Convolutional neural networks (CNN)
   - Automatic feature learning
   - Large labeled datasets

3. **Large language models (2020+):**
   - Transformers, GPT, BERT
   - Transfer learning
   - Few-shot and zero-shot classification

KNN vs Embeddings

Traditional KNN:

- Simple numeric features (coordinates, measurements)
- Euclidean distance
- No feature learning

Modern approach – Embeddings:

- Neural network maps data to embedding space
- Learned representation (e.g. 128-dimensional vector)
- Semantic distance (similar meaning = small distance)
- KNN in embedding space = powerful classifier

Example:

- Text: BERT embeddings + KNN
- Images: ResNet embeddings + KNN
- Better than raw features

Summary

- **KNN** – simple supervised learning algorithm
- **Lazy learning** – no training, computation at test time
- **Distance metric** – Euclidean distance for numeric features
- **Majority voting** – predict most frequent class among K neighbors
- **Parameter K** – critical choice, use cross-validation
- **Extensions** – weighted voting, normalization, K-D tree
- **Limitations** – curse of dimensionality, computational cost
- **Applications** – small datasets, rapid prototyping, interpretability
- **Modern approach** – embeddings + KNN