Unit 2: Supervised Learning
Part 1

Dr. Goonjan Jain
Mathematics and Computing
Department of Applied Mathematics
DTU

Contents

- K-nearest neighbours
- Bayesian Classification
- Naïve Bayes
- Linear Discriminant Analysis
- Linear Regression
- Logistic Regression
- Multilayer Perceptron
- Support Vector Machine
- Decision Trees

K-nearest neighbours

- K-Nearest Neighbours (KNN) is a supervised learning algorithm.
- It is used for both classification and regression tasks.
- The algorithm works based on similarity between data points.
- also called **lazy learner algorithm**
- does not learn from the training set immediately instead it stores the entire dataset and performs computations only at the time of classification.

Why KNN?

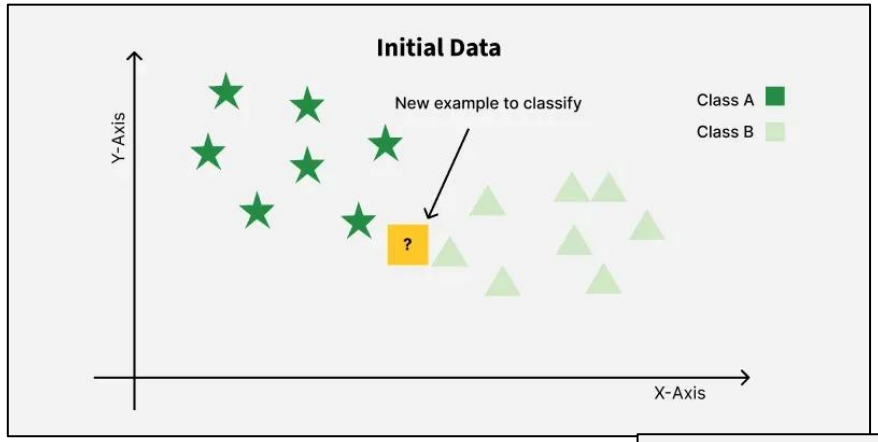
- Simple and intuitive algorithm.
- No explicit training phase.
- Works well with small datasets.
- Easy to understand and implement.

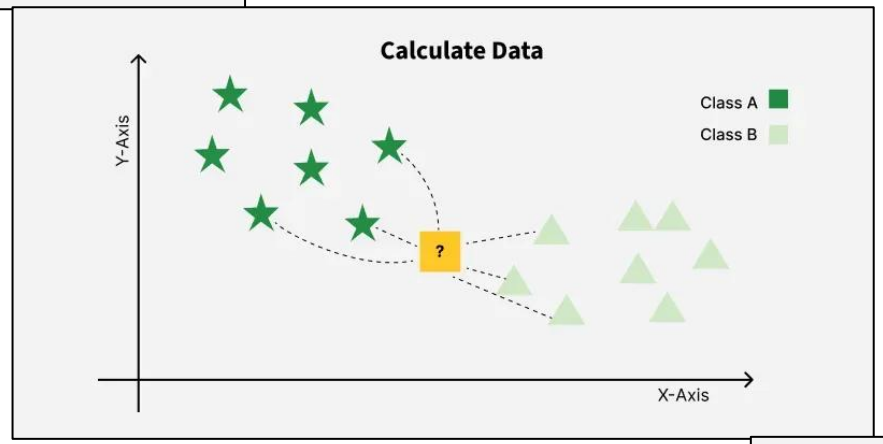
How KNN Works

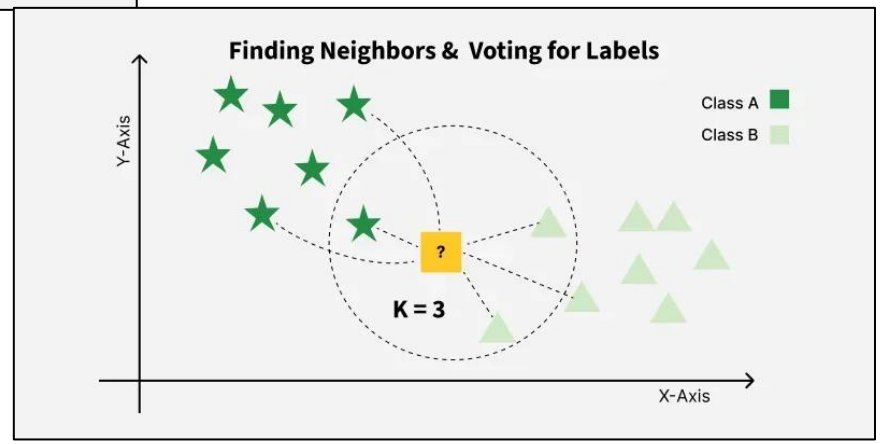
- Choose the value of K.
- Calculate distance between test point and training points.
- Select K nearest neighbors.
- Assign class by majority voting.

What is ‘K’?

- a number that tells the algorithm how many nearby points or neighbors to look at when it makes a decision.
- deciding which fruit it is based on its shape and size.
- compare it to fruits you already know.
- If k = 3, the algorithm looks at the 3 closest fruits to the new one.
- If 2 of those 3 fruits are apples and 1 is a banana, the algorithm says the new fruit is an apple because most of its neighbors are apples.


Initial Data
New example to classify
Class A
Class B
Y-Axis
X-Axis
?


Calculate Data
Class A
Class B
Y-Axis
X-Axis
?


Finding Neighbors & Voting for Labels
Class A
Class B
Y-Axis
X-Axis
?
K = 3

Choosing the Value of K

- Small K: Sensitive to noise.
- Large K: Smoother decision boundary.
- Optimal K is chosen using cross-validation.

Statistical Methods for Selecting k

- **Cross-Validation**:
  - use k-fold cross-validation.
  - divide the dataset into k parts
  - model is trained on some of these parts and tested on the remaining ones
  - process is repeated for each part
  - The k value that gives the highest average accuracy during these tests is usually the best one to use.
- **Elbow Method**:
  - draw a graph showing the error rate or accuracy for different k values.
  - As k increases the error usually drops at first. But after a certain point error stops decreasing quickly.
  - The point where the curve changes direction and looks like an "elbow" is usually the best choice for k.
- **Odd Values for k**:
  - use an odd number for k especially in classification problems.
  - helps avoid ties when deciding which class is the most common among the neighbors.

Distance Metrics

- Euclidean Distance
  - straight-line distance between two points in a plane or space

$$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^{d} (x_j - X_{i_j})^2]}$$

- Manhattan Distance
  - total distance you would travel if you could only move along horizontal and vertical lines like a grid or city streets

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

- Minkowski Distance
  - a family of distances, which includes both Euclidean and Manhattan distances as special cases.

$$d(x, y) = \left(\sum_{i=1}^{n} (x_i - y_i)^p\right)^{\frac{1}{p}}$$

- Euclidean distance is most commonly used.

Python Code using library

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

# Training data: [hours studied, attendance %]
X = np.array([[2, 60],  [4, 70],  [6, 80],  [8, 90],  [1, 50],  [9, 95]])

# Labels: 0 = Fail, 1 = Pass
y = np.array([0, 0, 1, 1, 0, 1])

# Create KNN model with k = 3
knn = KNeighborsClassifier(n_neighbors=3)

# Train the model
knn.fit(X, y)

# New student data
new_student = np.array([[5, 75]])

# Prediction
prediction = knn.predict(new_student)
print("Prediction:", "Pass" if prediction[0] == 1 else "Fail")
```

Python Code

```
import math

# Training data: [hours studied, attendance %]
X_train = [[2, 60],   [4, 70],   [6, 80],   [8, 90],   [1, 50],   [9, 95]]

# Labels: 0 = Fail, 1 = Pass
y_train = [0, 0, 1, 1, 0, 1]

# New data point
test_point = [5, 75]
k = 3

# Step 1: Euclidean distance function
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0])**2 + (p1[1] - p2[1])**2)
```

```
# Step 2: KNN function
def knn_predict(X, y, test, k):
    distances = []

    # Calculate distance from test point to all training points
    for i in range(len(X)):
        dist = euclidean_distance(X[i], test)
        distances.append((dist, y[i]))

    # Sort distances
    distances.sort(key=lambda x: x[0])

    # Take k nearest neighbors
    nearest_neighbors = distances[:k]

    # Majority voting
    votes = [label for (_, label) in nearest_neighbors]
    prediction = max(set(votes), key=votes.count)

    return prediction

# Make prediction
result = knn_predict(X_train, y_train, test_point, k)

print("Prediction:", "Pass" if result == 1 else "Fail")
```

Advantages and Disadvantages

- **Advantages:**
- Simple and effective
- No training phase

- **Disadvantages:**
- Slow for large datasets
- Sensitive to irrelevant features

Applications of KNN

- Pattern recognition
- Recommendation systems
- Medical diagnosis
- Image and text classification

Bayesian Classification

- Real-world data is uncertain
- Decisions should consider probability, not just rules
- Bayesian methods provide:
  - Mathematical foundation
  - Interpretability
- A **probabilistic classifier**
- Assigns a class label based on:
  - Prior probability of a class
  - Probability of observing data given the class
- Uses **Bayes' Theorem**
- Works well when:
  - Uncertainty is present
  - Prior knowledge is useful

Bayes' Theorem – Core Formula

$$P(C \mid X) = \frac{P(X \mid C) \cdot P(C)}{P(X)}$$

Where:

- X – input data
- C - class
- $P(C \mid X)$:Posterior probability - Updated belief after seeing data
- $P(X \mid C)$:Likelihood - How likely data is for a class
- $P(C)$:Prior probability - What we believe before seeing data
- $P(X)$:Evidence

- Choose class **C** with maximum $P(C \mid X)$

Bayesian Classification Idea

- Compute probability for each class
- Assign the class with **highest posterior probability**
- Decision Rule:

$$Class = argmax P(C \mid X)$$

Simple Example – Weather & Play Tennis

- Given:
  - Classes: Play / Not Play
  - Input: Weather = Sunny
- We compute:
  - P(Play | Sunny)
  - P(Not Play | Sunny)
- Choose the higher value
- Assume:
  - P(Play) = 0.6
  - P(Not Play) = 0.4
  - P(Sunny | Play) = 0.3
  - P(Sunny | Not Play) = 0.5
- Compute posterior probabilities

$$P(Play \mid Sunny) = 0.3 \times 0.6 = 0.18$$
$$P(NotPlay \mid Sunny) = 0.5 \times 0.4 = 0.20$$

- Prediction: **Not Play**

Example: Disease Diagnosis using Bayesian Classification

- **Problem Statement**
- A doctor wants to diagnose whether a patient has **Flu (F)** or **Allergy (A)** based on **two features**:
- **Fever (Fv)** → {High, Normal}
- **Body Ache (Ba)** → {Yes, No}
- ⚠️ **Important:**
  Fever and Body Ache are **not independent**.
  If fever is high, body ache is **more likely**.

Step 1: Prior Probabilities

- From hospital data:

Disease	**Probability**
Flu (F)	P(F) = 0.6
Allergy (A)	P(A) = 0.4

Step 2: Joint Conditional Probabilities

- Given Flu

Fever	Body Ache	P(Fv,Ba| F)
High	Yes	0.50
High	No	0.10
Normal	Yes	0.25
Normal	No	0.15

Given Allergy

Fever	Body Ache	P(Fv,Ba| A)
High	Yes	0.05
High	No	0.15
Normal	Yes	0.10
Normal	No	0.70

Step 3: New Patient (Test Case)

- A patient has:
  - **Fever = High**
  - **Body Ache = Yes**
- We want to find:

P(Flu | High fever, Body Ache)

P(Allergy | High fever, Body Ache)

Step 4: Bayesian Classification

- **For Flu:**

$$P(F \mid Fv = High, Ba = Yes)$$
$$= P(F).\ P(Fv = High, Ba = Yes \mid F)$$
$$= 0.50 * 0.6 = 0.30$$

- **For Allergy:**

$$P(A \mid Fv = High, Ba = Yes)$$
$$= P(A).\ P(Fv = High, Ba = Yes \mid A)$$
$$= 0.05 * 0.4 = 0.02$$

Class	Posterior Score
Flu	0.30
Allergy	0.02

✅ **Predicted Disease: Flu**

Advantages of Bayesian Classification

- Strong theoretical foundation
- Works well with small datasets
- Handles uncertainty

Limitations

- Requires probability estimation
- Can be computationally expensive
- Assumptions may not always hold

Summary – Bayesian Classification

- Bayesian classification uses probability
- Based on Bayes' theorem
- Foundation for Naive Bayes

Naïve Bayes

- A special case of Bayesian classification
- Assumes **conditional independence** among features
  - Words in an email are independent
  - This assumption simplifies computation
- Surprisingly effective in practice

- For a data point

$$X = (\mathrm{x}_1, \mathrm{x}_2, \mathrm{x}_3 \ldots, \mathrm{x}_\mathrm{n})$$

The predicted class is:

$$\hat{C} = \underbrace{argmax}_{C} \; P(C) \prod_{i=1}^{n} P(x_i|C)$$

Where:

- $P(C)$ = Prior probability
- $P(x_i \mid C)$ = Likelihood
- Independence assumption applies

Example - Email Spam Detection using Naive Bayes

- **Problem Statement**
  - We want to classify an email as **Spam (S)** or **Not Spam (N)** using **two features**:
- **Contains word "Offer" (O)** → {Yes, No}
- **Contains word "Free" (F)** → {Yes, No}
- **Naive Assumption**
  - The presence of the words **"Offer"** and **"Free"** are **conditionally independent** given the class.
- This assumption is **what differentiates Naive Bayes** from full Bayesian classification.

Step 1: Prior Probabilities

- From historical email data

Class	**Probability**
Spam (S)	P(S) = 0.5
Not Spam (N)	P(N) = 0.5

Step 2: Feature Probabilities (Independent)

- Given spam

Feature	Probability
P(Offer=Yes | S)	0.7
P(Free=Yes | S)	0.8

Given Not Spam

Feature	Probability
P(Offer=Yes | N)	0.2
P(Free=Yes | N)	0.1

Step 3: New Email (Test Case)

- Email contains:
- **Offer = Yes**
- **Free = Yes**

Step 4: Naive Bayes Calculation

- For Spam:

$$P(S| O = Yes, F = Yes)$$
$$= P(S) * P(O = Yes \mid S) * P(F = Yes \mid S)$$
$$= 0.5 * 0.7 * 0.8$$
$$= 0.28$$

- For Not Spam:

$$P(N| O = Yes, F = Yes)$$
$$= P(N) * P(O = Yes \mid N) * P(F = Yes \mid N)$$
$$= 0.5 * 0.2 * 0.1$$
$$= 0.01$$

Final Decision

- Final Decision

Class	Score
Spam	**0.28**
Not Spam	0.01

- ✅ **Predicted Class: Spam**

Bayesian vs Naïve Bayes

Aspect	Bayesian Classification	Naive Bayes
Feature dependence	Allowed	Not allowed
Probability used	Joint	Product of marginals
Accuracy	Higher (if dependencies exist)	Lower
Complexity	High	Very low
Data needed	Large	Small

Bayesian Classification models reality more accurately but is computationally expensive.
Naive Bayes simplifies reality for speed and scalability.

Types of Naïve Bayes

- Gaussian Naive Bayes – continuous data
- Multinomial Naive Bayes – text data
- Bernoulli Naive Bayes – binary features

- Naive Bayes classifiers differ based on **how they model the feature probabilities** $P(x_i \mid C)$.

Gaussian Naïve Bayes

- When features are **continuous (real-valued)**
- Data roughly follows a **normal (Gaussian) distribution**
- Examples:
  - Height, weight, temperature
  - Marks, sensor readings
  - Medical measurements

Assumption

- Each feature follows a **Gaussian (normal) distribution** for each class.

Probability Model

$$P(x_i \mid C) = \frac{1}{\sqrt{2\pi\sigma_{C,i}^2}} \exp\left(-\frac{(x_i - \mu_{C,i})^2}{2\sigma_{C,i}^2}\right)$$

Where:

μ= mean of the feature in a class

σ^2 = variance of the feature in a class

Multinomial Naïve Bayes

- When features represent **counts or frequencies**
- Especially good for **text classification**
- Examples:
  - Word counts in documents
  - Email spam detection
  - Sentiment analysis

- **Assumption**
  - Features follow a **multinomial distribution**
  - Counts matter (how many times a word appears)

- **Intuition**
  - If a word appears **frequently in spam emails**, it increases spam probability
  - Repeated words strengthen the decision

$$P(x_i \mid C) = \frac{N_{C,i} + \alpha}{N_C + \alpha d}$$

Where:

$N_{C,i}$ =count of feature $i$in class C

N_C =total feature count in class C

d= number of features (vocabulary size)

α= smoothing parameter (Laplace smoothing)

Bernoulli Naïve Bayes

- When features are **binary (0 or 1)**
- Presence or absence matters
- Examples:
  - Word appears or not
  - Yes/No features
  - Clicked or not clicked

- **Assumption**
  - Features follow a **Bernoulli distribution**
  - Only presence or absence is important

Python Code – Gaussian Naïve Bayes

```
from sklearn.naive_bayes import GaussianNB
import numpy as np

# Training data
X = np.array([[1], [2], [3], [4], [5]])
y = np.array([0, 0, 1, 1, 1])

model = GaussianNB()
model.fit(X, y)

# Prediction
print(model.predict([[2]]))
```

Python Code: Multinomial Naive Bayes

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB

texts = ["free money", "win prize", "hello friend", "how are you"]
labels = [1, 1, 0, 0]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

model = MultinomialNB()
model.fit(X, labels)

print(model.predict(vectorizer.transform(["free prize"])) )
```

Advantages

- Very fast
- Works well for text data
- Needs less training data

Limitations

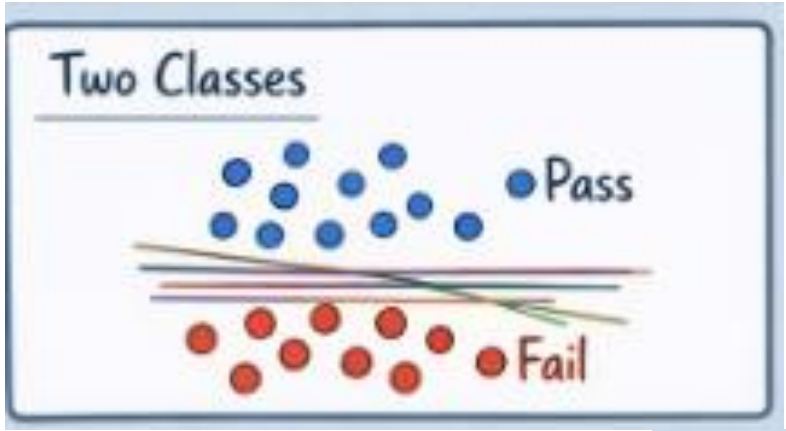
- Independence assumption is unrealistic
- Poor performance with correlated features

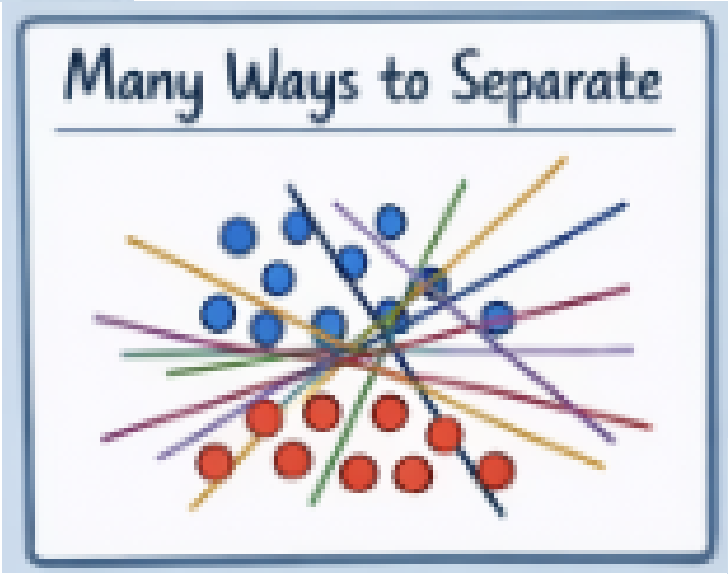
Key Takeaways

- Naive Bayes is simple yet powerful
- Based on Bayes’ theorem
- Widely used in NLP and spam detection

Linear Discriminant Analysis

- In many ML problems:
  - Data has **multiple features**
  - Visualizing and classifying becomes difficult
  - Some features may be **irrelevant or redundant**
- **Question:** Can we project data into a lower dimension **while preserving class separability**?
- ➡️ **Answer:** Linear Discriminant Analysis (LDA)


Two Classes
Pass
Fail


Many Ways to Separate


LDA finds the Best Line

LDA - Analogy

- 🎯 **Imagine shooting arrows at two targets**
  - Target A = Class 1
  - Target B = Class 2
- Bad scenario:
  - Arrows are spread everywhere
  - Targets are close
- Good scenario:
  - Arrows tightly grouped
  - Targets far apart
- 👉 LDA **moves and rotates the camera** to get the **clearest view** where targets look far apart and compact.

LDA as a smart projection

- Instead of working in 2D or 3D…
- LDA asks:
  - “If I project all points onto **one-line**, which line will make the classes easiest to separate?”
- **Bad projection** ❌
  - Classes overlap heavily
- **Good projection** ✅
  - Classes form **separate clusters**
- 👉 LDA finds the **best possible line**

What does LDA actually do?

- LDA does two things at the same time:
  - Push different classes far apart
  - Pull data points of the same class close together
- 📌 **Best separation =**
- Maximum distance between class means
- Minimum spread within each class

Example

- Each point is a data sample.
  - X-axis → **Feature 1**
  - Y-axis → **Feature 2**
- Two classes:
  - **Class 0** (left cluster)
  - **Class 1** (right cluster)
- **Key observation to tell students**
  - The data is **2-dimensional**
  - The classes are somewhat separable, but **not by a single obvious axis**
  - We need a **good direction** to project data so that the classes are best separated
- "At this stage, we only have data. No model, no line, no intelligence yet."

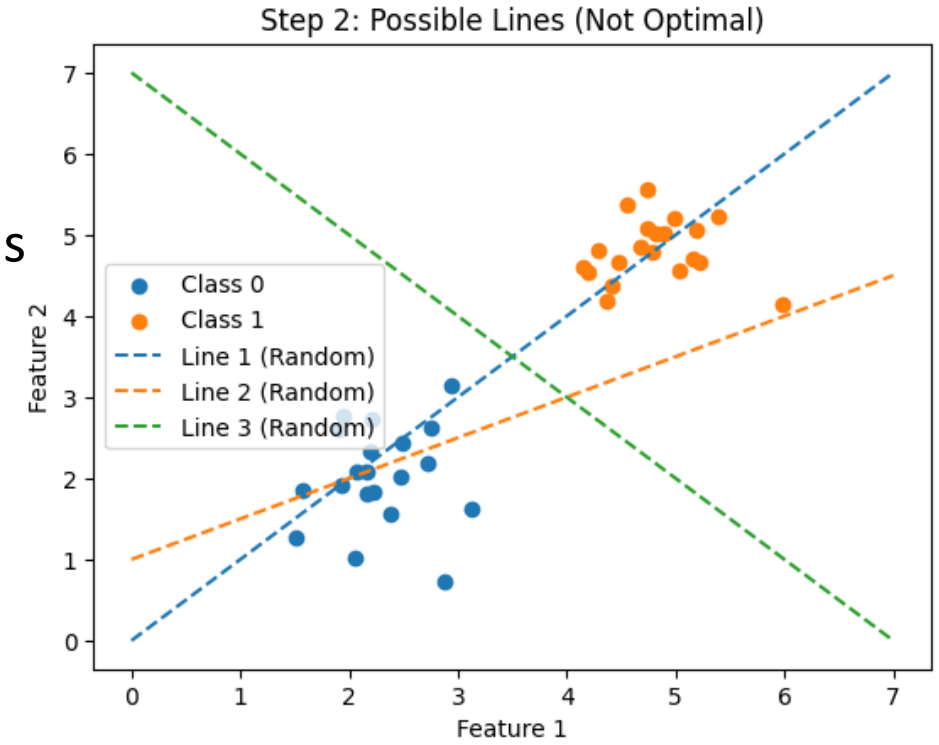
Step 2: Possible Lines (Bad Choices)

What the plot shows

- Same data as before
- Three different arbitrary lines
- These lines represent random projection directions

Why these lines are NOT good

- They cut through both classes
- When data is projected onto these lines:
  - Class overlap is high
  - Classification becomes difficult
- They do not maximize class separation



"Many lines are possible, but most of them do a poor job of separating classes."

👉 Which line should we choose?

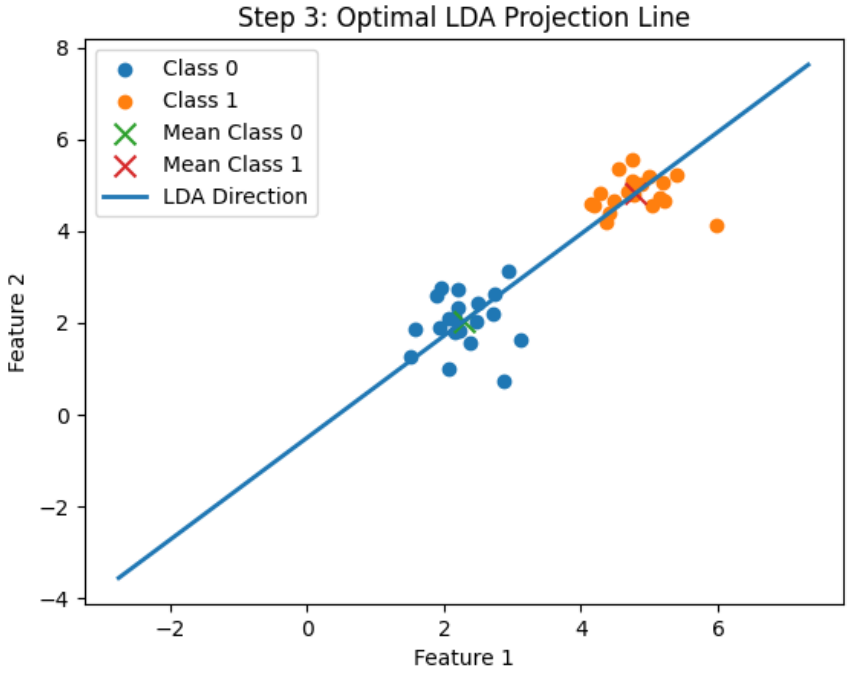
What LDA does?

LDA chooses a direction that:

1. **Maximizes distance between class means**
2. **Minimizes spread within each class**

Why this is a great choice

- The line **passes through class centers**
- Projection onto this line:
  - Pulls same-class points close together
  - Pushes different-class points far apart
- This gives **maximum class separability**



"LDA finds the line along which the classes look most different."

Key Assumptions

- For LDA to perform effectively, certain assumptions are made:
- **Gaussian Distribution:** The data in each class should follow a normal bell-shaped distribution.
- **Equal Covariance Matrices:** All classes should have the same covariance structure.
- **Linear Separability:** The data should be separable using a straight line or plane.
- For these reasons, LDA may not perform well in high-dimensional feature spaces.

Where is LDA Used?

- Face recognition
- Medical diagnosis (disease vs healthy)
- Spam vs non-spam classification
- Student performance classification

LDA

- LDA wants to maximize the ratio:

$$\frac{Distance\ between\ class\ centres}{Spread\ within\ each\ class}$$

- Large numerator → classes far apart
- Small denominator → classes tight
- **This single idea drives the entire algorithm.**

Key Quantities in LDA

- LDA works using:
  - **Mean of each class**
  - **Within-class scatter**
  - **Between-class scatter**
- These help quantify:
  - Compactness of each class
  - Separation between classes

How LDA works internally

- **Step 1: Compute class means**
  - Mean of Class 1
  - Mean of Class 2
  - These are the “centers” of each class.
- **Step 2: Measure how spread out each class is**
  - Are points tightly packed?
  - Or scattered?
  - This is called **within-class scatter**.
- **Step 3: Measure how far apart the classes are**
  - Distance between class means
  - This is **between-class scatter**.
- **Step 4: Find the direction that:**
  - ✓ Maximizes distance between means
    ✓ Minimizes spread inside classes
  - That direction = **LDA projection axis**
- **Step 5: Project data onto this axis**
  - Now classification becomes easy:
  - Just check **which side** of the line a point falls on

Example – Image Classification

- Suppose we want to **classify grayscale images of handwritten characters** into two classes:
  - **Class ω_1**: Image of digit **"0"**
  - **Class ω_2**: Image of digit **"1"**
- Each image is processed and converted into **numerical features**.

Step 1: Feature Extraction from Images

- Instead of using raw pixels, we extract **meaningful features** from each image.
- Let each image be represented by **3 features**:

Feature	Description
x_1	Average pixel intensity
x_2	Vertical symmetry score
x_3	Edge density

- So, each image = a 3-dimensional feature vector
- $\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$

Step 2: Training Data (Multiple Samples)

Class ω_1 (Digit “0”)

Sample	x_1	x_2	x_3
1	6	5	2
2	7	6	3
3	8	5	2
4	7	7	3

Class ω_2 (Digit “1”)

Sample	x_1	x_2	x_3
1	2	3	7
2	3	2	6
3	2	4	8
4	3	3	7

Step 3: Compute Class Mean Vectors

- **Mean of Class ω_1**

$$\mu_1 = \begin{bmatrix} 7 \\ 5.75 \\ 2.5 \end{bmatrix}$$

- **Mean of Class ω_2**

$$\mu_2 = \begin{bmatrix} 2.5 \\ 3 \\ 7 \end{bmatrix}$$

Step 4: Compute Within-Class Scatter Matrix (S_w)

- **Definition**

$$S_W = \sum_{i=1}^{C} \sum_{x \in D_i} (x - \mu_i)(x - \mu_i)^T$$

- This measures **how spread out each class is internally**.
- Contains variances on the diagonal
- Contains covariances off the diagonal
- After computing deviations and summing:

$$S_W = \begin{bmatrix} 6 & 3 & -2 \\ 3 & 5 & -1 \\ -2 & -1 & 4 \end{bmatrix}$$

Step 5: Compute Between-Class Scatter Matrix (S_B)

$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

$$\mu_1 - \mu_2 = \begin{bmatrix} 4.5 \\ 2.75 \\ -4.5 \end{bmatrix}$$

$$S_B = \begin{bmatrix} 20.25 & 12.38 & -20.25 \\ 12.38 & 7.56 & -12.38 \\ -20.25 & -12.38 & 20.25 \end{bmatrix}$$

Step 6: Compute LDA Projection Vector (w)

- **LDA Optimization Objective**

$$\mathbf{w} = S_W^{-1}(\mu_1 - \mu_2)$$

- After computing the inverse:

$$\mathbf{w} = \begin{bmatrix} 0.62 \\ 0.38 \\ -0.69 \end{bmatrix}$$

  - **This vector defines the optimal direction** that:
    - Maximizes class separation
    - Minimizes overlap

Step 7: Project Images onto 1D LDA Axis

- Each image is projected as:

$$y = \mathbf{w}^T \mathbf{x}$$

- **Image from Class "0"**

$$\mathbf{x} = [7, 6, 3]$$

$$y = (0.62)(7) + (0.38)(6) - (0.69)(3) = 4.45$$

- **Image from Class "1"**

$$\mathbf{x} = [2, 3, 7]$$

$$y = (0.62)(2) + (0.38)(3) - (0.69)(7) = -2.41$$

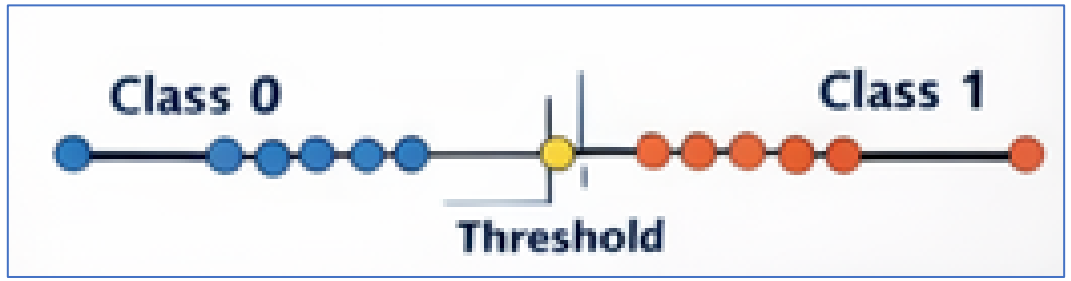
Step 8: Classification Rule

- Choose a **threshold** t(often the midpoint between projected means):

$$t \approx 1.0$$

- **Decision Rule**

$$\text{If } y > t \Rightarrow \textit{Class “0”}$$
$$\text{If } y \leq t \Rightarrow \textit{Class “1”}$$



“LDA squeezes multi-feature images onto a single line where different classes are as far apart as possible.”

Python Code

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.discriminant_analysis import
LinearDiscriminantAnalysis
from sklearn.datasets import load_iris

# Load Iris data
X, y = load_iris(return_X_y=True)

# Select ONLY two classes (0 and 1) and two features
X = X[y != 2][:, :2]   # sepal length & sepal width
y = y[y != 2]

# Fit LDA
lda = LinearDiscriminantAnalysis(n_components=1)
lda.fit(X, y)

# Scatter plot of original data
plt.scatter(X[y == 0, 0], X[y == 0, 1], label='Class 0')
plt.scatter(X[y == 1, 0], X[y == 1, 1], label='Class 1')
```

```
# LDA direction (weight vector)
w = lda.coef_[0]
w = w / np.linalg.norm(w)  # normalize

# Mean point
mean = X.mean(axis=0)

# Create LDA line
x_vals = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
y_vals = mean[1] + (w[1] / w[0]) * (x_vals - mean[0])

plt.plot(x_vals, y_vals, 'k--', label='LDA line')

# Labels
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('LDA: Data Scatter with Discriminant Line')
plt.legend()
plt.show()
```

Advantages of LDA

- Uses class label information
- Good class separability
- Reduces dimensionality
- Works well for linearly separable data

Limitations of LDA

- Assumes:
  - Normal distribution
  - Equal covariance matrices
- Not suitable for highly non-linear data

Key Takeaways

- LDA is a supervised dimensionality reduction technique
- Maximizes class separability
- Useful for both visualization and classification
- Widely used in real-world ML applications