Generative Learning Algorithms: Gaussian Discriminant Analysis (GDA) and Naive Bayes.

🧠 Summary of Key Learnings

- **GDA (Gaussian Discriminant Analysis):** A generative learning algorithm that models (P(x | y)) and (P(y)), then uses Bayes' Rule.
- **Shared Covariance:** When classes share the same covariance matrix, GDA decision boundaries are linear.
- **Naive Bayes:** Makes the simplifying assumption that features are conditionally independent given the class.
- **Generative vs. Discriminative:** GDA is generative; logistic regression is discriminative. GDA can outperform when assumptions hold.

🧪 Example Codes

```
import numpy as np

# Training GDA
def train_gda(X, y):
    m, n = X.shape
    phi = np.mean(y)
    mu0 = X[y == 0].mean(axis=0)
    mu1 = X[y == 1].mean(axis=0)
    diff0 = X[y == 0] - mu0
    diff1 = X[y == 1] - mu1
    Sigma = (diff0.T @ diff0 + diff1.T @ diff1) / m
    return phi, mu0, mu1, Sigma

def predict_gda(x, phi, mu0, mu1, Sigma):
    invSigma = np.linalg.inv(Sigma)
    p0 = -0.5 * (x - mu0).T @ invSigma @ (x - mu0) + np.log(1 - phi)
    p1 = -0.5 * (x - mu1).T @ invSigma @ (x - mu1) + np.log(phi)
    return 1 if p1 > p0 else 0
```

```
# Naive Bayes with binary features and Laplace smoothing
def train_naive_bayes(X, y):
    m, n = X.shape
    phi = np.mean(y)
    theta0 = (X[y == 0].sum(axis=0) + 1) / ((y == 0).sum() + 2)
    theta1 = (X[y == 1].sum(axis=0) + 1) / ((y == 1).sum() + 2)
    return phi, theta0, theta1

def predict_naive_bayes(x, phi, theta0, theta1):
    logp0 = np.sum((1 - x) * np.log(1 - theta0) + x * np.log(theta0)) + np.log(1 - phi)
    logp1 = np.sum((1 - x) * np.log(1 - theta1) + x * np.log(theta1)) + np.log(phi)
    return 1 if logp1 > logp0 else 0
```

```
# Sample test for GDA
from sklearn.datasets import make_classification

# Generate sample data with correct constraints
X, y = make_classification(
    n_samples=100,
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_repeated=0,
    n_classes=2,
    random_state=42
)


# Train GDA
phi, mu0, mu1, Sigma = train_gda(X, y)

# Predict one example
x_test = X[0]
pred = predict_gda(x_test, phi, mu0, mu1, Sigma)
print("GDA Prediction for first sample:", pred)
```

```
GDA Prediction for first sample: 0
```

```
# Sample binary features for Naive Bayes
X_bin = (X > X.mean(axis=0)).astype(int)  # convert to binary
phi, theta0, theta1 = train_naive_bayes(X_bin, y)

# Predict with Naive Bayes
x_test_bin = X_bin[0]
pred_nb = predict_naive_bayes(x_test_bin, phi, theta0, theta1)
print("Naive Bayes Prediction for first sample:", pred_nb)
```

```
Naive Bayes Prediction for first sample: 1
```

```
predictions = [predict_gda(x, phi, mu0, mu1, Sigma) for x in X[:5]]
print("Predictions for first 5 samples:", predictions)
print("Actual labels:                ", y[:5])
```

```
Predictions for first 5 samples: [0, 1, 0, 0, 1]
Actual labels:                  [0 1 0 0 1]
```

```
y_pred = [predict_gda(x, phi, mu0, mu1, Sigma) for x in X]
accuracy = np.mean(y_pred == y)
print("GDA Accuracy on dataset:", accuracy)
```

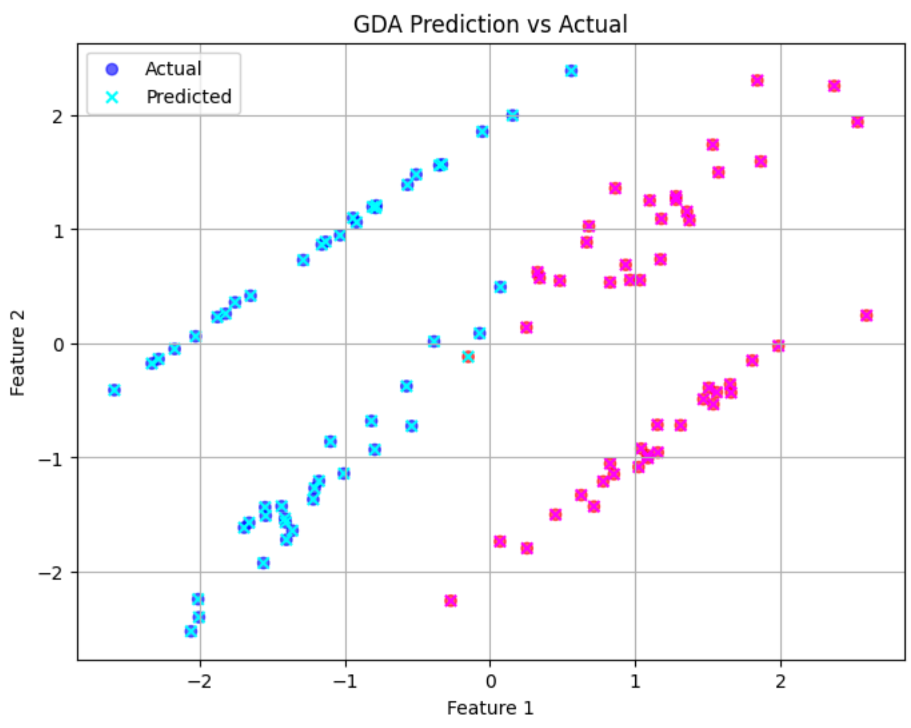
```
GDA Accuracy on dataset: 0.99
```

```
import matplotlib.pyplot as plt

plt.figure(figsize=(8,6))
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', alpha=0.6, label='Actual')

# Draw prediction overlay
y_pred = [predict_gda(x, phi, mu0, mu1, Sigma) for x in X]
plt.scatter(X[:, 0], X[:, 1], c=y_pred, cmap='cool', marker='x', label='Predicted')

plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("GDA Prediction vs Actual")
plt.legend()
plt.grid(True)
plt.show()
```



🔍 Reflection / Discussion

- What I learned from this video:

- Understanding the assumptions behind generative models helps select the right tool.
- GDA is efficient when Gaussian assumptions hold true.
- Naive Bayes is practical for high-dimensional sparse data (e.g. NLP).