Machine Learning
Course Project Report

spring 2024

12223550 문규원 | 12223557 여예진

Index

1. 서론

1.1 연구 배경 및 목적

현대 사회에서 데이터의 중요성은 날로 증가하고 있으며, 데이터 분석 및 해석 능력은 다양한 분야에서 핵심 경쟁력으로 자리 잡고 있다. 특히, 고차원 데이터를 다루는 생물학, 금융, 텍스트 분석 등 다양한 분야에서 효과적인 데이터 분류와 차원 축소 기법이 요구되고 있다. 전통적인 선형 분류 방법인 선형 판별 분석(LDA, Linear Discriminant Analysis)은 클래스 간의 분리를 극대화하면서 데이터의 차원을 축소하는 데 널리 사용되어 왔다. 그러나 LDA는 몇 가지 주요한 단점을 가지고 있다.

1. 선형성 가정
2. 등분산성 가정
3. 고차원 데이터 처리의 어려움

이 연구의 주요 목적은 기존 LDA의 단점을 보완한 새로운 차원 축소 및 분류 방법론을 제안하는 것이다. 이를 통해 고차원 데이터에서 보다 효과적인 분류 성능을 도출하고, 다양한 실제 데이터 셋에 대해 우수한 성능을 발휘할 수 있는 방법을 제시하고자 한다.

1.2 이론적 배경

1.2.1 LDA(Linear Discriminant Analysis)

LDA는 통계학과 머신러닝에서 널리 사용되는 차원 축소 및 선형 분류 기법으로, LDA의 주요 목표는 클래스 간 분산을 최대화하고 클래스 내 분산을 최소화하는 선형 판별 함수를 찾는 것이다.

LDA는 다음과 같은 단계로 이루어진다.

1. 각 클래스의 평균 벡터와 공분산 행렬을 계산한다.
2. 클래스 간 분산은 최대화하고, 클래스 내 분산은 최소화하는 방향 벡터 w를 찾는다.
3. 데이터를 새로운 축 w로 투영하여 차원을 축소하고 분류를 수행한다.

이러한 LDA는 몇 가지 가정과 한계가 존재한다.

1. **선형성 가정**: LDA는 데이터가 선형적으로 구분될 수 있다는 가정을 전제로 한다. 이는 실제로는 비선형적으로 분포된 데이터를 효과적으로 처리할 수 없게 만든다.
2. **등분산성 가정**: LDA는 각 클래스가 동일한 공분산 행렬을 가진다고 가정한다. 하지만 현실 세계의 데이터는 종종 이 가정을 만족하지 않으며, 이는 분류 성능의 저하로 이어질 수 있다.
3. **고차원 데이터 처리의 어려움**: LDA는 데이터의 차원이 매우 높을 때, 즉 차원보다 데이터의 샘플 수가 적을 때 효과적으로 작동하지 않는다. 이는 고차원 저표본 문제로, 차원의 저주

(curse of dimensionality)를 야기한다.

1.2.2 Kernel LDA(Kernel Linear Discriminant Analysis)

Kernel LDA는 LDA의 선형성 한계를 극복하기 위해 도입된 방법으로, 커널 트릭을 사용하여 데이터를 고차원 특성 공간으로 매핑하여 비선형적으로 분포된 데이터를 선형적으로 분리할 수 있다.

커널 트릭(Kernel Trick)이란 고차원 공간으로의 명시적 매핑 없이, 저차원 공간에서의 계산만으로 고차원 공간에서의 결과를 얻을 수 있도록 하는 방법이다. 두 점 x와 y 사이의 내적을 커널 함수 $k(x,y)$로 표현하여, 고차원 공간에서의 내적을 효율적으로 계산할 수 있다. 즉, kernel은 두 벡터의 내적이며 기하학적으로는 cosine 유사도를 의미한다. 다음은 주요 커널 함수로 본 연구에서는 RBF kernel(gaussian kernel)을 사용하였다.

$$
\begin{aligned}
linear \quad &: \quad K(x_1, x_2) = x_1^T x_2 \\
polynomial \quad &: \quad K(x_1, x_2) = (x_1^T x_2 + c)^d, \quad c > 0 \\
sigmoid \quad &: \quad K(x_1, x_2) = \tanh\{a(x_1^T x_2) + b\}, \quad a, b \geq 0 \\
gaussian \quad &: \quad K(x_1, x_2) = exp\left\{ -\frac{\|x_1 - x_2\|_2^2}{2\sigma^2} \right\}, \quad \sigma \neq 0
\end{aligned}
$$

Kernel LDA은 다음과 같은 과정을 거친다.

1. 커널 함수 선택 및 매핑

먼저 적절한 커널 함수를 선택하고 데이터를 고차원 공간으로 매핑한다. 앞서 언급했듯이 커널 함수는 RBF 커널을 선택했다.

$$
\boldsymbol{k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2} \right)}
$$

2. 커널 행렬 계산

데이터 $\{x_1, x_2, \ldots, x_n\}$에 대해 커널 행렬 K를 계산한다. $k(x_i, x_j)$는 커널 함수로, 커널 행렬의 각 원소 K_{ij}는 두 데이터 포인트 x_i와 x_j간의 커널 함수 값이다.

$$
\boldsymbol{K_{ij} = k(x_i, x_j)}
$$

3. 중심화된 커널 행렬 계산

중심화된 커널 행렬 $\tilde{K}$를 다음과 같이 계산한다. 여기서 1_n은 모든 원소가 $\frac{1}{n}$인 n x n 행렬이다.

$$
\boldsymbol{\tilde{K} = K - 1_n K - K 1_n + 1_n K 1_n}
$$

4. 산포 행렬 계산

고차원 공간에서의 클래스 간 산포 행렬 S_B와 클래스 내 산포 행렬 S_W를 다음과 같이 계산한다. 클래스 내 산포 행렬은 고차원 공간에서 클래스 간의 분산을 측정하고, 클래스 간 산포 행렬은 클래스 평균 간의 분산을 측정한다.

1) 클래스 내 산포 행렬 S_W

$$S_W = \sum_{i=1}^{C} \sum_{j=1}^{n_i} (\phi(x_{ij}) - \mu_i)(\phi(x_{ij}) - \mu_i)^T$$

여기서 $\emptyset(x_{ij})$는 고차원 공간으로 매핑된 데이터 벡터이다. 그러나 실제로는 $\emptyset(x)$를 명시적으로 계산하지 않고, 커널 함수로 대체한다. 이 과정은 다음 단계에서 언급하겠다.

2) 클래스 간 산포 행렬 S_B

$$S_B = \sum_{i=1}^{C} n_i(\mu_i - \mu)(\mu_i - \mu)^T$$

여기서 μ는 전체 데이터의 평균이다.

5. 고윳값 문제 해결

$\tilde{K}$를 사용하여 $\widetilde{S_B}$와 $\widetilde{S_W}$를 정의하고 이를 통해 일반화된 고유값 문제를 해결할 수 있다.

$$\widetilde{S_W}\alpha = \lambda\widetilde{S_B}\alpha$$

여기서 α는 고유벡터, λ는 고유값이다.

고유값 문제를 해결하기 위해서는 먼저 커널 행렬을 통해 $\widetilde{S_W}$와 $\widetilde{S_B}$를 구한다.

$$\widetilde{S_W} = \tilde{K}H\tilde{K}^T$$

$$\widetilde{S_B} = \tilde{K}M\tilde{K}^T$$

여기서 H는 클래스 내 산포를 나타내는 행렬, M은 클래스 간 산포를 나타내는 행렬이다.

6. 저차원 공간으로 변환

고유벡터 α를 사용하여 데이터를 저차원 공간으로 변환한다. 변환된 데이터는 다음과 같다.

$$Z = \tilde{K}\alpha$$

7. 최종 분류

저차원 공간에서 변환된 데이터 Z를 사용하여 최종 분류를 수행한다.

Kernel LDA는 LDA의 선형성을 극복하기 위해 고안된 방법이지만, 다음과 같은 몇 가지 단점을 가지고 있다:

1. 커널 함수 및 매개변수 선택의 어려움

커널 함수와 그 매개변수를 선택하는 것이 매우 중요하지만, 최적의 커널과 매개변수를 찾는 것은 쉽지 않으며 잘못된 매개변수를 선택하면 분류 성능이 크게 저하될 수 있다.

2. 높은 계산 복잡도

커널 행렬 K의 크기는 데이터 포인트 수 n에 따라 n x n이 되므로, 데이터 포인트가 많은 경우 계산량이 매우 많아진다. 특히, 고차원 공간에서의 계산은 더 많은 자원을 필요로 한다. 커널 행렬의 계산과 그에 따른 고유값 분해는 $O(n^3)$의 시간 복잡도를 가진다.

1.2.3 UMAP(Uniform Manifold Approximation and Projection)

UMAP은 이러한 Kernel LDA의 단점을 보완할 수 있는 강력한 도구로, 고차원 데이터를 저차원 공간으로 매핑하여 데이터의 전반적인 구조를 보존하는 동시에 계산 복잡도를 줄인다. UMAP의 주요 특징은 다음과 같다.

1. **리만 기하학과 대수적 위상수학 기반**: UMAP은 차원 축소를 위한 새로운 매니폴드 학습 기법으로, 리만 기하학과 대수적 위상수학을 기반으로 하여 데이터의 전반적인 구조와 분포를 효과적으로 학습한다.

2. **고차원 데이터 처리**: UMAP은 고차원 데이터의 구조를 저차원에서 효과적으로 보존하여, 차원 축소 이후에도 데이터의 중요 특성을 유지한다.

3. **계산 효율성**: UMAP은 근접 이웃 그래프와 퍼지 단순 복합체(fuzzy simplicial sets)를 이용하여 계산 복잡도를 줄인다.

UMAP의 주요 과정을 수학적으로 접근하면 다음과 같다.

1. 근접 이웃 그래프 생성
각 데이터 포인트 x_i에 대해 k개의 가장 가까운 이웃을 찾고, 가중치가 적용된 그래프 G를 만든다. 여기서 가중치는 다음과 같이 정의된다.

$$w((x_i, x_j)) = \exp\left(-\frac{\max(0, d(x_i, x_j) - \rho_i)}{\sigma_i}\right)$$

여기서 ρ_i는 x_i의 가장 가까운 이웃과의 거리, σ_i는 거리의 표준편차이다.

2. 퍼지 단순 복합체 구성
각 데이터 포인트에 대해 퍼지 단순 복합체를 구성하고, 이를 통합하여 전반적인 퍼지 단순 복합체를 형성한다. 이는 고차원 공간에서의 데이터 구조를 저차원에서 보존하는 데 도움이 된다.

3. 저차원 임베딩 최적화
저차원 임베딩 Y를 최적화하여 고차원 데이터와의 퍼지 교차 엔트로피를 최소화한다. 이 과정은 확률적 경사 하강법을 사용하여 다음과 같이 수행된다.

$$C((A,\mu),(A,\nu)) = \sum_{a\in A}\left(\mu(a)\log\left(\frac{\mu(a)}{\nu(a)}\right) + (1-\mu(a))\log\left(\frac{1-\mu(a)}{1-\nu(a)}\right)\right)$$

UMAP과 같이 언급되는 t-SNE와는 차이점이 존재한다. t-SNE(t-distributed Stochastic Neighbor Embedding)는 주로 데이터 시각화를 위한 차원 축소 기법으로, t-SNE는 고차원 데이터의 클러스터링 구조를 2차원 또는 3차원으로 시각화하는 데 탁월하지만, 계산 시간이 오래 걸리고 대규모 데이터셋에 적합하지 않은 단점이 있다. 반면, UMAP는 시각화에만 국한되지 않고 데이터 전처리와 특성 추출에도 사용할 수 있는 더 범용적인 차원 축소 기법이다. UMAP은 t-SNE보다 빠른 연산 속도를 제공하며, 데이터의 국소적 및 글로벌 구조를 모두 보존할 수 있기 때문에 효율적인 데이터 전처리 도구로 사용될 수 있다.

2. 새로운 방법론: QKULDA(Q-Kernel Optimized UMAP Linear Discriminant Analysis)

2.1 개요

이 연구는 고차원 데이터를 효과적으로 분류하기 위한 새로운 방법론 QKULDA를 제안한다. 기존 LDA의 단점을 보완하기 위해 다음과 같은 접근 방식을 도입했다.

1. kernel 도입
LDA의 선형성을 비선형성으로 확장하여 데이터가 선형적으로 구분되지 않을 때도 효과적인 분류가 가능하도록 커널 기법을 도입했다.

2. 커널 함수 파라미터 최적화를 통한 등분산성 가정의 완화
새로 설계한 Q 최적화 기준을 통해 커널 함수의 파라미터를 최적화한다. 동일 분산성을 최대화(Q1)하고 클래스 분리를 극대화(Q2)하여 다양한 분포를 가지는 클래스도 효과적으로 분류할 수 있다.

3. UMAP을 통한 고차원 데이터의 차원 축소
UMAP을 사용하여 고차원 데이터를 효과적으로 저차원으로 축소함으로써, 차원의 저주 문제를 해결하고 고차원 데이터에서의 분류 성능을 향상시킨다.

이 새로운 방법론 QKULDA는 kernel 기법과 UMAP을 결합하고, 새로 설계한 Q 최적화를 통해 커널 함수 파라미터를 최적화함으로써 LDA의 성능을 높이며, 고차원 데이터에 대한 효율적인 분류를 목표로 한다.

2.2 커널 함수의 파라미터 최적화

2.2.1 파라미터 최적화의 필요성

커널 함수의 파라미터는 Kernel LDA의 성능에 중요한 영향을 미친다. 최적의 파라미터를 선택하지 않으면, 모델의 분류 성능이 저하될 수 있고, 과대적합 또는 과소적합 문제가 발생할 수 있

다. 따라서, 파라미터 최적화는 Kernel LDA의 성능을 향상시키는 데 필수적이다.

커널 함수의 파라미터는 하이퍼파라미터이다. RBF kernel에서는 데이터 포인트 간의 유사성을 측정하는 스케일을 조정하는 파라미터인 γ와 $\gamma = \frac{1}{2\sigma^2}$에서의 σ가 하이퍼파라미터이다. 보통은 커널 함수의 파라미터를 최적화를 위해 주로 교차 검증을 사용했다. 그러나 교차 검증은 훈련 데이터에만 최적화되며 계산 비용이 높고, 모든 훈련 데이터를 최적화에 사용할 수 없다는 단점이 있다.

이 문제를 해결하기 위해 커널 함수의 최적의 하이퍼파라미터를 찾는 새로운 기준을 제시했다.

2.2.2 Q 최적화 방법론

커널 함수의 파라미터 최적화를 통해 다음 두 가지의 목표를 달성하고자 한다.

1. **동일 분산성 최대화**: 클래스 간 분포가 동일한 공분산을 가지도록 하는 것
2. **클래스 분리 극대화**: 클래스 간 거리를 최대화하여 분류 성능을 향상시키는 것

1) 동일 분산성 최대화(Q1)

Q1은 동일 분산성을 측정하는 기준으로, 두 클래스의 공분산 행렬이 얼마나 유사한지를 평가한다. 커널 공간에서의 정규 분포 $\mathcal{N}(\mu_1, \Sigma_1)$와 $\mathcal{N}(\mu_2, \Sigma_2)$ 의 커널 공분산 행렬이 있을 때, 아래와 같은 기준 Q1을 정의했다.

$$Q1 = \frac{1}{2}\left(\frac{\mathrm{tr}(\Sigma_1 {\Sigma_2}^{-1}) + \mathrm{tr}(\Sigma_2 {\Sigma_1}^{-1})}{2}\right)$$

여기서 Σ_1과 Σ_2는 각각 클래스 1과 클래스 2의 공분산 행렬이고, tr(.)은 행렬의 대각합(trace)이다. 식의 도출 과정은 다음과 같이 진행되었다.

1. 유사성 측정
$\mathrm{tr}(\Sigma_1 {\Sigma_2}^{-1})$는 Σ_1이 Σ_2과 얼마나 유사한지를 나타내는 척도로, Σ_1을 Σ_2로 변환했을 때의 변형 정도를 의미한다. $\mathrm{tr}(\Sigma_2 {\Sigma_1}^{-1})$는 Σ_2가 Σ_1와 얼마나 유사한지를 나타내는 척도이다.

2. 평균화
두 유사성을 평균내어 대칭성을 확보했다. 즉, $\mathrm{tr}(\Sigma_1 {\Sigma_2}^{-1})$과 $\mathrm{tr}(\Sigma_2 {\Sigma_1}^{-1})$의 평균을 구하여 어느 한쪽으로 치우치지 않도록 했다.

3. 스케일 조정
1/2를 곱하여 전체 값을 스케일 조정했다. 이는 수식의 값이 항상 0.5를 넘지 않도록 하여 최대값이 0.5가 되도록 했다.

동일 분산성을 최대화한다는 것은 두 클래스의 공분산 행렬이 최대한 동일하게 된다는 것을 의미한다. Q1이 0.5가 되는 조건은 다음과 같다.

- $\Sigma_1 = \Sigma_2$일 때, $\Sigma_1 {\Sigma_2}^{-1}$와 $\Sigma_2 {\Sigma_1}^{-1}$는 단위 행렬이 된다. 단위 행렬의 대각합은 행렬의 크기와 같으므로, $\mathrm{tr(I)} = d$(여기서 d는 행렬의 차원 수)이다.
- 따라서 $\mathrm{tr}(\Sigma_1 {\Sigma_2}^{-1}) = \mathrm{tr}(\Sigma_2 {\Sigma_1}^{-1}) = d$이다.

- 이를 식에 대입하면 $\frac{1}{2}\left(\frac{d+d}{2}\right) = 0.5$가 된다.

즉, Q1이 0.5가 되는 조건은 두 클래스의 공분산 행렬이 동일할 때이다. 따라서 Q1을 최대화하는 것은 클래스 간의 동일 분산성을 최대화하는 것과 동일하다. 이는 기존 LDA의 등분산성 가정을 최대한 안고 가려는 의도로, 클래스 간의 데이터 분포가 유사하게 되도록 커널을 선택하거나 조정하는데 도움을 준다.

```python
def calculate_Q1(self, K_centered, y):
    unique_classes = np.unique(y)
    num_classes = len(unique_classes)
    covariance_matrices = []

    for cls in unique_classes:
        K_c = K_centered[y == cls]
        covariance_c = np.cov(K_c, rowvar=False) + 1e-10 * np.eye(K_c.shape[1])
        covariance_matrices.append(covariance_c)

    Q1_num = 0
    for i in range(num_classes):
        for j in range(num_classes):
            Q1_num += np.trace(np.linalg.inv(covariance_matrices[i]) @ covariance_matrices[j])

    Q1 = 0.5 * (Q1_num / num_classes)
    return Q1
```

2) 클래스 분리 극대화(Q2)

$$Q2 = \mathrm{tr}(S_B)$$
$$S_B = \sum_{i=1}^{C} n_i(\mu_i - \mu)(\mu_i - \mu)^T$$

Q2는 클래스 간 분산을 최대화하여 서로 다른 클래스들의 데이터가 더 잘 분리되도록 하기 위해 도입한 것으로, 클래스 간 분산 행렬(S_B)의 trace를 계산하는 방식으로 구현했다. 클래스 간 분산 행렬 S_B는 위와 같이 정의된다. 식의 도출 과정은 다음과 같이 진행되었다.

1. 클래스 중심 간 거리 측정

S_B는 각 클래스 중심(평균 벡터)과 전체 데이터의 평균 벡터 간의 거리를 측정하며, 클래스 중심 간의 거리가 멀수록 클래스 분리 가능성이 커진다.

2. 분산의 총합

행렬의 트레이스(tr)는 대각합으로, S_B의 트레이스는 클래스 중심 간의 분산의 총합을 의미한다. 이는 모든 클래스 중심 간의 거리의 제곱합과 같다.

3. 클래스 분리 극대화

$\mathrm{tr}(S_B)$를 최대화하면 클래스 간의 중심 간의 거리를 최대화하는 효과가 있을 것이기 때문에 데이터 분류 성능을 향상시킬 것이라고 판단했다.

```python
def calculate_Q2(self, K_centered, y):
    unique_classes = np.unique(y)
    mean_overall = np.mean(K_centered, axis=0)

    S_B = np.zeros((K_centered.shape[1], K_centered.shape[1]))
    for cls in unique_classes:
        K_c = K_centered[y == cls]
        n_c = K_c.shape[0]
        mean_c = np.mean(K_c, axis=0)
        mean_diff = (mean_c - mean_overall).reshape(-1, 1)
        S_B += n_c * np.dot(mean_diff, mean_diff.T)

    Q2 = np.trace(S_B)
    return Q2
```

3-1) 최적화 기준 목적함수 Q 설정 : $\mathrm{Q} = Q1 \times \mathrm{Q2}$

초기에 설계한 목적함수는 동일 분산성과 클래스 분리 두 기준을 모두 최대화하여 최적의 성능을 달성하도록 곱셈 방식으로 설계했다. 동일 분산성(Q1)과 클래스 분리 가능성(Q2)을 동시에 최대화되어야 Q가 최대화되어 최적화가 이루어지기 때문에 한쪽 값이 매우 작으면 Q도 매우 작어져 최적화가 어려워질 수 있다는 한계를 발견했다.

3-2) 최적화 기준 목적함수 Q 재설정 : $\mathrm{Q} = \lambda 1 \times Q1 + \lambda 2 \times \mathrm{Q2}$

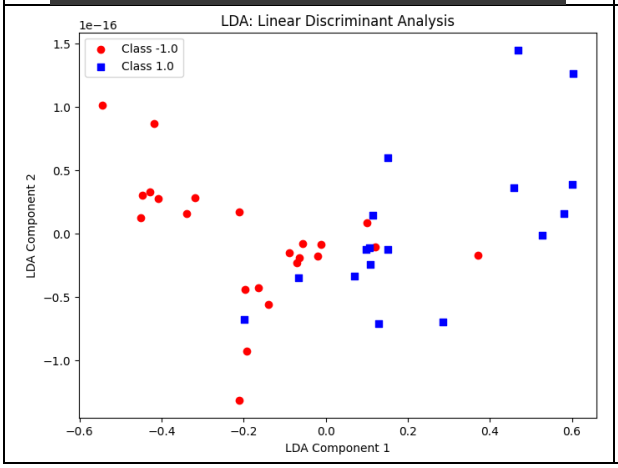
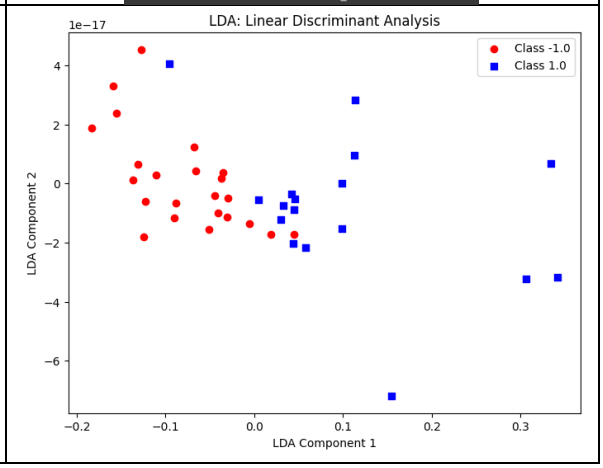
3-1에서의 목적함수의 한계를 극복하기 위해 두 기준의 상대적 중요성을 조절할 수 있는 람다를 도입하여 Q를 Q1과 Q2의 가중합으로 정의하였다. 람다를 도입한 $\mathrm{Q} = \lambda 1 \times Q1 + \lambda 2 \times \mathrm{Q2}$를 목적함수로 설정했을 때 다음과 같은 효과를 기대했고, 이는 실제로 QKULDA의 성능으로 최적의 목적함수임을 확인할 수 있었다.

1. 유연성 증가
   가중 합 방식은 $\lambda 1$과 $\lambda 2$를 통해 각 기준의 중요성을 조절할 수 있는 유연성을 제공한다. 이는 모델이 특정 기준에 더 중점을 두어야 할 때 유용하다.

2. 안정성 향상
   3-1의 곱셈 방식은 두 기준 중 하나가 매우 작으면 전체 Q값이 작아지는 문제를 가졌다. 반면, 가중 합 방식은 한 기준이 작더라도 다른 기준이 충분히 크다면 Q값이 크게 유지될 수 있기 때문에 최적화 과정에서 더 안정적인 결과를 도출할 수 있다.

3. 성능 개선
   가중 합 방식은 두 기준을 적절히 조절함으로써, 모델이 한 기준에 치우치지 않고 균형 잡힌 성능을 발휘할 수 있기 때문에 더 나은 일반화 성능을 기대할 수 있다.

각 방식의 목적함수에 따른 성능 차이는 다음과 같다.

곱셈 방식 : $Q = Q1 \times Q2$	가중 합 방식 : $Q = \lambda 1 \times Q1 + \lambda 2 \times Q2$
Mean Accuracy: 0.8800000000000001	Mean Accuracy: 0.925
	

QKULDA에 서로 다른 방식의 목적함수를 도입했을 때에 가중 합 방식이 더 높은 성능을 보임을 확인할 수 있다.

2.3 UMAP 도입 (1.2.3 UMAP(Uniform Manifold Approximation and Projection) 참조)

LDA는 데이터 분류와 차원 축소에서 널리 사용되는 기법이지만 고차원 데이터와 복잡한 비선형 경계를 가진 데이터셋을 처리하는 데는 한계가 있어 이를 극복하기 위해 LDA에 kernel을 적용하고 파라미터 최적화의 어려움을 Q 최적화를 도입하여 해결했다. 그러나 kernel LDA에는 여전히 고차원 데이터셋에서 높은 계산 복잡도의 문제를 갖고 있다. 이러한 문제를 해결하기 위해 UMAP을 도입하여 고차원 데이터를 저차원 공간으로 변환하여 데이터의 전반적인 구조를 보존하면서도 계산 복잡도를 크게 줄일 수 있도록 했다.

UMAP를 Kernel LDA의 전처리 단계로 도입하면 다음과 같은 이점을 얻을 수 있다.

1. 차원 축소: UMAP를 사용하여 고차원 데이터를 저차원으로 축소함으로써, 커널 행렬의 크기를 줄이고 계산 복잡도를 감소시킨다.

2. 구조 보존: UMAP는 데이터의 전반적인 구조를 보존하여, Kernel LDA가 저차원 공간에서도 효과적으로 수 있다.

3. 계산 자원 절약: 차원 축소를 통해 메모리 사용과 계산 시간을 절약할 수 있다.

UMAP의 자세한 원리는 앞의 1.2.3 UMAP(Uniform Manifold Approximation and Projection) 부분을 참조하면 된다.

3. 실험 및 결과

3.1 데이터셋

3.1.1 데이터셋 개요

Arcene 데이터셋은 UCI 머신러닝 저장소에서 제공하는 데이터셋으로, 암과 정상 패턴을 구분하는 이진 분류 작업에 사용된다. 이 데이터셋은 질량분석 데이터로 구성되어 있으며, 900개의 인스턴스와 10,000개의 연속형 특징(feature)을 포함하고 있다. 원래 특징들은 인간 혈청의 특정 질량 값을 가진 단백질의 양을 나타내며, 'probes'라고 불리는 예측력이 없는 방해 특징들도 포함되어 있다. 데이터의 특성상 매우 고차원적(10,000개의 특징)이며, 비선형적인 경향을 보이기 때문에 기존 LDA로 좋은 성능을 보장하기에는 힘든 데이터셋이다.
(dataset : https://archive.ics.uci.edu/dataset/167/arcene)

3.1.2 데이터셋 구성

훈련 및 검증 데이터셋 모두 100개의 샘플과 10,000개의 특징을 포함하고 있으며, 레이블은 각각 44개의 클래스 1과 56개의 클래스 -1으로 구성되어 있다.

```
--- Training Dataset Info ---
Number of samples: 100
Number of features: 10000
Number of '1' labels: 44
Number of '-1' labels: 56
Unique label values: [-1.  1.]
--- Validation Dataset Info ---
Number of samples: 100
Number of features: 10000
Number of '1' labels: 44
Number of '-1' labels: 56
Unique label values: [-1.  1.]
```

- 훈련 데이터, 검증 데이터: 100개의 샘플과 10,000개의 특징으로 구성.
- 훈련 레이블, 검증 레이블 각 샘플에 대한 이진 레이블(1:양성, -1:음성)로 구성.

3.1.2 데이터 전처리

1) 데이터 로드 및 전처리

DataLoader 클래스는 데이터 파일을 로드하고 표준화하는 기능을 제공한다.

```
class DataLoader:
    def __init__(self, train_data_path, train_labels_path, valid_data_path, valid_labels_path):
        self.train_data_path = train_data_path
        self.train_labels_path = train_labels_path
        self.valid_data_path = valid_data_path
        self.valid_labels_path = valid_labels_path
```

훈련 데이터, 훈련 레이블, 검증 데이터, 검증 레이블의 파일 경로를 입력으로 받고, 파일 경로를 변수로 저장한다.

```
    def load_data(self):
        train_data = np.genfromtxt(self.train_data_path)
        train_labels = np.genfromtxt(self.train_labels_path)
        valid_data = np.genfromtxt(self.valid_data_path)
        valid_labels = np.genfromtxt(self.valid_labels_path)

        X = np.vstack((train_data, valid_data))
        y = np.hstack((train_labels, valid_labels))

        return X, y
```

load_data 메서드는 np.genfromtxt 함수를 사용하여 텍스트 파일에서 데이터를 읽어온다. 훈련 데이터와 검증 데이터를 수직으로 결합하여 X에 저장하고, 레이블을 수평으로 결합하여 y에 저장한 후 결합된 데이터와 레이블을 반환한다.

```
    def standardize_data(self, X):
        scaler = StandardScaler()
        return scaler.fit_transform(X)
```

standardize_data 메서드는 StandardScaler를 사용하여 데이터를 표준화한다. 이를 통해 평균이 0, 분산이 1이 되도록 변환한다.

3.2 모델 구현

3.2.1 주요 진행 순서

이 모델의 주요 진행 순서는 다음과 같다.

1. UMAP을 이용한 차원 축소
2. Q 최적화를 통한 커널 함수의 파라미터(sigma) 최적화
3. 최적의 매개변수 탐색 (λ1, λ2)
4. Kernel LDA 적용
5. LDA를 이용한 분류
6. 교차 검증을 통한 모델 평가

3.2.2 구현 내용

QKULDA(Q-Kernel Optimized UMAP Linear Discriminant Analysis)는 UMAP을 사용하여 차원 축소를 수행하고, Kernel LDA를 사용하여 분류 성능을 극대화한다. 특히 Q 최적화를 통해 커널 함수의 하이퍼파라미터를 최적화하는 것이 특징이다. QKULDA를 class로 구현했고, 자세한 내용은 다음과 같다.

```
class QKULDA:
    def __init__(self, n_components=100, lambda1=0.5, lambda2=0.5, reg_param=0.1, random_state=42):
        self.n_components = n_components
        self.lambda1 = lambda1
        self.lambda2 = lambda2
        self.reg_param = reg_param
        self.random_state = random_state
        self.umap = UMAP(n_components=self.n_components, random_state=self.random_state)
        self.weights = None
        self.means = None
        self.classes = None
```

QKULDA 클래스는 다음과 같은 매개변수로 초기화된다. UMAP 인스턴스를 초기화하고, 모델 가중치, 클래스 평균, 고유 클래스의 자리 표시자를 설정한다.

- n_components: UMAP을 사용하여 데이터의 차원을 줄일 때 사용할 컴포넌트 수 (기본값은 100).
- lambda1, lambda2: Q 값을 계산할 때 사용하는 정규화 매개변수 (기본값은 0.5).

- reg_param: 선형 판별 분석의 정규화 매개변수 (기본값은 0.1).
- random_state: 랜덤 숫자 생성 시드 (기본값은 42).

1) UMAP 차원 축소 (fit_transform_umap 메서드)

```
def fit_transform_umap(self, X):
    return self.umap.fit_transform(X)
```

UMAP을 사용하려 입력 데이터 X의 차원을 축소하고 축소된 데이터를 반환한다. 'n_components = 100'으로 설정하여 데이터의 특성 수를 100개로 축소했다. 차원의 수를 100으로 설정한 이유는 모델 성능, 정보 보존 등을 고려한 결과이다. 다양한 차원 수를 실험한 결과 50~100차원이 가장 좋은 성능을 보였고, 그 이상의 차원에선 과적합이 발생했다.

2) RBF커널 계산 (compute_rbf_kernel 메서드)

```
def compute_rbf_kernel(self, X, gamma=None):
    if gamma is None:
        gamma = 1.0 / X.shape[1]
    return rbf_kernel(X, gamma=gamma)
```

rbf_kernel 함수를 사용하여 RBF 커널 행렬 K를 계산하고 반환한다. X는 입력 데이터 행렬, gamma는 RBF 커널의 하이퍼파라미터이다.

3) Kernel LDA (klda 메서드)

```
def klda(self, X, y, num_components=2, gamma=None):
    K = self.compute_rbf_kernel(X, gamma=gamma)
    n_samples = K.shape[0]

    one_n = np.ones((n_samples, n_samples)) / n_samples
    K_centered = K - one_n @ K - K @ one_n + one_n @ K @ one_n

    unique_classes = np.unique(y)
    num_classes = len(unique_classes)
    N = np.zeros((n_samples, num_classes))
    for idx, cls in enumerate(unique_classes):
        N[:, idx] = (y == cls).astype(float) / np.sum(y == cls)

    M = K_centered @ N
    M_centered = M - np.mean(M, axis=0)

    eigvals, eigvecs = np.linalg.eigh(M_centered.T @ M_centered)
    sorted_indices = np.argsort(eigvals)[::-1]
    eigvals = eigvals[sorted_indices]
    eigvecs = eigvecs[:, sorted_indices]

    W = eigvecs[:, :num_components]
    Z = K_centered @ M_centered @ W

    return Z
```

klda는 입력 데이터 X와 레이블 y를 사용하여 커널 LDA(Kernel Linear Discriminant Analysis)를 수행하고, 저차원 임베딩 Z를 반환한다. klda를 단계별로 설명하면 다음과 같다.

1. compute_rbf_kernel 메서드를 사용하여 RBF 커널 행렬 K를 계산한다.
2. 커널 행렬 K를 중심화하여 K_centered를 얻는다.
3. 클래스별 평균 행렬 N을 계산하고, 이를 사용하여 M과 중심화된 M_centered를 계산한다.
4. M_centered.T @ M_centered의 고유값 분해를 수행하여 고유값과 고유벡터를 얻는다
5. 상위 num_components개의 고유벡터를 선택하여 변환 행렬 W를 생성하고, 이를 사용하여 저차원 임베딩 Z를 반환한다.

4) Q1 계산 (calculate_Q1 메서드)

```
def calculate_Q1(self, K_centered, y):
    unique_classes = np.unique(y)
    num_classes = len(unique_classes)
    covariance_matrices = []

    for cls in unique_classes:
        K_c = K_centered[y == cls]
        covariance_c = np.cov(K_c, rowvar=False) + 1e-10 * np.eye(K_c.shape[1])
        covariance_matrices.append(covariance_c)

    Q1_num = 0
    for i in range(num_classes):
        for j in range(num_classes):
            Q1_num += np.trace(np.linalg.inv(covariance_matrices[i]) @ covariance_matrices[j])

    Q1 = 0.5 * (Q1_num / num_classes)
    return Q1
```

(자세한 내용은 본문 2.2.2 Q 최적화 방법론 참고)

5) Q2 계산 (calculate_Q2 메서드)

```
def calculate_Q2(self, K_centered, y):
    unique_classes = np.unique(y)
    mean_overall = np.mean(K_centered, axis=0)

    S_B = np.zeros((K_centered.shape[1], K_centered.shape[1]))
    for cls in unique_classes:
        K_c = K_centered[y == cls]
        n_c = K_c.shape[0]
        mean_c = np.mean(K_c, axis=0)
        mean_diff = (mean_c - mean_overall).reshape(-1, 1)
        S_B += n_c * np.dot(mean_diff, mean_diff.T)

    Q2 = np.trace(S_B)
    return Q2
```

(자세한 내용은 본문 2.2.2 Q 최적화 방법론 참고)

5) Q값 계산 (calculate_Q 메서드)

```
def calculate_Q(self, X, y, sigma):
    K = rbf_kernel(X, gamma=1.0 / (2 * sigma ** 2))
    n_samples = K.shape[0]

    one_n = np.ones((n_samples, n_samples)) / n_samples
    K_centered = K - one_n @ K - K @ one_n + one_n @ K @ one_n

    Q1 = self.calculate_Q1(K_centered, y)
    Q2 = self.calculate_Q2(K_centered, y)

    Q = self.lambda1 * Q1 + self.lambda2 * Q2
    return Q
```

'calculate_Q' 메서드는 Q1과 Q2를 계산한 후 가중합을 통해 최종 Q 값을 반환한다. (자세한 내용은 본문 2.2.2 Q 최적화 방법론 참고)

6) 최적의 sigma 찾기 (optimize_sigma 메서드)

```
def optimize_sigma(self, X, y):
    def objective(sigma):
        Q_value = self.calculate_Q(X, y, sigma)
        return -Q_value

    initial_sigma = np.sqrt(np.mean(np.var(X, axis=0)))
    result = minimize(objective, initial_sigma, method='Nelder-Mead', bounds=[(1e-3, 1e3)])
    return result.x[0]
```

주어진 데이터에 대해 최적의 sigma 값을 찾는다.

7) 모델 학습 (fit 메서드)

```
def fit(self, X, y):
    optimal_sigma = self.optimize_sigma(X, y)
    gamma_optimal = 1.0 / (2 * optimal_sigma ** 2)
    Z = self.klda(X, y, num_components=2, gamma=gamma_optimal)

    n_samples, n_features = Z.shape
    self.classes = np.unique(y)
    n_classes = len(self.classes)

    self.means = np.zeros((n_classes, n_features))
    for idx, cls in enumerate(self.classes):
        Z_c = Z[y == cls]
        self.means[idx, :] = np.mean(Z_c, axis=0)

    SW = np.zeros((n_features, n_features))
    for idx, cls in enumerate(self.classes):
        Z_c = Z[y == cls]
        SW += np.dot((Z_c - self.means[idx]).T, (Z_c - self.means[idx]))

    overall_mean = np.mean(Z, axis=0)
    SB = np.zeros((n_features, n_features))
    for idx, cls in enumerate(self.classes):
        n_c = Z[y == cls].shape[0]
        mean_diff = (self.means[idx] - overall_mean).reshape(n_features, 1)
        SB += n_c * np.dot(mean_diff, mean_diff.T)

    A = np.linalg.pinv(SW + self.reg_param * np.eye(n_features)).dot(SB)
    eigvals, eigvecs = np.linalg.eig(A)
    eigvecs = eigvecs.T
    idxs = np.argsort(abs(eigvals))[::-1]
    self.weights = eigvecs[idxs]
```

주어진 데이터와 레이블을 사용하여 모델을 학습시키는 과정으로, 입력 데이터 X와 레이블 y를 갖고 Q최적화를 통해 최적의 시그마 값을 찾고, 커널 LDA를 수행하여 학습한다. fit 메서드의 모델 학습 단계 다음과 같다.

1. 최적의 시그마 값을 찾는다
2. 커널 LDA를 수행하여 저차원 임베딩을 얻는다.
3. 클래스별 평균을 계산한다.
4. 클래스 내 공분산 행렬과 클래스 간 공분산 행렬을 계산한다.
5. 고유값 분해를 통해 LDA의 변환 행렬을 구한다.

8) 예측 (predict 메서드)

```
def predict(self, X):
    Z = self.klda(X, np.zeros(X.shape[0]), num_components=2, gamma=1.0 / (2 * self.optimize_sigma(X, np.zeros(X.shape[0])) ** 2))
    projected_means = np.dot(self.means, self.weights.T)
    projections = np.dot(Z, self.weights.T)
    y_pred = np.zeros(Z.shape[0])
    for i, projection in enumerate(projections):
        distances = np.linalg.norm(projection - projected_means, axis=1)
        y_pred[i] = self.classes[np.argmin(distances)]
    return y_pred
```

학습된 모델을 사용하여 새로운 입력 데이터 X의 레이블을 예측한다. 각 단계는 다음과 같다.

1. 입력 데이터 X에 대해 커널 LDA를 수행하여 저차원 임베딩 Z를 얻는다.
2. 학습된 변환 행렬을 사용하여 클래스별 평균을 변환한다.
3. 입력 데이터 X를 변환하여 저차원 임베딩 Z를 변환한다.
4. 변환된 데이터 포인트와 변환된 클래스별 평균 간의 거리를 계산하여 가장 가까운 클래스를 예측한다.

9) 교차 검증 (cross_validate 메서드)

```
def cross_validate(self, X_umap, y):
    best_lambda1, best_lambda2, best_score = self.find_best_params(X_umap, y)
    cv = StratifiedKFold(n_splits=5)

    accuracy_scores = []
    conf_matrices = []
    class_reports = []

    for train_index, test_index in cv.split(X_umap, y):
        X_train, X_test = X_umap[train_index], X_umap[test_index]
        y_train, y_test = y[train_index], y[test_index]

        optimal_sigma = self.optimize_sigma(X_train, y_train)
        gamma_optimal = 1.0 / (2 * optimal_sigma ** 2)

        Z_train_klda = self.klda(X_train, y_train, num_components=2, gamma=gamma_optimal)
        Z_test_klda = self.klda(X_test, y_test, num_components=2, gamma=gamma_optimal)

        self.fit_lda(Z_train_klda, y_train)
        y_pred = self.predict_lda(Z_test_klda)

        accuracy = accuracy_score(y_test, y_pred)
        conf_matrix = confusion_matrix(y_test, y_pred)
        class_report = classification_report(y_test, y_pred)

        accuracy_scores.append(accuracy)
        conf_matrices.append(conf_matrix)
        class_reports.append(class_report)

    print(f'Best λ1: {best_lambda1}, Best λ2: {best_lambda2}, Best Accuracy: {best_score}')
    print(f'Mean Accuracy: {np.mean(accuracy_scores)}')
    print('Confusion Matrices:')
    for cm in conf_matrices:
        print(cm)
    print('Classification Reports:')
    for cr in class_reports:
        print(cr)

    return Z_test_klda, y[test_index]
```

교차 검증을 수행하여 최적의 lambda1, lambda2 값을 찾고, 모델의 성능을 평가한다. 각 단계에 대한 find_best_params 메서드를 통해 최적의 λ1과 λ2 값을 찾고, StratifiedKFold를 사용하여 5-폴드 교차 검증을 설정하고 각 폴드에 대해 모델을 학습 및 평가한다.

10) 최적의 파라미터 찾기 (find_best_params 메서드)

```python
def find_best_params(self, X_umap, y):
    lambda1_values = [0.1, 0.5, 1, 2, 5]
    lambda2_values = [0.1, 0.5, 1, 2, 5]
    best_lambda1 = 0
    best_lambda2 = 0
    best_score = -np.inf

    for lambda1 in lambda1_values:
        for lambda2 in lambda2_values:
            cv = StratifiedKFold(n_splits=5)
            temp_scores = []

            for train_index, test_index in cv.split(X_umap, y):
                X_train, X_test = X_umap[train_index], X_umap[test_index]
                y_train, y_test = y[train_index], y[test_index]

                self.lambda1 = lambda1
                self.lambda2 = lambda2
                optimal_sigma = self.optimize_sigma(X_train, y_train)
                gamma_optimal = 1.0 / (2 * optimal_sigma ** 2)

                Z_train_klda = self.klda(X_train, y_train, num_components=2, gamma=gamma_optimal)
                Z_test_klda = self.klda(X_test, y_test, num_components=2, gamma=gamma_optimal)

                self.fit_lda(Z_train_klda, y_train)
                y_pred = self.predict_lda(Z_test_klda)

                accuracy = accuracy_score(y_test, y_pred)
                temp_scores.append(accuracy)

            mean_score = np.mean(temp_scores)
            if mean_score > best_score:
                best_score = mean_score
                best_lambda1 = lambda1
                best_lambda2 = lambda2

    self.lambda1 = best_lambda1
    self.lambda2 = best_lambda2
    return best_lambda1, best_lambda2, best_score
```

다양한 lambda1, lambda2 값 조합에 대해 교차 검증을 수행하여 최적의 파라미터를 찾는다. 여러 값의 λ1과 λ2를 시험해 보고, 5-폴드 교차 검증을 통해 평균 정확도가 가장 높은 조합을 선택하는 방식으로 이루어진다.

아래 그림은 교차 검증 과정에서 다양한 lambda1, lambda2 값 조합에 따른 초기 시그마값과 최적 시그마값의 변화의 일부를 출력한 것이다. 이를 통해 Q 최적화를 통해 시그마가 최적화되고 있음을 알 수 있다.

```
Testing λ1: 0.1, λ2: 0.1
Initial Sigma: 1.0802619457244873
Optimal Sigma: 0.2870467306865422
Initial Sigma: 1.0827503204345703
Optimal Sigma: 0.2452183095033314
Initial Sigma: 1.091261863708496
Optimal Sigma: 0.2543924231877223
Initial Sigma: 1.08285653591156
Optimal Sigma: 0.26975930632152423
Initial Sigma: 1.0848419666290283
Optimal Sigma: 0.25471006782728384
```

```
Testing λ1: 0.1, λ2: 1
Initial Sigma: 1.0802619457244873
Optimal Sigma: 0.2870467306865422
Initial Sigma: 1.0827503204345703
Optimal Sigma: 0.2452183095033314
Initial Sigma: 1.091261863708496
Optimal Sigma: 0.2543924231877223
Initial Sigma: 1.08285653591156
Optimal Sigma: 0.26975930632152423
Initial Sigma: 1.0848419666290283
Optimal Sigma: 0.25471006782728384
```

```
Testing λ1: 0.1, λ2: 0.5
Initial Sigma: 1.0802619457244873
Optimal Sigma: 0.2870467306865422
Initial Sigma: 1.0827503204345703
Optimal Sigma: 0.2452183095033314
Initial Sigma: 1.091261863708496
Optimal Sigma: 0.2543924231877223
Initial Sigma: 1.08285653591156
Optimal Sigma: 0.26975930632152423
Initial Sigma: 1.0848419666290283
Optimal Sigma: 0.25471006782728384
```

11) LDA 학습 (fit_lda 메서드)

```
def fit_lda(self, X, y):
    n_samples, n_features = X.shape
    self.classes = np.unique(y)
    n_classes = len(self.classes)

    self.means = np.zeros((n_classes, n_features))
    for idx, cls in enumerate(self.classes):
        X_c = X[y == cls]
        self.means[idx, :] = np.mean(X_c, axis=0)

    SW = np.zeros((n_features, n_features))
    for idx, cls in enumerate(self.classes):
        X_c = X[y == cls]
        SW += np.dot((X_c - self.means[idx]).T, (X_c - self.means[idx]))

    overall_mean = np.mean(X, axis=0)
    SB = np.zeros((n_features, n_features))
    for idx, cls in enumerate(self.classes):
        n_c = X[y == cls].shape[0]
        mean_diff = (self.means[idx] - overall_mean).reshape(n_features, 1)
        SB += n_c * np.dot(mean_diff, mean_diff.T)

    A = np.linalg.pinv(SW + self.reg_param * np.eye(n_features)).dot(SB)
    eigvals, eigvecs = np.linalg.eig(A)
    eigvecs = eigvecs.T
    idxs = np.argsort(abs(eigvals))[::-1]
    self.weights = eigvecs[idxs]
```

입력 데이터 X와 레이블 y를 사용하여 LDA 모델을 학습시킨다. 주요 단계는 다음과 같다.

1. 각 클래스의 평균을 계산.
2. 클래스 내 공분산 행렬(SW)을 계산.
3. 클래스 간 공분산 행렬(SB)을 계산.
4. SW와 SB를 사용하여 LDA 변환 행렬(weights)을 계산.

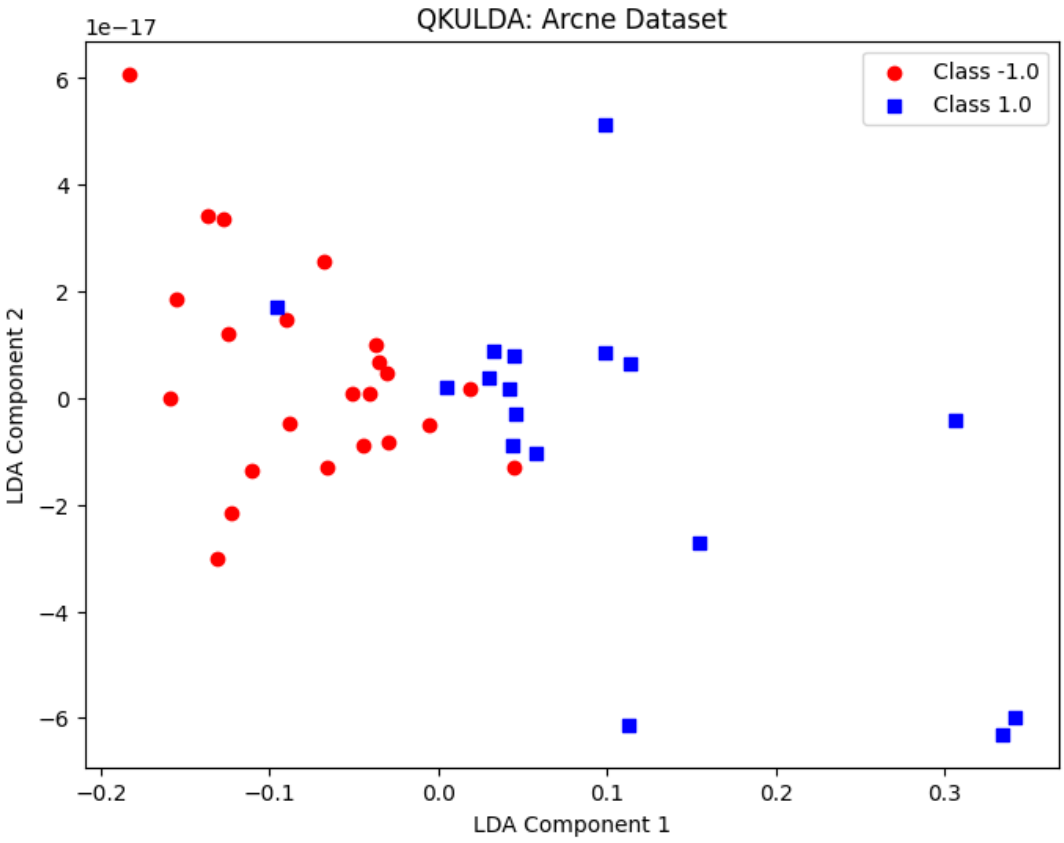
12) LDA 예측 (predict_lda 메서드)

```
def predict_lda(self, X):
    projected_means = np.dot(self.means, self.weights.T)
    projections = np.dot(X, self.weights.T)
    y_pred = np.zeros(X.shape[0])
    for i, projection in enumerate(projections):
        distances = np.linalg.norm(projection - projected_means, axis=1)
        y_pred[i] = self.classes[np.argmin(distances)]
    return y_pred
```

입력 데이터 X를 학습된 LDA 모델을 통해 변환하고, 변환된 데이터와 클래스별 평균의 거리를 계산하여 가장 가까운 클래스를 예측한다.

3.2.3 결과 시각화

QKULDA 모델을 사용하여 Arcene 데이터셋을 학습하고 교차 검증한 결과는 다음과 같다. 각 클래스는 빨간색 원(Class -1.0)과 파란색 사각형(Class 1.0)으로 표시되어있다.



1. 최적의 파라미터

- Best λ1: 0.1
- Best λ2: 0.1
- Best Accuracy: 0.925

2. 평균 정확도

- Mean Accuracy: 0.925

2. 혼동 행렬

1. [[21, 1], [3, 15]]
2. [[20, 2], [1, 17]]
3. [[19, 3], [0, 18]]
4. [[22, 1], [0, 17]]
5. [[21, 2], [2, 15]]

결과적으로 QKULDA 모델은 데이터를 효과적으로 학습하고, 높은 정확도로 분류할 수 있음을 보여준다. 특히, Q 최적화를 통해 λ1과 λ2 값을 조정하여 최적의 성능을 얻었다. 각 클래스 간의 분

리가 잘 이루어졌고, 시각화된 결과는 모델이 효과적으로 작동했음을 나타낸다.

3.3 성능 비교

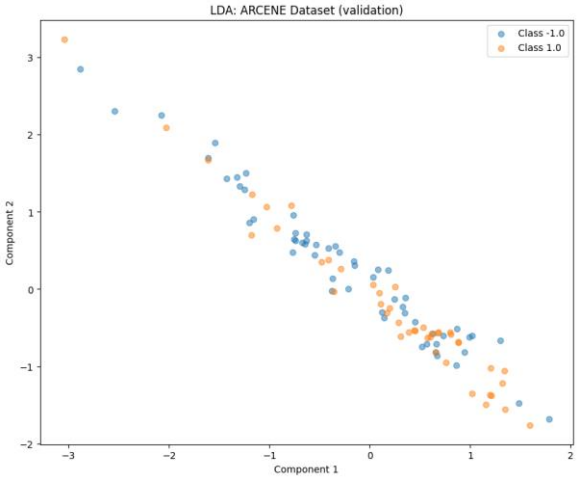
일반적인 LDA에 Arcene 데이터셋을 학습시킨 결과는 다음과 같다. 왼쪽이 일반 LDA 결과, 오른쪽이 QKULDA 결과이다.

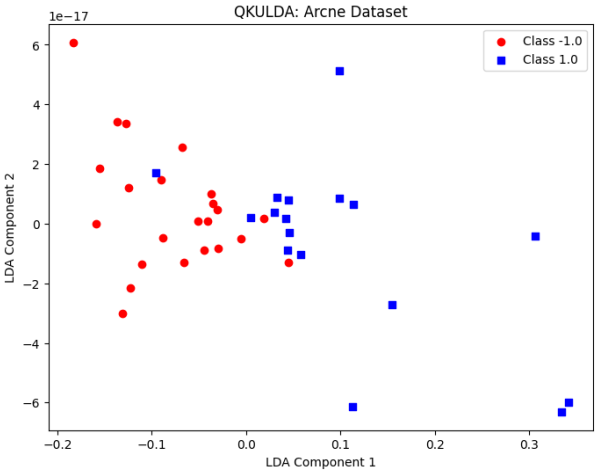
```
# 모델 학습
lda = LDA()
lda.fit(train_data, train_labels)
```

```
[15] # 성능 평가
valid_predictions = lda.predict(valid_data)

accuracy = accuracy_score(valid_labels, valid_predictions)
conf_matrix = confusion_matrix(valid_labels, valid_predictions)
class_report = classification_report(valid_labels, valid_predictions)

print(f'Accuracy: {accuracy}')
print(f'Confusion Matrix:₩n{conf_matrix}')
print(f'Classification Report:₩n{class_report}')
```

```
Accuracy: 0.47
Confusion Matrix:
[[29 27]
 [26 18]]
Classification Report:
              precision    recall  f1-score   support

        -1.0       0.53      0.52      0.52        56
         1.0       0.40      0.41      0.40        44

    accuracy                           0.47       100
   macro avg       0.46      0.46      0.46       100
weighted avg       0.47      0.47      0.47       100
```

```
Best λ1: 0.1, Best λ2: 0.1, Best Accuracy: 0.925
Mean Accuracy: 0.925
Confusion Matrices:
[[21  1]
 [ 3 15]]
[[20  2]
 [ 1 17]]
[[19  3]
 [ 0 18]]
[[22  1]
 [ 0 17]]
[[21  2]
 [ 2 15]]
Classification Reports:
              precision    recall  f1-score   support

        -1.0       0.91      0.91      0.91        23
         1.0       0.88      0.88      0.88        17

    accuracy                           0.90        40
   macro avg       0.90      0.90      0.90        40
weighted avg       0.90      0.90      0.90        40
```





일반 LDA의 정확도는 0.47로, 그래프를 보았을때에도 분류가 잘 되고 있지 않다는 것을 알 수 있다. 또한 기존 LDA로 학습한 시간이 30분 이상이 소요되었다. 반면, QKULDA는 최대 5분의 소요시간이 걸렸다. 또한 QKULDA 정확도: 0.925. 일반 LDA 정확도: 0.47를 통해 QKULDA가 시간복잡도와 정확도 등의 측면에서 더 좋은 성능을 나타내고, 데이터를 훨씬 잘 분류하고 있는 것을 알 수 있다.

4. 결론

QKULDA(Q-Kernel Optimized UMAP Linear Discriminant Analysis)

Final LDA Vs QKULDA Comparison

	Feature	LDA	QKULDA
1	Method	Linear Discriminant Analysis	Q-Kernel Optimized UMAP Linear Discriminant Analysis
2	Accuracy	0.47	0.925
3	Computation Time	30+ minutes	Up to 5 minutes
4	Additional Dimensionality Reduction	No	Yes (UMAP)
5	Additional Kernel Optimization	No	Yes (Q optimization)
6	Additional Handling Non-linearity	No	Yes (Kernel LDA)

본 과제에서는 고차원 데이터의 효과적인 분류를 위해 새로운 방법론인 QKULDA(Q-Kernel Optimized UMAP Linear Discriminant Analysis)를 제안했다. Arcene 데이터셋을 사용하여 기존 LDA와 QKULDA를 비교한 결과, QKULDA가 기존 LDA에 비해 우수한 성능을 보였다.

1) 성능 비교

- 정확도: QKULDA는 0.925의 정확도를 보였으나, 기존 LDA는 0.47에 그쳤다.

- 시간 효율성: QKULDA의 학습 시간은 최대 5분으로, 기존 LDA의 30분 이상 소요된 시간에 비해 크게 단축되었다.

2) 주요 개선 사항

- 커널 도입: LDA의 선형성 한계를 극복하여 비선형적으로 분포된 데이터를 효과적으로 분류할 수 있다.

- Q 최적화: 커널 함수의 파라미터 최적화를 통해 등분산성 가정을 완화하고, 클래스 분리를 극대화하여 다양한 분포를 가지는 클래스도 효과적으로 분류할 수 있다.

- UMAP 도입: 고차원 데이터를 저차원으로 축소하여 차원의 저주 문제를 해결하고, 계산 복잡도를 줄임으로써 모델의 성능을 향상시켰다.

결론적으로, QKULDA는 기존 LDA의 한계를 보완하며 고차원 데이터의 분류 성능을 크게 향상시킬 수 있는 강력한 도구임을 입증하였다. 이는 다양한 실제 데이터셋에 확장 가능하며, 고차원 데이터 분석의 새로운 패러다임을 제시할 수 있을 것으로 기대된다.

5. reference

Ghojogh, B., Karray, F., & Crowley, M. (2022). Fisher and Kernel Fisher Discriminant Analysis: Tutorial.

McInnes, L., Healy, J., & Melville, J. (2020). UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction.

Yang, J., Jin, Z., Yang, J.-y., Zhang, D., & Frangi, A. F. (2004). Essence of kernel Fisher discriminant: KPCA plus LDA.

Arcene dataset - https://archive.ics.uci.edu/dataset/167/arcene