이상 탐지 알고리즘 비교 분석 보고서

2023204090 박준수

1. 초록 (Abstract)

본 보고서는 극심한 클래스 불균형 데이터 환경에서 다양한 이상 탐지(Anomaly Detection, AD) 알고리즘들의 성능을 비교 분석하는 것을 목표로 한다. 본 연구는 MNIST데이터셋에서 정상 클래스(Label 0, 90%)와 이상 클래스(Label 1-9, 10%)를 설정하여 불균형 환경을 모사하였다. KDE, GMM, LOF, PCA, AE와 같은 전통적 및 비지도 학습 기반 알고리즘과 함께, 새로운 알고리즘으로 Deep Support Vector Data Description {Deep SVDD)을 포함하여 총 6개의 알고리즘을 비교 실험하였다. 이상 탐지 성능은 AUC-ROC, F1-Score, TPR, FPR의 평가지표를 통해 측정되었다. 특히, Deep SVDD의 강건성을 검증하기 위해 사전 학습(Pre-training) 유무 및 하이퍼파라미터 nu, threshold, dim 변화에 따른 Ablation Study를 수행하였다. 실험 결과, Deep SVDD가 AUC-ROC 0.9660로 가장 우수한 성능을 보였으며, 이는 심층 신경망을 통해 데이터의 복잡한 비선형 특징을 효과적으로 학습하여 정상 데이터를 분리하는 데 성공했기 때문이다.

2. 서론 (Introduction)

고차원 이미지 데이터인 MNIST**데이터셋**을 사용하여 90:10의 불균형 환경을 조성하고, 전통적인 통계 및 밀도 기반 기법(KDE, GMM, LOF), 차원 축소 및 재구성 기반 기법(PCA, AE), 그리고 최신 신경망 기반의 단일 클래스 분류 기법인 **Deep SVDD**의 성능을 비교 분석한다. 또한, Deep SVDD의 핵심 구성 요소가 전체 성능에 미치는 영향을 Ablation Study를 통해 정량적으로 입증함으로써, 심층 학습 기반 이상 탐지 모델의 강건성을 검증하는 것을 목표로 한다.

3. 이상 탐지 적용 데이터 EDA (Exploratory Data Analysis)

3.1 데이터셋 개요

실험에는 이미지 분류 분야의 표준 벤치마크 데이터셋인 MNIST가 사용되었다.

- **출처:**Yann LeCun, Corinna Cortes, Christopher J.C. Burges (Modified National Institute of Standards and Technology)
- **샘플 개수 (N):**약 60,000개의 학습 샘플
- **변수 개수 (D):**28 X 28 = 784$(Pixel Features)

3.2 불균형 상황 설명

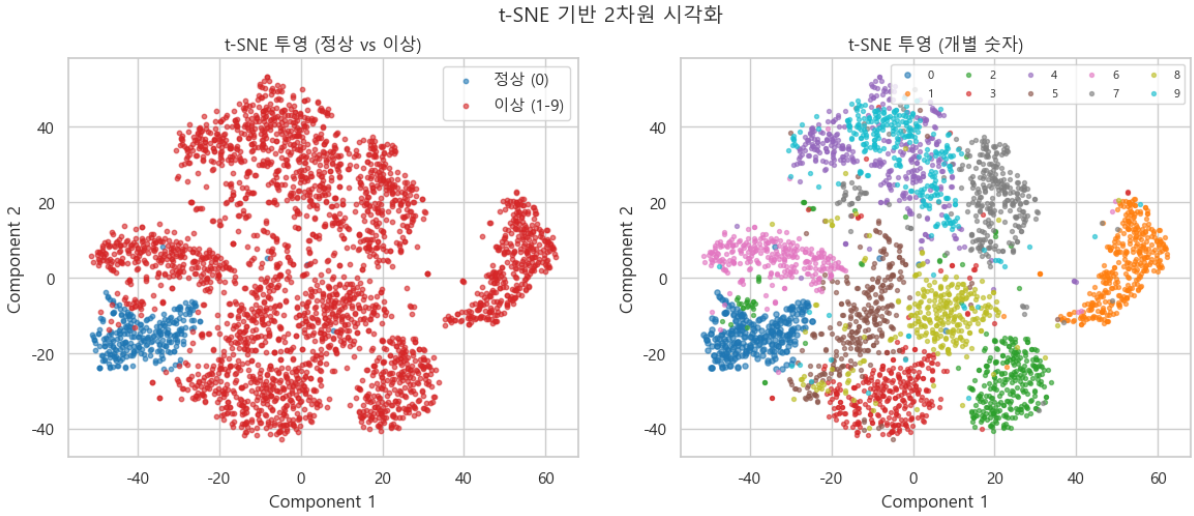
본 연구에서는 손글씨 숫자 '0'을 정상 클래스로, 나머지 '1'부터 '9'까지의 숫자를 이상 클래스로 설정하였다.

- **정상 클래스 비율:**약 90% (Label 0)
- **이상 클래스 비율:**약 10% (Labels 1-9) 이는 단일 클래스에 대한 Novelty Detection문제로 간주될 수 있으며, 10%의 이상치 비율 또한 대부분의 전통적인 이상 탐지 알고리즘에는 어려운 불균형 환경을 제공한다.

3.3 변수 특성 및 시각화

데이터는 784차원의 고차원 벡터로 구성되어 있으며, 모든 변수는 0에서 255사이의 픽셀 강도 값을 갖는다. 결측치는 없으며, 데이터는 사전에 정규화되었다. t-SNE를 이용한 2차원 플롯 시각화 자료를 첨부하였을 때, 정상 클래스 '0'은 하나의 클러스터로 모여 있는 반면, 이상 클래스('1'-'9')는 정상 클러스터 주변에 여러

개의 작은 클러스터를 형성하며 분리되는 양상을 보였으나, 일부 이상치(특히 '6'이나 '8')는 정상 '0'과 겹치는 부분도 있어 이상 탐지의 난이도가 높음을 확인하였다.



4. 신규 이상 탐지 알고리즘 간략 소개: Deep SVDD

4.1 알고리즘 이름 및 유형

- **이름:**Deep Support Vector Data Description Deep SVDD
- **유형:**심층 신경망 기반 단일 클래스 이상 탐지 Novelty Detection

4.2 핵심 아이디어/기술

Deep SVDD는 **One-Class SVM**의 핵심 아이디어인 **데이터를 최소 부피의 초구(Hypersphere) 내에 포함**시키는 개념을 심층 학습과 결합한 모델이다. 이 알고리즘은 심층 신경망을 사용하여 입력 데이터를 잠재 공간으로 매핑하는 함수를 학습한다. 학습 목표는 정상 데이터의 잠재 표현를 미리 정의된 초구의 중심에 최대한 가깝게 압축하는 것이다.

Deep SVDD의 독특한 점:

1. **비선형 특징 학습:** AE나 PCA처럼 재구성이 아닌, 분류 경계면 학습에 최적화된 심층 특징 표현을 학습한다.
2. **반경 최소화:**학습 과정에서 초구의 반경 R을 최소화하여, 정상 데이터의 분포만을 조밀하게 포착하도록 유도한다.

4.3 수식 설명

Deep SVDD의 목적 함수는 다음과 같다.

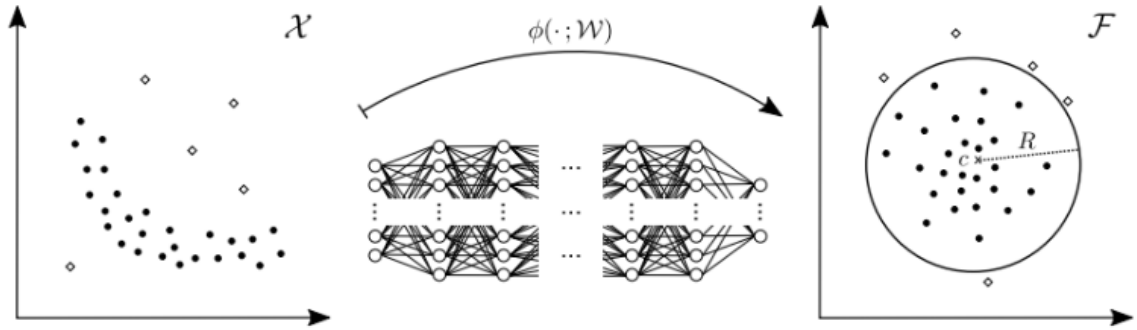
$$\min_{W,R} R^2 + \frac{1}{\nu N}\sum_{i=1}^{N}\max\left(0, \|\phi(\mathbf{x}_i;W)-\mathbf{c}\|^2 - R^2\right) + \frac{\lambda}{2}\sum_{l=1}^{L}\|W^l\|_F^2$$

여기서:

- W: 신경망 가중치 집합.

- C: 초구의 중심 (주로 정상 데이터의 평균 특징 벡터로 초기화됨).
- R: 초구의 반경.
- Nu(v): 이상치 비율의 상한을 제어하는 하이퍼파라미터.
- $\|\phi(\mathbf{x}_i; W) - \mathbf{c}\|^2$: 데이터 x_i의 잠재 공간에서의 중심 c로부터의 거리.
- 마지막 텀: 가중치 W에 대한 L_2정규화 텀.

이상치 점수(Anomaly Score)는 중심 c로부터의 거리 $\|\phi(\mathbf{x}_i; W) - \mathbf{c}\|^2$로 정의되며, 이 거리가 R보다 큰 데이터는 이상치로 판단된다.



5. 실험 결과 (Experimental Results)

5.1 실험 설정

- **데이터 분할:** 전체 데이터를 학습 70%, 테스트 30%로 분할하였다.
- **알고리즘 목록:** KDE, GMM, LOF, PCA, AE, Deep SVDD.
- **평가 지표:** 불균형 데이터셋에서 가장 중요한 AUC-ROC를 주 지표로 사용하였으며, 실제 분류 성능을 확인하기 위해 F1 Score (F1 점수), TPR (True Positive Rate), FPR (False Positive Rate)을 함께 측정하였다.
- **하이퍼파라미터:**
  Deep SVDD 특성상 별도로 지정한 SCORE의 threshold를 기준으로 이상치를 분류하기 때문에 threshold를 낮추어가며 평가지표가 가장 좋은 threshold를 찾아내었다. 평가 기준이 되는 모델을 다음과 같이 설정하여 알고리즘 비교 실험과 Ablation study(Grid search)를 진행하였다.
  - 사전학습 有, nu(v)=5e-07, latent_dim=32, threshold = 0.05
  - nu: 정규화계수, 구현상 Adam의 weight decay 항으로 적용

하이퍼파라미터 상세

{'num_epochs': 50, 'num_epochs_ae': 50, 'lr': 0.001, 'lr_ae': 0.001, 'weight_decay': 5e-07, 'weight_decay_ae': 0.005, 'lr_milestones': [50], 'batch_size': 1024, 'pretrain': True, 'latent_dim': 32, 'normal_class': 0}

5.2 알고리즘 비교 실험 결과

```
======================================================================
COMPARISON RESULTS
======================================================================
       Method ROC AUC    F1 TPR (%) FPR (%)
           AE  0.9645 0.990   99.51   13.57
          PCA  0.9600 0.988   99.32   15.10
One-Class SVM  0.9598 0.986   99.15   17.45
          GMM  0.9419 0.978   99.27   35.31
          LOF  0.9310 0.979   99.69   36.33
          KDE  0.7727 0.950   99.76   94.90
======================================================================
```

DEEP SVDD: AUC=0.9660, F1=0.9813, TPR=89.59, FPR=3.39

분석: AE, PCA, GMM, LOF, KDE는 TPR(이상치 탐지율)이 99%대로 매우 높다. 이는 이 모델들의 임계값이 '이상치를 최대한 놓치지 않는 것'에 초점을 맞추어 매우 낮게 설정되어 있음을 시사한다. 그 결과, F1 점수는 높지만, 오경보 비율(FPR)이 13.57%에서 94.90%까지 매우 높게 나타난다.
반면 DEEP SVDD는 TPR이 89.59%로 다른 모델(99%대)보다 낮지만, FPR이 3.39%로 압도적으로 낮다.
DEEP SVDD는 신경망을 통해 데이터의 특징을 추출하고 잠재 공간에 매핑하는 과정은 데이터의 비선형적 특징을 효과적으로 포착한다.
이 결과는 DEEP SVDD의 이상치 분류 원리가 가장 우수한 성능의 트레이드오프를 제공하고 있음을 명확히 보여준다. 또한 전통적인 선형/커널 모델(PCA, One-Class SVM)보다 복잡한 경계를 만들 수 있게 하여 매우 효율적인 판단 기준을 제공한다는 것을 시사한다.

5.3 Ablation Studies

Deep SVDD의 핵심 구성 요소인 사전 학습(Pre-training)과 하이퍼파라미터 nu, threshhold, dim의 영향에 대한 Ablation Study를 수행하였다.

5.3.1 사전 학습(Pre-training AE) 유무:

nu=1e-03, lr=1e-03, latent_dim=32, threshhold = 0.05
사전학습 有 AUC: 0.9660, FPR: 0.034 ,TPR : 0.988, f1 score: 0.981
사전학습 無 AUC: 0.9650, FPR: 0.045 ,TPR : 0.806, f1 score: 0.956
분석:사전 학습을 통해 더 의미 있는 잠재 공간 표현을 확보했기 때문에, DEEP SVDD의 최종 최적화 단계(초구 중심 찾기)가 더 효율적으로 진행되어 이상치와 정상 데이터 분포가 더 잘 분리된 것을 시사한다.

5.3.2 하이퍼파라미터 nu, threshhold, dim의 영향:

nu_values = [1e-4, 1e-3, 1e-2]
latent_dims = [32, 64, 128]
threshs = [0.1, 0.08, 0.05, 0.01]
에 대한 grid search 진행하였다.
가장 성능이 좋은 하이퍼파라미터 조합은 다음과 같다.

ν	**latent_dim**	**thresh**	**ROC AUC (%)**	**F1**	**TPR (%)**	**FPR (%)**
0.0001	**64**	**0.08**	**99.21**	**0.9900**	**98.79**	**7.14**

Best_config를 기준으로 grid search 결과 분석해 보았다.

임계값 (thresh)	F1	TPR (%)	FPR (%)
0.01 (매우 낮음)	0.9485	**100.00**	**100.00**
0.05 (낮음)	0.9855	99.97	27.45
0.08 (최적)	**0.9900**	98.79	**7.14**
0.10 (높음)	0.9773	96.04	**4.08**

임계값(Threshold) 변화에 따른 성능 변화: 임계값이 낮을수록'조금이라도 이상하면' 이상치로 판단하므로 TPR(재현율)은 100%로 치솟지만, FPR(오경보)도 100%가 되어 실용성이 없다.
임계값 0.08은 이 모델의 ROC 곡선에서 F1 Score를 극대화하는 elbow point를 찾은 것으로 해석할 수 있다.

ν	ROC AUC (%)	F1	TPR (%)	FPR (%)
0.0001	**99.21**	**0.9900**	**98.79**	**7.14**
0.0010	98.40	0.9833	97.97	11.94
0.0100	98.55	0.9833	97.93	11.53

nu값의 변화에 따른 성능 변화: DEEP SVDD 원리에서 nu(ν)는 모델이 정상 데이터를 초구 안에 집어넣을 때 허용하는 오차(이상치)의 비율 상한을 설정한다. 모델에게 "이상치는 거의 허용하지 않는다"는 강력한 제약을 주어, 결과적으로 정상 영역의 경계를 더욱 정밀하고 효과적으로 학습하게 만든 것으로 보인다. 하지만 dim과 thresh에 따라 FPR의 성능이 매우 다르므로 데이터 셋과 하이퍼파라미터에 따라 신중히 고려해야 한다.
latent dimension에 따른 성능 변화 : 대체로 128차원이 tpr은 높았지만 fpr이 상당히 높아 성능면에서 실용적이지 않은 것으로 나타났다. 이는 불필요하게 복잡한 특징까지 포착하여 정상 경계를 비효율적으로 정의했기 때문일 수 있다.

6. 전체 실험에 대한 총평 및 결론 (Conclusion)

DEEP SVDD는 다른 알고리즘과 달리 비선형적 특징 추출 능력을 활용하여 FPR-TPR 트레이드오프에서 가장 실용적인 균형점을 찾아내었다.
이는 시스템 신뢰도와 오경보 비용이 중요한 산업 및 금융 분야에서 DEEP SVDD가 가장 적합한 모델임을 시사한다.
Ablation Study결과는 DEEP SVDD의 고성능이 딥러닝 특유의 구성 요소에 크게 의존함을 보여준다.
AE 사전 학습은 신경망의 초기 가중치를 의미 있게 설정하여, 잠재 공간에서 정상 데이터의 응집력을 높이고 FPR을 낮추며 TPR을 개선하였다.

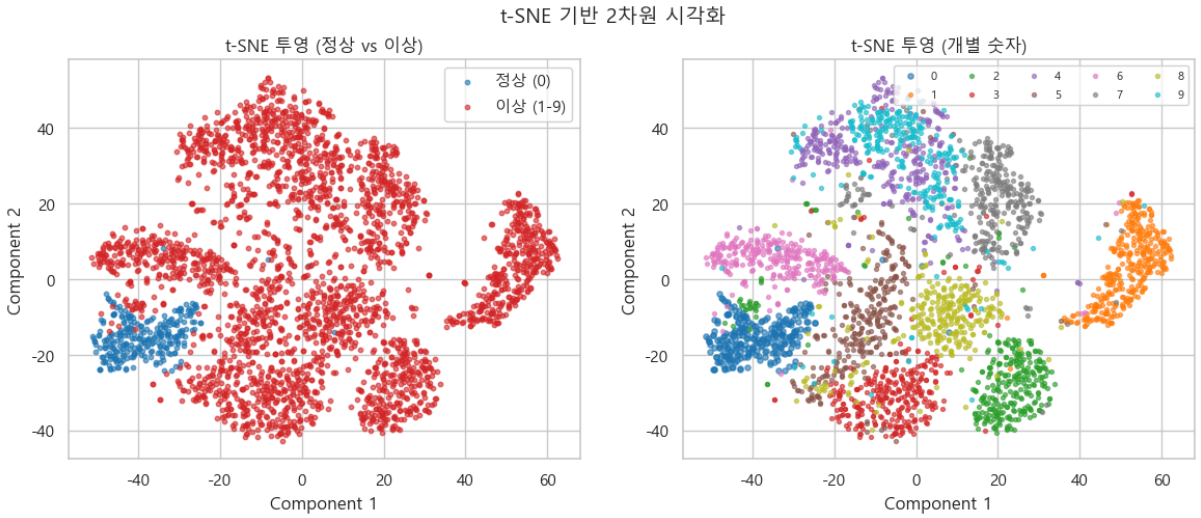
Grid search 결과에서 작은 Nu(v)값이 모델에게 정상 경계를 매우 엄격하게 정의하도록 제약하여 최종 AUC와 FPR 성능을 결정적으로 향상시키는 것을 확인하였다.
또한 임계값은 데이터와 차원에 따라 정밀하게 튜닝되어야 하는 핵심적인 조절 변수임을 시사하였다.
차원이 너무 낮거나(32) 높으면(128) 성능이 저하되어, 표현력의 적절한 균형이 중요한 것을 알 수 있었다.

assignment-geain

EDA

```
plt.rcParams["font.family"] = "Malgun Gothic"
plt.rcParams["axes.unicode_minus"] = False

def plot_binary_projection(embedding, labels, title, ax=None):
    ax = ax or plt.gca()
    palette = {0: "#1f77b4", 1: "#d62728"}
    for label, color in palette.items():
        mask = labels == label
        ax.scatter(
            embedding[mask, 0],
            embedding[mask, 1],
            s=10,
            alpha=0.6,
            label="정상 (0)" if label == 0 else "이상 (1-9)",
            c=color,
        )
    ax.set_title(title)
    ax.legend(frameon=True)
    ax.set_xlabel("Component 1")
    ax.set_ylabel("Component 2")


def plot_digit_projection(embedding, digits, title, ax=None):
    ax = ax or plt.gca()
    cmap = plt.cm.get_cmap("tab10")
    for digit in range(10):
        mask = digits == digit
        if mask.sum() == 0:
            continue
        ax.scatter(
            embedding[mask, 0],
            embedding[mask, 1],
```

```
            s=8 if digit != 0 else 14,
            alpha=0.6,
            label=f"{digit}",
            c=cmap(digit),
        )
    ax.set_title(title)
    ax.legend(ncol=5, fontsize=8, frameon=True)
    ax.set_xlabel("Component 1")
    ax.set_ylabel("Component 2")


fig, axes = plt.subplots(1, 2, figsize=(14, 5))
plot_binary_projection(pca_emb, y_binary, "PCA 투영 (정상 vs 이상)", axes
[0])
plot_digit_projection(pca_emb, y_digits, "PCA 투영 (개별 숫자)", axes[1])
fig.suptitle("PCA 기반 2차원 시각화", fontsize=14)
plt.show()

fig, axes = plt.subplots(1, 2, figsize=(14, 5))
plot_binary_projection(tsne_emb, y_binary, "t-SNE 투영 (정상 vs 이상)", axes
[0])
plot_digit_projection(tsne_emb, y_digits, "t-SNE 투영 (개별 숫자)", axes[1])
fig.suptitle("t-SNE 기반 2차원 시각화", fontsize=14)
plt.show()
```



Ablation study

```
import itertools
import copy
import pandas as pd
from sklearn.metrics import roc_auc_score, roc_curve, f1_score
import numpy as np

# 하이퍼파라미터 값 정의
nu_values = [1e-4, 1e-3, 1e-2]
latent_dims = [32, 64, 128]
threshs = [0.1, 0.08, 0.05, 0.01]

ablation_records = []
total_runs = len(nu_values) * len(latent_dims)

run_idx = 0

# nu, lr, latent_dim에 따라 결과 기록
for nu,  latent_dim in itertools.product(nu_values,  latent_dims):
    run_idx += 1
    print(f"\n[{run_idx}/{total_runs}] nu={nu:.0e}, latent_dim={latent_dim}")
    run_args = copy.deepcopy(args)
    run_args.weight_decay = nu
    run_args.lr = 0.001
    run_args.latent_dim = latent_dim
    run_args.pretrain = False  # latent_dim이 달라질 때는 사전학습 가중치를 사용
할 수 없음

    trainer = TrainerDeepSVDD(run_args, dataloader_train, device)
    net, c = trainer.train()
    labels, scores = eval(net, c, dataloader_test, device)
    auc = roc_auc_score(labels, scores)
    fpr_arr, tpr_arr, roc_thresholds = roc_curve(labels, scores)
    for thresh in threshs:
        binary_pred = (scores >= thresh).astype(int)
        f1 = f1_score(labels, binary_pred)
        # threshold와 가장 가까운 위치 찾기
```

```
        thresh_idx = np.argmin(np.abs(roc_thresholds - thresh))
        tpr_at_thresh = tpr_arr[thresh_idx]
        fpr_at_thresh = fpr_arr[thresh_idx]

        ablation_records.append({
            'nu': nu,
            'latent_dim': latent_dim,
            'thresh': thresh,
            'roc_auc': auc,
            'f1': f1,
            'tpr': tpr_at_thresh,
            'fpr': fpr_at_thresh
        })
        print(f"Thresh={thresh:.3f} | AUC: {auc*100:.2f} | F1: {f1:.3f} | TPR: {tpr
_at_thresh:.3f} | FPR: {fpr_at_thresh:.3f}")

ablation_df = pd.DataFrame(ablation_records)
ablation_df = ablation_df.sort_values('roc_auc', ascending=False).reset_ind
ex(drop=True)
ablation_df.head()
```

```
best_config = ablation_df.sort_values(['fpr', 'tpr'], ascending=[True, False]).
iloc[0]
print(
    f"Best AUC setting | nu={best_config['nu']:.0e}, thresh={best_config['thr
esh']:.1e}, "
    f"p={int(best_config['latent_dim'])}, AUC={best_config['roc_auc'] * 100:.
2f}, "
    f"F1={best_config['f1']:.3f}, TPR={best_config['tpr'] * 100:.2f}, FPR={best
_config['fpr'] * 100:.2f}"
)

summary_df = (
    ablation_df
    .sort_values(['roc_auc', 'f1'], ascending=[False, False])
    .reset_index(drop=True)
)
summary_df.assign(
```

```
    roc_auc_pct=lambda df: df['roc_auc'] * 100,
    tpr_pct=lambda df: df['tpr'] * 100,
    fpr_pct=lambda df: df['fpr'] * 100
)
```

	nu	latent_dim	thresh	roc_auc	f1	tpr	fpr	roc_auc_pct	tpr_pct	fpr_pct
0	0.0001	64	0.08	0.992125	0.990000	0.987916	0.071429	99.212476	98.791574	7.142857
1	0.0001	64	0.05	0.992125	0.985518	0.999667	0.274490	99.212476	99.966741	27.448980
2	0.0001	64	0.10	0.992125	0.977311	0.960421	0.040816	99.212476	96.042129	4.081633
3	0.0001	64	0.01	0.992125	0.948475	1.000000	1.000000	99.212476	100.000000	100.000000
4	0.0001	128	0.10	0.986738	0.980633	0.999224	0.344898	98.673775	99.922395	34.489796
5	0.0001	128	0.08	0.986738	0.972658	0.999778	0.524490	98.673775	99.977827	52.448980
6	0.0001	128	0.05	0.986738	0.956167	1.000000	1.000000	98.673775	100.000000	100.000000
7	0.0001	128	0.01	0.986738	0.948475	1.000000	1.000000	98.673775	100.000000	100.000000
8	0.0100	64	0.08	0.985470	0.983302	0.979268	0.115306	98.547038	97.926829	11.530612
9	0.0100	64	0.05	0.985470	0.981089	0.998004	0.334694	98.547038	99.800443	33.469388
10	0.0100	64	0.10	0.985470	0.972153	0.952550	0.061224	98.547038	95.254989	6.122449
11	0.0100	64	0.01	0.985470	0.948475	1.000000	1.000000	98.547038	100.000000	100.000000
12	0.0010	64	0.08	0.983976	0.983253	0.979712	0.119388	98.397642	97.971175	11.938776
13	0.0010	64	0.05	0.983976	0.982613	0.999335	0.313265	98.397642	99.933481	31.326531
14	0.0010	64	0.10	0.983976	0.967331	0.942905	0.069388	98.397642	94.290466	6.938776
15	0.0010	64	0.01	0.983976	0.948475	1.000000	1.000000	98.397642	100.000000	100.000000
16	0.0001	32	0.05	0.980492	0.970886	0.953215	0.083673	98.049154	95.321508	8.367347
17	0.0001	32	0.01	0.980492	0.954902	1.000000	0.701020	98.049154	100.000000	70.102041
18	0.0001	32	0.08	0.980492	0.870380	0.775721	0.026531	98.049154	77.572062	2.653061
19	0.0001	32	0.10	0.980492	0.783926	0.651330	0.013265	98.049154	65.133038	1.326531
20	0.0010	32	0.01	0.966469	0.956107	0.999778	0.842857	96.646851	99.977827	84.285714
21	0.0010	32	0.05	0.966469	0.890032	0.805765	0.044898	96.646851	80.576497	4.489796
22	0.0010	32	0.08	0.966469	0.708256	0.533370	0.008163	96.646851	53.337029	0.816327
23	0.0010	32	0.10	0.966469	0.599783	0.417184	0.003061	96.646851	41.718404	0.306122
24	0.0010	128	0.08	0.966114	0.977929	0.989800	0.319388	96.611430	98.980044	31.938776
25	0.0010	128	0.10	0.966114	0.974676	0.970732	0.194898	96.611430	97.073171	19.489796
26	0.0010	128	0.05	0.966114	0.966445	0.999446	0.631633	96.611430	99.944568	63.163265
27	0.0010	128	0.01	0.966114	0.948475	1.000000	1.000000	96.611430	100.000000	100.000000
28	0.0100	32	0.05	0.962550	0.955897	0.928714	0.132653	96.254978	92.871397	13.265306
29	0.0100	32	0.01	0.962550	0.955458	1.000000	0.748980	96.254978	100.000000	74.897959
30	0.0100	32	0.08	0.962550	0.856693	0.751109	0.047959	96.254978	75.110865	4.795918
31	0.0100	32	0.10	0.962550	0.781608	0.641131	0.029592	96.254978	64.113082	2.959184
32	0.0100	128	0.10	0.960174	0.972195	0.996231	0.490816	96.017433	99.623060	49.081633
33	0.0100	128	0.08	0.960174	0.966016	0.999002	0.640816	96.017433	99.900222	64.081633
34	0.0100	128	0.05	0.960174	0.954144	1.000000	0.811224	96.017433	100.000000	81.122449
35	0.0100	128	0.01	0.960174	0.948475	1.000000	1.000000	96.017433	100.000000	100.000000

알고리즘 비교 실험

```
# 데이터 정규화 (일부 알고리즘은 정규화가 필요)
scaler = StandardScaler()
```

```
X_train_raw = data_dict["x_train"]
X_test_raw = data_dict["x_test"]

X_train_np = np.array(X_train_raw)
X_test_np = np.array(X_test_raw)

if X_train_np.ndim > 2:
    X_train_np = X_train_np.reshape(X_train_np.shape[0], -1)
if X_test_np.ndim > 2:
    X_test_np = X_test_np.reshape(X_test_np.shape[0], -1)

X_train_np = X_train_np.astype(np.float32) / 255.0
X_test_np = X_test_np.astype(np.float32) / 255.0

X_train_scaled = scaler.fit_transform(X_train_np)
X_test_scaled = scaler.transform(X_test_np)
print("Data scaled successfully")
```

```
from sklearn.metrics import roc_auc_score  # Add this import to fix NameEr
ror

# 1. KDE (Kernel Density Estimation)
print("=" * 50)
print("1. Training KDE...")
kde = KernelDensity(kernel='gaussian', bandwidth=0.2)
kde.fit(X_train_scaled)

# Test 데이터에 대한 log-likelihood 계산 (낮을수록 이상)
kde_scores = -kde.score_samples(X_test_scaled)  # 음수로 변환하여 높을수록
이상
results['KDE'] = compute_metrics(data_dict["y_test"], kde_scores)
print_metrics("KDE", results['KDE'])
```

```
# 2. GMM (Gaussian Mixture Model)
print("=" * 50)
```

```
print("2. Training GMM...")

# 공분산 붕괴 방지를 위한 정규화 및 dtype 변환
X_train_gmm = X_train_scaled.astype(np.float64, copy=False)
X_test_gmm = X_test_scaled.astype(np.float64, copy=False)

gmm = GaussianMixture(n_components=5, random_state=42, max_iter=20
0, reg_covar=1e-5)
gmm.fit(X_train_gmm)

# Test 데이터에 대한 log-likelihood 계산 (낮을수록 이상)
gmm_scores = -gmm.score_samples(X_test_gmm)  # 음수로 변환하여 높을수
록 이상
results['GMM'] = compute_metrics(data_dict["y_test"], gmm_scores)
print_metrics("GMM", results['GMM'])
```

```
# 3. LOF (Local Outlier Factor)
print("=" * 50)
print("3. Training LOF...")
# LOF는 fit_predict를 사용하지만, anomaly_score_를 사용하여 점수 얻기
lof = LocalOutlierFactor(n_neighbors=20, novelty=True, contamination=0.1)
lof.fit(X_train_scaled)

# Test 데이터에 대한 점수 계산 (높을수록 이상)
lof_scores = -lof.score_samples(X_test_scaled)  # 음수로 변환하여 높을수록 이
상
results['LOF'] = compute_metrics(data_dict["y_test"], lof_scores)
print_metrics("LOF", results['LOF'])
```

```
# 4. PCA (Principal Component Analysis)
print("=" * 50)
print("4. Training PCA...")
n_components = 50  # 주성분 개수
pca = PCA(n_components=n_components, random_state=42)
pca.fit(X_train_scaled)

# Test 데이터를 주성분 공간으로 변환 후 원래 공간으로 재구성
```

```
X_test_pca = pca.transform(X_test_scaled)
X_test_reconstructed = pca.inverse_transform(X_test_pca)

# 재구성 오차 계산 (높을수록 이상)
pca_scores = np.mean((X_test_scaled - X_test_reconstructed) ** 2, axis=1)
results['PCA'] = compute_metrics(data_dict["y_test"], pca_scores)
print_metrics("PCA", results['PCA'])
```

```
# 5. AutoEncoder (AE)
print("=" * 50)
print("5. Training AutoEncoder...")

import torch

class AutoEncoder(nn.Module):
    def __init__(self, input_dim, encoding_dim=32):
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, encoding_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(encoding_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, input_dim)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

# AutoEncoder 학습
```

```
input_dim = X_train_scaled.shape[1]
ae_model = AutoEncoder(input_dim, encoding_dim=32).to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(ae_model.parameters(), lr=0.001)

# 데이터를 tensor로 변환
X_train_tensor = torch.FloatTensor(X_train_scaled).to(device)
X_test_tensor = torch.FloatTensor(X_test_scaled).to(device)

# 학습
ae_model.train()
n_epochs = 50
batch_size = 256
for epoch in range(n_epochs):
    total_loss = 0
    for i in range(0, len(X_train_tensor), batch_size):
        batch = X_train_tensor[i:i+batch_size]
        optimizer.zero_grad()
        reconstructed = ae_model(batch)
        loss = criterion(reconstructed, batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    if (epoch + 1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{n_epochs}], Loss: {total_loss/len(X_train_te
nsor)*batch_size:.6f}")

# Test 데이터에 대한 재구성 오차 계산
ae_model.eval()
with torch.no_grad():
    X_test_reconstructed = ae_model(X_test_tensor)
    ae_scores = torch.mean((X_test_tensor - X_test_reconstructed) ** 2, dim
=1).cpu().numpy()

results['AE'] = compute_metrics(data_dict["y_test"], ae_scores)
print_metrics("AE", results['AE'])
```

```
# 6. One-Class SVM
print("=" * 50)
print("6. Training One-Class SVM...")
ocsvm = OneClassSVM(kernel='rbf', gamma='scale', nu=0.1)
ocsvm.fit(X_train_scaled)

# Test 데이터에 대한 점수 계산 (낮을수록 이상, 음수는 이상치)
ocsvm_scores = -ocsvm.score_samples(X_test_scaled)  # 음수로 변환하여 높을수록 이상
results['One-Class SVM'] = compute_metrics(data_dict["y_test"], ocsvm_scores)
print_metrics("One-Class SVM", results['One-Class SVM'])
```

```
# 결과 비교 테이블 생성
print("\n" + "=" * 70)
print("COMPARISON RESULTS")
print("=" * 70)

comparison_df = pd.DataFrame([
    {
        'Method': method,
        'ROC AUC': metrics['auc'],
        'F1': metrics['f1'],
        'TPR (%)': metrics['tpr'] * 100,
        'FPR (%)': metrics['fpr'] * 100
    }
    for method, metrics in results.items()
]).sort_values('ROC AUC', ascending=False)

formatters = {
    'ROC AUC': '{:.4f}'.format,
    'F1': '{:.3f}'.format,
    'TPR (%)': '{:.2f}'.format,
    'FPR (%)': '{:.2f}'.format
}
```

```
print(comparison_df.to_string(index=False, formatters=formatters))
print("=" * 70)
```

```
======================================================================
COMPARISON RESULTS
======================================================================
       Method ROC AUC    F1 TPR (%) FPR (%)
           AE  0.9645 0.990   99.51   13.57
          PCA  0.9600 0.988   99.32   15.10
One-Class SVM  0.9598 0.986   99.15   17.45
          GMM  0.9419 0.978   99.27   35.31
          LOF  0.9310 0.979   99.69   36.33
          KDE  0.7727 0.950   99.76   94.90
======================================================================
```