고등지능정보공학2 중간고사

지능형데이터·최적화학과
2024321646 김현진

1 - (1)

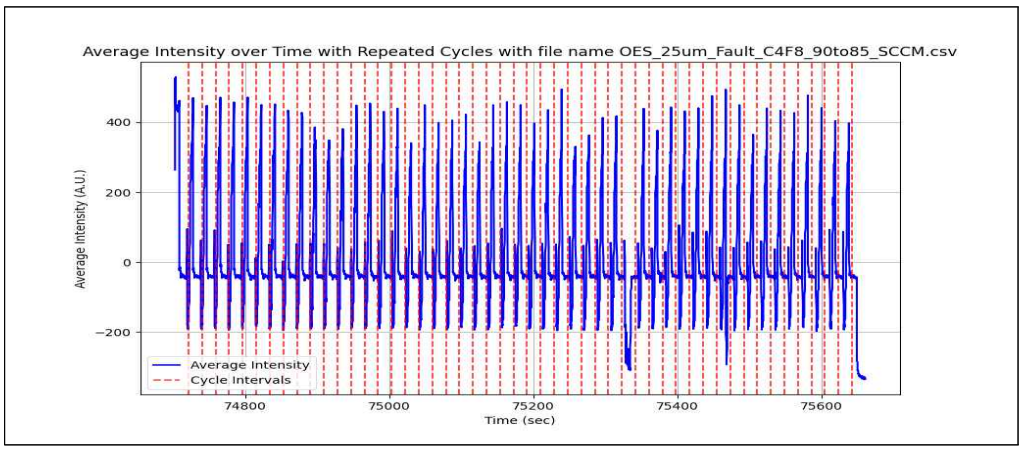
FDC 데이터와 OES 데이터는 각각 3차원 데이터 셋(샘플 수×변수×가공 시간)이며, 17개의 샘플로 이루어져 있다. 따라서, 각 샘플을 (샘플 수×(변수×가공시간))의 2차원 데이터 셋으로 unfolding 해야 한다. unfolding은 다음과 같은 과정을 거친다.

1) 각 샘플(변수×가공시간)을 1차원의 row vector로 unfolding한다.
2) 17개의 row vector를 열 방향으로 합쳐 (샘플 수×(변수×가공 시간))의 2차원 데이터셋으로 변환

그러나 unfolding 과정을 진행하기 위해서는 FDC 데이터와 OES 데이터 간의 서로 일치하는 time stamp가 필요한데, FDC 데이터와 OES 데이터의 가공시간은 일치하지 않는다. 따라서 두 데이터 간의 동일한 Time Stamp를 맞추는 작업이 필요하다. 또한 OES 데이터의 경우 데이터가 매우 대용량이기 때문에 단순하게 모든 데이터를 unfolding 했다가는 약 3천만 차원의 1차원 row vector를 얻게 되어 계산량이 급증한다. 따라서 적절한 요약 통계량으로 OES 데이터의 크기를 줄이고 FDC 데이터와 OES 데이터의 Time Stamp 일치시키는 작업이 필요하다.

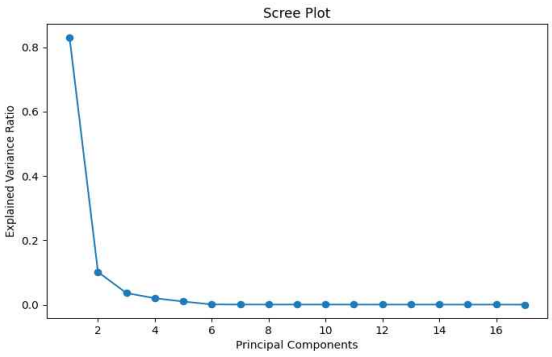
FDC 데이터에는 ASE Cycles 변수가 있다. 이는 Etching과 Passivation을 1회의 cycle로 보고, 잔여 cycle의 수를 기록한 변수이다. 웨이퍼 에칭 동안 FDC 데이터와 OES 데이터가 모두 수집되므로, 1회의 cycle은 FDC 데이터와 OES 데이터를 일치시키기 위한 타당한 기준이 된다. 따라서 FDC 데이터는 ASE Cycles 변수를 기준으로 각 cycle별 수치 데이터의 평균을 활용한다.

OES 데이터의 경우 cycle이 표기된 변수가 따로 없으나, OES 데이터는 모두 파장별 기록값을 나타내므로 시간별 강도를 측정하여 Fast Fourier Transform(FFT)을 활용, 50개의 주기로 계산하였다. 아래 그림은 한 개 데이터의 예시이다.

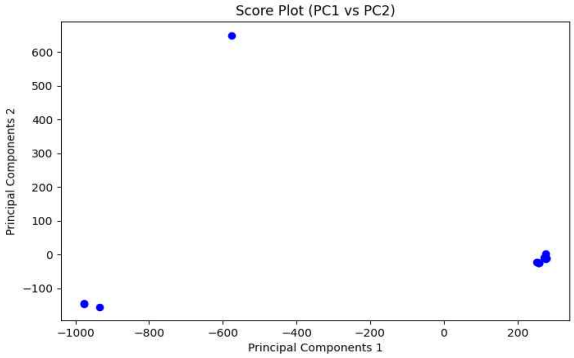


FFT를 활용하여 모든 OES 데이터의 cycle을 분석한 뒤 FDC 데이터와 마찬가지로 cycle별로 묶어 평균을 활용하여 unfolding을 수행했다. 따라서 각각 변수의 수는 변하지 않았지만 time stamp의 수가 cycle의 수인 50개로 줄어들어 훨씬 수월하게 3차원 데이터 셋(샘플 수×변수×50)을 2차원 데이터 셋(샘플 수×(변수×50))으로 unfolding 할 수 있다.

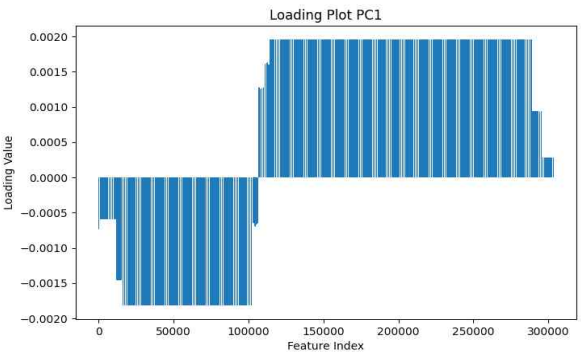
FDC 데이터와 OES 데이터를 unfolding 하여 합치는 과정을 모든 샘플(17개)에 대해서 반복하면 필요한 입력 데이터 X(17×303502 차원)를 얻을 수 있고, 입력 데이터 X를 활용해 PCA 및 kernel PCA를 수행할 수 있다. PCA 수행 후 남을 주성분의 수는 SVD를 통해 계산한 eigenvalue가 90% 이상의 데이터를 설명할 수 있는 최소 주성분의 수로 결정하였으며, Scree Plot을 활용하여 시각화하였다. 본 문제의 Scree Plot은 아래 좌상단과 같으며, 따라서 2개의 주성분을 축으로 결정하였다. 또한 PC1과 PC2의 Score Plot은 아래 우상단, PC1의 Loadiong Plot은 아래 좌하단, Kernel PCA의 누적설명변수 비율은 아래 우하단과 같다. Kernel PCA 수행 시에는 Polynomial Kernel을 활용하였다.



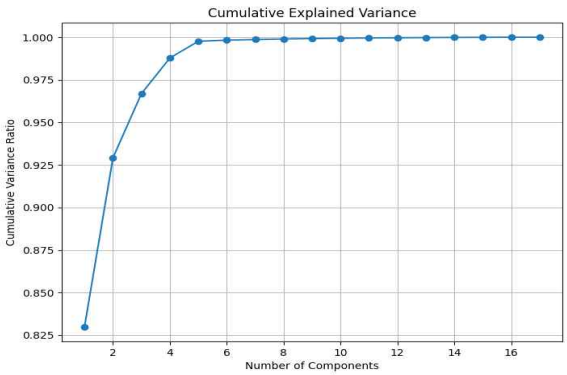
PCA Scree Plot



PCA Score Plot (PC1 vs PC2)



Loading Plot of PC1



Kernel PCA Cumulative Explained Variance(poly)

이제 303,502차원이었던 입력 데이터 X를 2개의 주성분을 축으로 하는 2차원으로 축소시키고, 선형회귀(PCR)를 수행할 수 있다. 선형회귀의 식은 다음과 같으며, 각 회귀계수는 아래의 표와 같다.

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{(i)j} + \epsilon_i \quad \text{for } i = 1, \cdots, n$$

	β_0	β_1	β_2
PCA	37.92929411764	0.00096398	-0.00147323
Kernel PCA(poly)	37.92929411764	$-4.86266381 \times 10^{-13}$	$-7.69197595 \times 10^{-13}$

각 PCA와 Kernel PCA 모델로 수행한 PCR 모델의 성능을 LOOCV를 통해 MSE, MAPE, R^2 로 검증한 결과는 아래 표와 같다. 전반적으로 PCA 모델의 설명력이 Polynomial Kernel을 사용한 Kernel PCA 모델보다 설명력이 좋음을 알 수 있는데, 먼저 MSE를 기준으로 보면 Polynomial Kernel을 사용한 Kernel PCA 모델은 361.6242, PCA 모델은 208.3308이다. MSE는 실제값과 예측값의 오차 제곱의 평균으로 정의되므로, MSE가 크다는 것은 그만큼 오차가 크다는 것을 의미한다. 따라서 Kernel PCA 모델의 오차가 PCA 모델보다 더 크다고 설명할 수 있다. MAPE는 절대 오차의 백분율 평균으로, MSE와 비슷하게 오차를 기준으로 계산되며 클수록 상대적인 오차가 크다는 것을 의미한다. 따라서 MAPE의 측면에서도 Kernel PCA 모델의 오차가 PCA 모델보다 더 크다고 설명할 수 있다. R^2 는 전체 데이터 변동성 중에 모델이 설명하는 변동성을 나타내는 지표로, 1에 가까울수록 모델의 데이터 설명력이 더 좋음을 의미한다. 비록 Kernel PCA 모델의 R^2 값이 PCA 모델보다 조금 높지만, 두 모델 모두 R^2 이 매우 낮아 데이터의 변동성을 충분히 설명하지 못하므로, 예측의 품질이 좋다고 설명할 수 없다.

	MSE	MAPE	R^2	실행 시간(s)
PCA	208.3308	0.1357	0.07036	0.02
Kernel PCA(poly)	361.6242	0.1644	0.07292	0.02

1 - (2)

동일한 데이터에 대해 다른 예측모델을 이용해서 X 데이터를 활용해 Y 값을 예측해보자. 먼저 Random Forest 알고리즘을 활용하여 원본 데이터 X에서 Y값에 대한 회귀를 실시할 수 있다. Random Forest 알고리즘은 여러 개의 Decision Tree 모델을 만들어 Ensemble을 구성한다. 각 개별 Decision Tree는 데이터를 무작위로 샘플링 해 학습하며, 회귀 문제의 최종 예측은 평균값을 활용한다.

학습의 속도를 높이기 위해 각 Tree의 depth를 5로 동일하게 고정하고, 총 200번의 반복 수행을 통해 계산될 것이며, 각 Decision Tree는 전체 변수의 약 $\frac{1}{3}$을 사용하였다. 이후 LOOCV를 17번 반복해 Random Forest 알고리즘의 성능을 각 지표별로 측정하면 아래의 표와 같다.

	MSE	MAPE	R^2	실행 시간(s)
Random Forest	0.7562	0.0178	0.8884	654.79

각 지표의 값을 볼 때, Random Forest 알고리즘의 성능은 PCA를 이용한 PCR이나 Kernel PCA를 이용한 PCR보다 월등히 높다고 할 수 있다. MSE와 MAPE의 값도 매우 작고, R^2 값도 1에 가까워 원본 데이터를 매우 잘 설명한다. 이는 실행 시간이 매우 오래 걸린 점을 고려했을 때, 별도의 차원 축소 기법을 이용하지 않고 데이터를 모두 사용했기 때문으로 추정된다.

또한 각 데이터의 특성 수가 너무 많아 계산 속도를 높인 LightGBM 알고리즘을 적용해 보았다. LightGBM 알고리즘은 Decision Tree를 weak learner로 활용한 GBM Ensemble 모델인 GBDT를 사용하며, 계산 시간을 가장 높이는 과정인 Tree의 optimal split point 탐색을 histogram based algorithm을 활용하여 시간을 단축시킨다. 또한 Leaf-wise tree splitting, Gradient-based One-Side Sampling, Exclusive Feature Bundling 등을 활용해 계산 속도를 비약적으로 향상시켰다. Light GBM 모델의 hyperparameter는 learning rate, number of leaves, max depth이다. 따라서 learning rate를 0.01, 0.05, 0.1로, number of leaves를 31, 50, 70으로, max depth를 -1(제한 없음), 10, 20 으로 조절해가며 grid search를 통해 모델의 성능이 제일 높아지는 hyperparameter의 조합을 아래와 같이 선정하고, 그 성능을 각 지표별로 측정해 보았다. 측정 결과는 아래 표와 같으며, 이 때의 learning rate는 0.01, number of leaves는 31, max depth는 -1이다.

	MSE	MAPE	R^2	실행 시간(s)
LightGBM	4.4416	0.0420	-0.0007	21.19

Light GBM 모델의 경우 분명 Random Forest 모델보다 실행 시간은 확실히 줄었지만, MSE 값이 크게 증가했으며, R^2 값이 음수로 나타나 underfitting 문제가 발생한 것으로 보인다. 즉 Light GBM 모델은 데이터를 충분히 설명하지 못하는 것으로 평가된다. hyperparameter의 추가적인 조절로 성능이 개선될 여지는 있겠으나, 이미 hyperparameter를 충분히 조절했다고 판단, Light GBM 모델이 입력 데이터와 잘 맞지 않는 모델이라고 평가할 수 있다.

전체 알고리즘의 지표를 종합하면 아래의 표와 같다. 종합해보면, PCA 후 PCR, Kernel PCA 후 PCR, Random Forest, LightGBM 의 성능을 비교해 보면 차원을 축소하고 나면 실행 시간은 월등히 빨라지지만, MSE 값이 매우 커지고 R^2 이 매우 작아져 데이터에 대한 설명력을 잃어버린다고 평가할 수 있다. PCR 모델들의 낮은 설명력은 데이터의 비선형성이 잘 반영되지 않은 결과로 해석할 수 있다. 따라서 선형 회귀를 실시하는 PCR 보다 비선형 모델을 고려하는 편이 더 추천된다. Random Forest 모델은 전체적으로 성능이 매우 뛰어나지만, 실행시간이 오래 걸린다는 단점이 있다. Light GBM 모델은 Random Forest에 비해 실행 시간이 매우 짧지만 PCR 모델에 비할 수 없었으며, R^2 값이 음수로 나온다는 점과 MSE 값이 매우 낮다는 점에서 데이터에 대한 설명력이 부족하다고 판단된다.

	MSE	MAPE	R^2	실행 시간(s)
PCA	208.3308	0.1357	0.07036	0.02
Kernel PCA(poly)	361.6242	0.1644	0.07292	0.02
Random Forest	0.7562	0.0178	0.8884	654.79
LightGBM	4.4416	0.0420	-0.0007	21.19

2

주어진 데이터 셋은 가공품의 정상/불량을 판단하는 데이터로, predictor(독립변수)는 435개, response(목표변수)는 0(정상) / 1(불량)이고, 총 학습 데이터 수는 426개이다. 데이터의 수가 독립변수보다 많은 고차원 문제이며, response 데이터의 정상 / 불량의 개수가 각각 117개 / 309개로 클래스 간 데이터 수가 과하게 치중되지 않은 클래스 균형 문제이다. 또한, 테스트 데이터의 정답 데이터가 별도로 주어지지는 않았으나, Leave-One-Out Cross Validation을 통해 F1 Score를 측정할 수 있다.

F1 Score란 precision과 recall의 조화평균이며 0과 1 사이의 값을 가진다. precision이란 모델이 참으로 분류한 것 중 실제로 참일 비율이며, recall이란 실제로 참인 것 중 모델이 참으로 분류한 것을 의미한다. 즉 precision과 recall이 모두 높으면 F1 Score 또한 높은 값을 가지며, 이는 모델의 분류 성능이 좋음을 의미한다.

먼저 Random Forest를 활용한 분류 모델을 학습시키면 결과는 다음과 같다. 학습의 효율성을 높이기 위해 각 Tree의 depth를 5로 동일하게 고정하고, 총 200번의 반복 수행을 통해 계산하였으며, 각 Decision Tree는 전체 변수의 약 $\frac{1}{3}$을 변수로 사용할 것이다.

	F1 Score	실행 시간(s)
Random Forest	0.3313	307.42

Random Forest 의 F1 Score는 위 표와 같다. F1 Score가 0.3313이라는 것은 성능이 좋지는 않음을 의미하는데, 이를 개선하기 위해 GBM(Gradient Boosting Machine) 알고리즘을 활용하여 모델을 학습시켰다. GBM은 weak learner가 원래 output과 이전 항의 추정 output의 잔차를 예측하는 과정을 반복하여 더한 알고리즘으로, 수식으로 표현하면 아래와 같다.

$$\hat{y_i} = H_M(x_i) = h_0(x_i) + \sum_{k=1}^{M} \nu h_k(x_i)$$

where M : # of weak learner, $\nu \in [0,1]$: learning rate, $h(x_i)$: weak learner, $H_M(x_i)$: strong learner

각 weak learner $h_t(x_i)$는 잔차항인 $e_t(x_i)$를 학습하여 오차를 줄여나간다. 다만, 최초 weak learner인 $h_0(x_i)$는 y에 대해 직접 학습하여 초기 정보를 제공한다. M과 ν는 Hyperparameter로, 이번 학습에서는 과적합을 줄이기 위해 초기 값을 $M = 200, \nu = 0.05$으로 설정하고 값을 변화시켜가며 결과를 관찰하였다.

분류 문제에서 GBM은 Binary Cross Entropy를 Loss Function으로 사용한다. Binary Cross Entropy란 i번째 데이터 p_i에 대해서 정상이면 1, 불량이면 0일 때, Loss Function을 아래와 같이 정의할 수 있다.

$$\max\left\{\prod_{i=1}^{n} p_i^{y_i}\left(1-p_i^{1-y_i}\right)\right\}$$

위 식에 자연로그를 취하고 최대우도함수로서 정리하면 아래 식과 같다.

$$L(y,\hat{y}) = -\frac{1}{n}\sum_{i=1}^{n}\left\{y_i \ln\left(\hat{y_i}\right) + (1-y_i)\ln\left(1-\hat{y_i}\right)\right\}$$

y_i는 실제 분류값이고 $\hat{y_i}$는 class 1로 분류될 확률이다. 또한 LOGIT을 다음과 같이 정의할 수 있는데,

$$LOGIT = \ln(odds) = \ln\left(\frac{H_m(x_i)}{1-H_m(x_i)}\right)$$

이를 활용하여 기존의 Loss Function을 편미분하면 음의 경사가 잔차에 해당함을 알 수 있다. 이를 활용하여 잔차를 학습해 모델의 성능 향상을 도모하였다.

$$L(y, H_m) = -\sum_{i=1}^{n} y_i \ln\{H_m(x_i)\} + (1-y_i)\ln 1 - H_m(x_i)$$

$$= \sum_{i=1}^{n} -y_i[LOGIT] + \ln 1 + e^{LOGIT}$$

$$\frac{\partial L(y, H_m)}{\partial H_m} = \frac{\partial}{\partial LOGIT} \sum_{i=1}^{n} -y_i[LOGIT] + \ln\{1 + e^{LOGIT}\}$$

$$= \sum_{i=1}^{n} (-y_i + H_m(x_i))$$

즉, LOGIT을 예측하는 weak learner h_m 으로 y일 확률을 예측하는 strong learner H_m을 구성하였다. 각 weak learner는 Decision Tree 모델을 활용했으며, 최대 깊이는 3으로 Leave-One-Out Cross Validation을 통해 성능을 검증한 결과는 아래 표와 같다.

	F1 Score	실행 시간(s)
GBM($M = 200, \nu = 0.05$)	0.3575	3385.93

이는 기존 Random Forest 모델의 F1 Score인 0.3313에 비해 유의미하게 높은 수치라고 보기 어려우며, 오히려 연산량이 더 많아 실행 시간이 약 10배 더 많이 소요된다. 따라서 성능 향상을 위해 hyperparameter인 weak learner의 수 M 과 learning rate ν를 변화시켜가며 학습을 진행했다. 그 결과는 아래의 표와 같다.

GBM F1 Score (실행 시간)	$\nu = 0.01$	$\nu = 0.02$	$\nu = 0.05$	$\nu = 0.1$
$M = 100$	0.1168 (1797.85s)	0.2733 (1800.18s)	0.2907 (1758.35s)	0.3616 (1965.87s)
$M = 200$	0.2767 (3374.45s)	0.2788 (3381.06s)	0.3575 (3385.93s)	0.3810 (3855.52s)
$M = 300$	0.3049 (4942.52s)	0.3095 (4921.31s)	0.3867 (4937.88s)	0.3958 (5835.20s)

위 표의 결과를 보면 먼저 weak learner의 수인 M 이 증가할수록 F1 score는 증가하는 경향이 나타남을 알 수 있다. 이는 weak learner가 증가함에 따라 GBM 모델이 데이터의 패턴을 잘 학습하게 되기 때문이다. 그러나 적절한 범위보다 커졌을 때는 과적합으로 이어질 수 있어 weak learner의 수를 조절하는데 있어 주의가 필요하다. 또한 weak learner의 수가 동일할 때, learning rate가 높아질수록 F1 Score가 증가하는 경향이 나타난다. 일반적으로 learning rate의 증가는 GBM 모델이 개별 weak learner에 부여하는 가중치가 커진다는 의미로, 모델의 빠른 수렴을 예측할 수 있으나, 과적합의 위험이 크다. 그러나 너무 낮으면 모델의 변화가 작아지므로 학습 시간이 너무 길어지는 단점이 있다. 주로 weak learner의 수 M 과 learning rate ν 는 서로 trade-off 관계에 있다.

결론적으로, 주어진 문제와 같이 데이터의 수보다 독립변수의 수가 더 많은 고차원 데이터의 클래스 균형 분류 문제를 해결하기 위해서는 Random Forest 모델보다는 GBM 모델의 성능이 더 높은 F1 Score를 나타낸다는 것을 알 수 있다. 또한 GBM 모델의 성능을 개선하기 위해 weak learner의 수와 learnng rate를 조절해 본 결과, weak learner의 개수가 300개일 때, learning rate가 0.1일 때 성능이 더 좋아짐을 알 수 있었다.

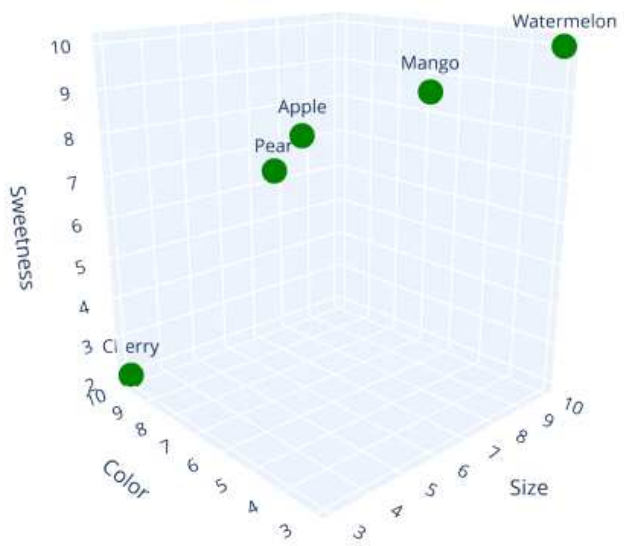
3

Metric Multidimensional Scaling(Metric MDS) 알고리즘은 고차원 데이터를 저차원(2~3차원)으로 변경하며 데이터 간의 거리와 유사도를 보존하여 임베딩하는 기법이다. 주로 고차원 데이터의 데이터 포인트 간의 거리 혹은 유사도를 그대로 보존하며 2차원이나 3차원으로 공간에 임베딩하여 시각화하기 위해 사용된다. Metric MDS 알고리즘의 핵심 아이디어는, 결국 두 데이터 포인트 간의 거리를 최대한 보존하는 저차원 임베딩 벡터를 찾는 것이다.

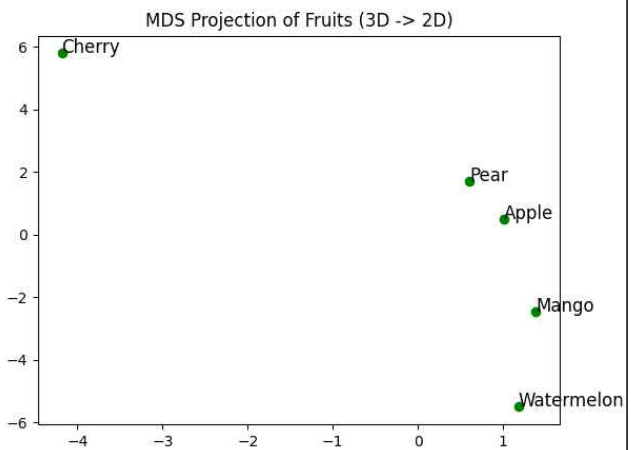
간단한 예를 들어 Metric MDS 알고리즘을 직관적으로 이해해 보자. 다음과 같은 과일 데이터가 있다고 가정하자.

과일명	크기	색상	당도
사과	6	7	8
배	6	8	7
수박	10	3	10
체리	3	10	2
망고	8	5	9

이 데이터를 3차원에서 나타내면 아래 왼쪽 그림과 같고, 여기에 MDS를 수행하면 아래 오른쪽 그림과 같다.



과일 데이터의 3차원 plot



과일 데이터의 2차원 plot(MDS 적용)

Metric MDS 알고리즘을 활용하여 각 데이터 포인트 간의 거리는 그대로 보존한 채 2차원으로 차원을 축소하여 시각화하였다. 본 예시에서는 원본 데이터가 3차원이었기에 시각화에 무리가 없었지만 차원이 높아질수록, 즉 데이터의 feature가 많아질수록 시각화가 어렵기 때문에 Metric MDS 알고리즘을 활용하여 시각화함으로써 데이터에 대한 이해를 높일 수 있다.

이제 Metric MDS 알고리즘을 수학적으로 접근하기 위해 각 데이터 포인트 간의 유클리드 거리 정보가 담겨있는 거리 행렬 D 가 있다고 가정하자. 이 때 거리행렬은 D 가 두 점 x_i, x_j 의 거리인 d_{ij} 를 활용하여 아래와 같이 정의할 수 있다.

$$D = (d_{ij})$$

점 i 와 점 j 의 거리 d_{ij} 의 제곱을 다음과 같이 계산한 뒤,

$$d_{ij}^2 = ||x_i - x_j||^2 = (x_i - x_j) \cdot (x_i - x_j)$$

각 열의 평균을 0으로 조정한 중심화 행렬 B 를 다음과 같이 만들 수 있다.

$$B_{ij} = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{n}\sum_k d_{ik}^2 - \frac{1}{n}\sum_k d_{kj}^2 + \frac{1}{n^2}\sum_{kl} d_{kl}^2\right)$$

중심화 행렬 B 는 내적행렬로, Eigen-decomposition을 사용해 아래와 같이 분해할 수 있다. 이 때, Q 는 B 의 Eigenvector로 이루어진 행렬, Λ는 Eigenvalue를 값으로 가지는 대각행렬이다.

$$B = Q\Lambda Q^T$$

Eigen-decompostion의 결과에 따라 저차원(k 차원) embadding Y 는 아래와 같이 계산할 수 있다.

$$Y = Q_k \Lambda_k^{\frac{1}{2}}$$

거리행렬 D 가 주어졌을 때의 metric MDS 알고리즘은 위와 같이 진행된다. 하지만 거리행렬 D가 아닌 원본 데이터 X가 주어지고 오히려 거리 행렬 D 가 주어지지 않은 상황이라면, 원본 데이터 X 를 통해 위 과정을 진행해야 한다. 이 때, 원본 데이터 X 의 각 열의 평균이 0이 되도록 조정한 데이터 $\hat{X}$ 를 활용하여 중심화 행렬 B 를 아래와 같이 구할 수 있다.

$$B = \hat{X}\hat{X}^T$$

$\hat{X}\hat{X}^T$는 데이터 $\hat{X}$ 의 내적으로 이루어진 행렬이며, 이는 $\hat{X}$ 의 데이터 벡터 간의 유사도를 간직하고 있는 행렬이라고 이야기할 수 있다. 또한, 이미 중심화가 완료되었기 때문에 중심화 행렬 B 를 손쉽게 대체할 수 있다. 즉, $\hat{X}\hat{X}^T$에 Eigen Decomposition을 수행하여 다음과 같은 결과를 얻을 수 있다. 이 때, Q 는 $\hat{X}\hat{X}^T$의 Eigenvector로 이루어진 행렬, Λ는 Eigenvalue를 값으로 가지는 대각행렬이다.

$$\hat{X}\hat{X}^T = Q\Lambda Q^T$$

이 때 저차원(k 차원) embadding Y 는 위의 일반적인 MDS 알고리즘과 같이 아래처럼 계산할 수 있다.

$$Y = Q_k \Lambda_k^{\frac{1}{2}}$$

이제 처음에 제시한 과일의 사례를 다시 보면, 3차원 데이터 X 를 2차원으로 embedding하는 문제로 바라볼 수 있다. 원본 데이터를 행렬로 만들면 $\begin{pmatrix} 6 & 7 & 8 \\ 6 & 8 & 7 \\ 10 & 3 & 10 \\ 3 & 10 & 2 \\ 8 & 5 & 9 \end{pmatrix}$ 인데, 이를 각 열의 평균을 0으로 조정해 다시 행렬 X 를 구성하고, XX^T를 구하면 아래와 같다.

$$X = \begin{pmatrix} -0.6 & 0.4 & 0.8 \\ -0.6 & 1.4 & -0.2 \\ 3.4 & -3.6 & 2.8 \\ -3.6 & 3.4 & -5.2 \\ 1.4 & -1.6 & -1.8 \end{pmatrix}, XX^T = \begin{pmatrix} 1.16 & 0.76 & -1.24 & -0.64 & -0.04 \\ 0.76 & 2.36 & -7.64 & 7.96 & -3.44 \\ -1.24 & -7.64 & 32.36 & -39.04 & 15.56 \\ -0.64 & 7.96 & -39.04 & 51.56 & -19.84 \\ -0.04 & -3.44 & 15.56 & -19.84 & 7.76 \end{pmatrix}$$

이제 XX^T에 대해 Eigen Decomposition을 수행하면 아래와 같은 결과를 얻을 수 있다. 이 때, Q 는 $\hat{X}\hat{X}^T$의 Eigenvector로 이루어진 행렬, Λ는 Eigenvalue를 값으로 가지는 대각행렬이다. 계산 상의 편의를 위해 소수점 넷째 자리까지 계산했다.

$$XX^T = Q\Lambda Q^T$$

$$Q = \begin{pmatrix} -0.004 & -0.5493 & 0.5758 & -0.0929 & -0.5984 \\ -0.1282 & -0.4401 & -0.7681 & 0.2473 & -0.3726 \\ 0.5862 & 0.5208 & -0.1359 & -0.0929 & -0.5984 \\ -0.7451 & 0.4828 & 0.1081 & 0.2473 & -0.3726 \\ 0.2911 & -0.0143 & 0.22 & 0.9276 & 0.0788 \end{pmatrix}$$

$$\Lambda = \begin{pmatrix} 91.3907 & 0 & 0 & 0 & 0 \\ 0 & 3.506 & 0 & 0 & 0 \\ 0 & 0 & 0.3034 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

X의 2차원 embedding Y 는 아래 식과 같이 계산할 수 있으며, 따라서

$$Y = Q_2 \Lambda_2^{\frac{1}{2}}$$

따라서, X의 2차원 embedding Y 는 아래와 같다.

$$Q_2 = \begin{pmatrix} -0.004 & -0.5493 \\ -0.1282 & -0.4401 \\ 0.5862 & 0.5208 \\ -0.7451 & 0.4828 \\ 0.2911 & -0.0143 \end{pmatrix}, \Lambda_2 = \begin{pmatrix} 91.9307 & 0 \\ 0 & 3.506 \end{pmatrix}, \Lambda_2^{\frac{1}{2}} = \begin{pmatrix} 9.5598 & 0 \\ 0 & 1.8724 \end{pmatrix}$$

$$Y = Q_2 \Lambda_2^{\frac{1}{2}} = \begin{pmatrix} -0.004 & -0.5493 \\ -0.1282 & -0.4401 \\ 0.5862 & 0.5208 \\ -0.7451 & 0.4828 \\ 0.2911 & -0.0143 \end{pmatrix} \begin{pmatrix} 9.5598 & 0 \\ 0 & 1.8724 \end{pmatrix} = \begin{pmatrix} -0.038 & -1.0286 \\ -1.2256 & -0.824 \\ 5.6039 & 0.9752 \\ -7.1231 & 0.904 \\ 2.7829 & -0.0267 \end{pmatrix}$$

이제 2차원으로 embedding 된 Y 와 원래 데이터 X 간의 유사도가 보존되고 있는지 확인할 수 있다. 각 X 와 Y 의 유사도가 보존되고 있는지 확인하기 위해, XX^T 와 YY^T 간의 상대적 Forbenius Norm인 다음의 식을 활용하여 계산하면 그 값이 0.3%로 매우 두 행렬이 매우 유사함을 알 수 있다. 따라서 2차원 embedding Y 는 원본 데이터 X 의 데이터간 유사도를 잘 보존하면서 2차원으로 embedding 됐음을 알 수 있다.

$$\frac{||XX^T - YY^T||_F}{||XX^T||_F} \text{ where } ||A||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|A_{ij}|^2}$$

이 과정은 PCA를 통한 차원 축소 과정과도 언뜻 비슷해보인다. PCA는 아래 식처럼 원본 데이터 X 를 분해하는데, Metric MDS와 수식이 유사하다.

$$X^TX = Q\Lambda Q^T$$

여기서는 X^TX 와 XX^T 의 차이에 대해서 살펴보아야 한다. PCA에서 사용되는 X^TX 는 데이터 X 의 공분산 행렬로 이해할 수 있다. PCA의 주 목적은 분산이 가장 큰 방향으로 축을 새로 설정하는 것으로, 분산이 큰 방향의 축이 데이터를 가장 잘 설명한다는 전제 하에 이루어진다. 따라서 공분산 행렬 X^TX 의 eigenvector와 eigenvalue를 구하여 k 개의 축을 갖는 저차원 좌표계로 변환함으로써 데이터의 차원을 축소시킨다. 즉, PCA는 원본 데이터의 변수 간 분산을 보존하는 방향으로 차원이 축소된다. 이 때 원본 데이터의 각 변수가 PCA 후의 주성분에 얼마나 기여하는지 나타내는 가중치가 loading vector이다.

이와 반대로 Metric MDS는 XX^T 를 사용하는데, 이는 데이터 X 의 데이터 간의 내적 행렬로 이해할 수 있다. Metric MDS의 주 목적은 각 데이터 간의 거리를 보존할 수 있도록 각 데이터를 embedding 하는 것으로, 각 데이터 간의 유사도를 시각화하거나 거리 데이터를 활용할 수 있을 때 사용된다. Metric MDS는 데이터의 내적 행렬 XX^T 의 eigenvector와 eigenvalue를 구하고, k 개의 축을 갖는 저차원 좌표계로 변환함으로써 데이터의 차원을 축소시킨다. 즉 Metric MDS는 원본 데이터 간의 거리를 보존하는 방향으로 차원이 축소된다. 이 때 내적 행렬 XX^T 의 eigenvector는 각 데이터가 저차원 방향의 축에 얼마나 기여하는지를 나타내는 축 방향의 벡터이다.

만약 두 데이터 포인트 간의 거리가 비유클리드 거리거나 순서나 순위를 보존하고 싶을 때, 혹은 고유값 중 일부가 음수여서 고유값 분해에 실패할 때는 Non-metric MDS라고 부르며, iterative한 방식으로 gradient descent 방식을 활용한다. 이 때의 Loss function은 stress function을 활용한다. stress function은 원본 데이터와 저차원 embedding 데이터 간의 거리를 정량적으로 측정하며, 아래와 같이 정의된다.

$$STRESS = \sqrt{\frac{\sum_{i<j}\left(d_{ij} - \hat{d}_{ij}\right)^2}{\sum_{i<j} d_{ij}^2}}$$

Non-metric MDS 알고리즘은 이 stress function을 최소화하는 것을 목표로 최적화를 진행한다. 이 때 최적화는 이 stress function의 편미분을 통해 gradient descent metho로 업데이트를 진행하며 이루어지게 된다. 먼저 stress function의 기울기는 아래와 같다.

$$\frac{\partial STRESS}{\partial y_{ia}} = \sum_{j \neq i}\left(\frac{\widehat{d_{ij}} - d_{ij}}{\widehat{d_{ij}}}\right)\left(y_i - y_j\right)_a$$

이 기울기를 활용해 좌표를 갱신하는 방법은 아래의 식으로 진행된다.

$$y_i^{(t+1)} = y_i^{(t)} - \alpha \frac{\partial STRESS}{\partial y_{ia}}$$

이 때, $y_i^{(t+1)}$ 은 $t+1$ 반복에서의 점 i 의 새로운 좌표, $y_i^{(t)}$ 는 t 반복에서의 점 i 의 좌표이며, α는 learning rate이다. 이를 반복해가며 좌표를 갱신하다가 stress function의 값이 수렴하거나 최대 반복 횟수에 도달하면 갱신을 멈추고 최종 embedding matrix인 Y 를 출력한다.

아래는 위의 과정을 pseudo-code로 표현한 것이다.

$D \leftarrow$ Distance Matrix ($n \times n$)

$k \leftarrow$ Target dimension(usually 2 or 3)

$\alpha \leftarrow$ Learning Rate

$\epsilon \leftarrow$ Convergence threshold

$T \leftarrow$ Maximum number of iteration

$Y \leftarrow$ Low Dimensional Embedding ($n \times k$)

$Y_0 \leftarrow$ Random initializing as initial embedding ← **최초 embedding은 random하게 생성**

for $t = 1$ to T do

for all i, j:

$\partial_{ij} = ||Y_{ti} - Y_{tj}||$ ← **현재 embedding에서 모든 점 간의 거리 계산**

$$STRESS_t = \sqrt{\frac{\sum_{i<j}(D_{ij} - \partial_{ij})^2}{\sum_{i<j} D_{ij}^2}}$$ ← **현재 t 시점에서의 stress 계산**

for all point i:

$$\frac{\partial STRESS}{\partial Y_t} = \sum\left(\frac{\partial_{ij} - d_{ij}}{\partial_{ij}}\right)(Y_t(i) - Y_t(j))$$ ← **현재 t 시점에서의 stress gradient 계산**

$$Y_{t+1} := Y_t - \alpha \frac{\partial STRESS}{\partial Y_t}$$ ← **t+1 시점에서의 좌표 업데이트**

if $|STRESS_t - STRESS_{t-1}| < \epsilon$ or $t \geq T$:

return ← **stress 값이 임계치보다 작거나 최대 반복 횟수에 도달하면 중지**

end if

end for

end for

만약 거리 행렬 D 가 주어지지 않아 두 점 사이의 거리 d_{ij} 를 알 수 없다면, 두 점 사이의 거리 d_{ij} 를 기반으로 하는 stress function을 계산할 수 없다. 하지만 stress function은 서로 다른 두 차원에서의 거리를 보존하므로, 거리 d_{ij} 자리에 두 벡터 간의 유사도를 대입하면 다음과 같은 stress function을 정의하여 Non-metric MDS 알고리즘에 사용할 수 있다. (X 는 각 열벡터의 평균을 0으로 조정한 데이터 행렬이다.)

$$STRESS = \sqrt{\frac{\sum_{i<j}\left((XX^T)_{ij} - \left(\widehat{XX^T}\right)_{ij}\right)^2}{\sum_{i<j}(XX^T)_{ij}^2}}$$

XX^T 는 내적 행렬로서 두 벡터 간의 유사도, 즉 내적에 대한 정보를 담고 있으므로, 위와 같은 stress function은 서로 다른 두 차원끼리의 벡터 간의 유사도, 즉 내적의 차이를 보존하게 된다. 따라서, 저차원 공간에서의 내적이 고차원 공간에서의 내적과 유사할수록 stress 값은 0에 가까워지고, 위에서 서술한 Non-metric MDS 알고리즘을 적용할 수 있다.