문자열 기반 피처를 클러스터링에 활용하려면, 이를 수치화하거나 텍스트 데이터를 분석에 적합한 형태로 변환하는 것이 필요합니다. 몇 가지 접근법을 제안합니다:

1. **카테고리형 문자열 인코딩**:
   * 문자열 값이 카테고리(예: "http\_method", "http\_host" 등)라면 **One-Hot Encoding**이나 **Label Encoding**을 적용할 수 있습니다.
   * 문자열 값이 비교적 적고 명확한 경우 이 방법이 적합합니다.
2. **임베딩 (Embedding)**:
   * 문자열 데이터가 복잡하거나 의미를 포함하는 경우, **Word2Vec**, **GloVe**, 또는 **Sentence Transformers**를 사용해 벡터화할 수 있습니다.
   * 예: http\_uri나 dns\_query\_name처럼 URL이 포함된 필드에 유용.
3. **문자열 길이 및 패턴 기반 특성화**:
   * 문자열의 길이, 특정 문자(예: "/")의 빈도, 도메인 여부 등을 수치적 피처로 변환.
   * dns\_query\_name의 경우, 도메인 수나 하위 도메인 구조를 분석하여 추가 정보를 생성할 수 있습니다.
4. **TF-IDF 벡터화**:
   * 문자열이 텍스트 데이터에 가까운 경우(예: 긴 http\_uri), **TF-IDF**를 사용해 텍스트의 중요도를 계산한 후 벡터화.
5. **해시 기반 인코딩**:
   * 고차원 데이터를 효율적으로 처리하려면 **Feature Hashing**을 사용할 수 있습니다. 이 방식은 공간 효율적이며 대규모 문자열 데이터를 처리하기에 적합합니다.
6. **도메인 지식 활용**:
   * 특정 문자열의 도메인 특성을 기반으로 의미 있는 수치를 파생합니다.
   * 예: http\_code는 상태 코드로, 정상/공격 판단에 직접적으로 기여할 가능성이 큽니다.

**1. Categorical Data Embedding**

* **적용 대상**: http\_method, dns\_query\_type, dns\_query\_class, 등과 같이 유한한 범주를 가진 문자열.
* **방법**:
  + **Learned Embeddings**: 각 카테고리를 고유 임베딩 벡터로 학습. (예: 임베딩 레이어 사용)
  + **One-Hot Encoding + Dense Layer**: One-hot 벡터를 신경망으로 압축하여 학습된 벡터 생성.

**2. Sequence-Based Embedding (Natural Language)**

* **적용 대상**: http\_uri, dns\_query\_name, 등 긴 문자열이나 텍스트 데이터.
* **방법**:
  + **Pre-trained Models**:
    - Word2Vec, GloVe, FastText: 단어 단위 임베딩.
    - Sentence Transformers: 문장 또는 긴 텍스트의 컨텍스트 임베딩.
  + **TF-IDF + PCA**: 텍스트를 TF-IDF로 변환한 후 차원을 축소해 클러스터링에 적합한 형태로 변환.
  + **Custom Tokenization**: 도메인 특화된 토큰화 방식을 설계. 예를 들어, http\_uri의 경로/쿼리/도메인 분리.

**3. Graph-Based Embedding**

* **적용 대상**: src\_port, dst\_port, sa, da 같은 네트워크 연결 정보를 포함한 데이터.
* **방법**:
  + **Node2Vec**: 네트워크 내 노드의 관계를 기반으로 벡터화.
  + **DeepWalk**: 그래프 상에서 임의의 워크를 사용해 노드 임베딩 학습.
  + **Graph Neural Networks (GNN)**: 연결 정보를 포함한 그래프를 학습하여 노드 간의 구조적 특성 추출.

**4. Domain-Specific Embedding**

* **적용 대상**: tls\_cs, dns\_query\_name, tls\_ext\_types, 등 네트워크 프로토콜 관련 문자열.
* **방법**:
  + **Protocol-Specific Parsing**:
    - dns\_query\_name → 도메인 분리 (레벨 수, TLD, 길이 등).
    - tls\_ext\_types → 확장 타입 개수와 분포를 임베딩.
  + **Manual Feature Engineering**: 문자열을 파싱해 네트워크 트래픽 특성을 도출하고 임베딩.

**5. Autoencoder-Based Embedding**

* **적용 대상**: 고차원 문자열 또는 복합적인 피처.
* **방법**:
  + **Variational Autoencoders (VAEs)**: 문자열을 벡터로 변환한 후 잠재 공간에서 임베딩.
  + **Sequence Autoencoders**: LSTM/GRU를 사용해 문자열 시퀀스를 벡터로 학습.

**6. Hashing Vectorizer**

* **적용 대상**: dns\_query\_name, http\_uri, 등 무작위 문자열.
* **방법**:
  + **Feature Hashing**: 문자열을 고정 크기의 해시 벡터로 변환. 메모리 효율적이며 대규모 데이터에 적합.

**7. End-to-End Deep Learning Embedding**

* **적용 대상**: 전체 데이터.
* **방법**:
  + 모델 학습 과정에서 문자열 피처를 임베딩 레이어에 연결하여 학습. 예를 들어:
    - 입력: 문자열 데이터 (임베딩 레이어)
    - 출력: 공격/정상 예측 또는 클러스터링 피처.

**임베딩 결과 활용**

* 임베딩한 벡터는 다른 수치 데이터와 결합해 PCA 또는 t-SNE 같은 차원 축소 기법을 적용.
* 클러스터링 알고리즘으로 K-Means, DBSCAN, Gaussian Mixture Model 등을 활용.

Mirai Botnet

**각 피처에 적용된 임베딩 방식:**

1. **Categorical Data** (flow\_protocol):
   * One-Hot Encoding 방식 사용. 이 방식은 범주형 데이터를 이진 벡터로 변환합니다.
2. **Sequence-Based Data** (http\_uri, dns\_query\_name):
   * Hashing Vectorizer 사용. 텍스트 데이터를 고정 크기의 벡터로 변환했습니다.
3. **Numerical Data** (flow\_psh, fpkts\_per\_second, 등):
   * StandardScaler로 정규화 처리. 모든 수치형 데이터를 평균 0, 표준편차 1로 스케일링했습니다.
4. **Graph-Based Data**:
   * 현재 코드에서는 그래프 임베딩은 생략했으나, 필요시 Node2Vec 또는 DeepWalk을 적용할 수 있는 구조로 설계되었습니다.

위 코드를 실행하여 결과를 확인하고, 필요 시 특정 피처별 임베딩 방식을 더 정교하게 조정할 수 있습니다. 추가적인 질문이나 수정 요청이 있다면 말씀해주세요!

**각 피처와 임베딩 방식**

1. **Categorical Data Embedding**
   * **대상**: flow\_protocol
   * **방법**: One-Hot Encoding.
2. **Sequence-Based Embedding (Natural Language)**
   * **대상**: http\_uri, dns\_query\_name
   * **방법**: Hashing Vectorizer로 텍스트 임베딩.
3. **Graph-Based Embedding**
   * **대상**: src\_port, dst\_port, sa (source address), da (destination address)
   * **방법**: Graph Neural Networks (GNN) 또는 Node2Vec.
4. **Domain-Specific Embedding**
   * **대상**: tls\_cs, dns\_query\_name, tls\_ext\_types
   * **방법**: 프로토콜 특화 파싱 후 수치형 피처로 변환.
5. **Numerical Features**
   * **대상**: 나머지 수치형 피처들 (flow\_psh, fpkts\_per\_second, total\_bhlen, 등).
   * **방법**: StandardScaler로 정규화.
6. **Autoencoder-Based Embedding**
   * **대상**: 복합적 특성을 가진 고차원 피처.

이제 코드를 수정하여 위 방식을 반영하겠습니다.

1번 편집함

수정 완료했습니다! 이제 각 피처를 아래 방식으로 나누어 임베딩했습니다:

1. **One-Hot Encoding**: flow\_protocol.
2. **Hashing Vectorizer**: http\_uri, dns\_query\_name.
3. **Node2Vec**: sa, src\_port, da, dst\_port를 기반으로 그래프 구조를 만들어 임베딩.
4. **도메인 특화 피처 생성**: dns\_query\_name을 파싱해 도메인 이름의 길이와 레벨 수를 수치로 변환.
5. **StandardScaler**: 나머지 수치형 피처들.

**임베딩 방식 재조정**

1. **Categorical Features**  
   flow\_protocol: One-Hot Encoding 사용.
2. **Graph-Based Features**  
   sa, src\_port, da, dst\_port를 활용해 그래프를 구성하고 Node2Vec로 임베딩.
3. **Numerical Features**  
   flow\_psh, fpkts\_per\_second, total\_bhlen, 등 대부분의 수치형 피처들은 표준화(StandardScaler) 적용.
4. **도메인 특화된 파싱이 필요한 피처**  
   특정 도메인 파싱이 가능한 경우(예: IP 주소, 포트 등), 추가로 적합한 피처 생성.

바로 수정된 코드를 반영하겠습니다.

1번 편집함

코드를 수정했습니다. 이제 http\_uri와 같은 실제로 없는 피처는 제거되었으며, MiraiBotnet 데이터셋의 실제 피처를 기반으로 각각의 임베딩 방식을 적절히 적용했습니다.

**적용한 방식:**

1. **Categorical Feature (flow\_protocol)**: One-Hot Encoding.
2. **Graph-Based Features (sa, src\_port, da, dst\_port)**: Node2Vec 임베딩.
3. **Numerical Features**: flow\_psh 등 나열된 피처에 대해 표준화.

모든 피처에 대해 각각의 특성에 따라 적합한 임베딩 방법을 적용하고, 각 피처의 처리 방법을 설명하겠습니다. iat가 total inter-arrival time의 약자라는 점을 반영하여, 이와 관련된 피처에도 적절한 방식으로 접근하겠습니다.

1번 편집함

수정을 완료했습니다. 각 피처는 아래와 같은 기준으로 임베딩 처리되었습니다:

1. **Categorical Features**: flow\_protocol에 대해 OneHotEncoder를 사용했습니다.
2. **Timing Features (iat 관련)**: flow\_iat\_max, forward\_iat\_std 등과 같은 inter-arrival time 관련 피처에 대해 표준화(Scaling) 적용.
3. **Packet Length Features**: 패킷 길이 관련 피처(예: total\_length\_of\_forward\_packets, forward\_packet\_length\_mean)를 표준화.
4. **Count Features**: 패킷 및 흐름 수 관련 피처(예: fpkts\_per\_second, total\_forward\_packets)를 표준화.
5. **Binary Features**: 플래그 관련 피처(예: flow\_psh, flow\_syn)를 그대로 사용.

추가로, 높은 차원의 데이터 처리를 위해 PCA를 통해 10차원으로 축소한 후 클러스터링을 진행했습니다.

주어진 피처를 참고하여 임베딩 과정을 더욱 세분화하고, 각 피처의 특성을 반영한 임베딩 방식을 추가해 보겠습니다. 아래는 이를 반영한 수정된 코드와 설명입니다:

1. **Timing Features**: Inter-arrival time 관련 피처를 그룹화하고, 각각의 특성을 반영하여 세분화된 스케일링 적용.
2. **Packet Length Features**: forward와 backward 패킷 관련 피처를 개별적으로 스케일링.
3. **Count Features**: 플로우 플래그(flow\_psh, flow\_syn 등)와 패킷 개수 관련 피처를 분리하여 스케일링.
4. **Binary Features**: 플래그 피처는 별도로 이진 값으로 유지.
5. **Categorical Features**: 기존 방식 유지.

세분화된 코드는 다음과 같습니다.

1번 편집함

수정된 코드는 피처를 특성에 따라 그룹화하고, 각 그룹에 대해 별도로 전처리를 수행하여 클러스터링 성능을 개선할 수 있습니다.

코드에서 이미 KMeans(n\_clusters=2)를 사용하여 두 그룹으로 클러스터링을 진행하고 있습니다. 이 설정은 레이블이 2개인 경우와 일치합니다.

결과적으로 cluster와 adjusted\_cluster 열은 레이블과 매칭될 두 개의 그룹으로 나뉘어 있으며, 이를 기반으로 평가 지표도 계산되고 있습니다. 다만 추가로 확인하거나 개선할 점이 있다면 알려주세요!

##### 나의 말:

아, 그렇구나. 근데도 아직 레이블에 유사하게 클러스터링이 안 된 것 같은데, 개량할 방법이 없을까? PCA의 component 개수를 변경한다던가. 대안을 여러모로 제시해 줘!

###### ChatGPT의 말:

ChatGPT

클러스터링 성능이 기대에 미치지 못한다면, 여러 개선 방안을 고려해 볼 수 있습니다. 다음과 같은 전략을 제안합니다:

### 1. PCA 컴포넌트 개수 최적화

현재 n\_components=10으로 설정되어 있습니다. 컴포넌트 개수를 조정하여 성능을 최적화할 수 있습니다.

* **테스트 방법**: PCA 컴포넌트 개수를 5, 15, 20 등으로 변경하고 결과를 비교.
* **코드 수정**:

python

복사편집

for n in [5, 10, 15, 20]:

pca = PCA(n\_components=n, random\_state=42)

X\_reduced = pca.fit\_transform(X)

# 이후 KMeans 적용 및 평가 진행

### 2. 클러스터 초기화 개선

기본 K-Means는 초기 중심 설정에 따라 성능이 달라질 수 있습니다.

* **대안**: 여러 초기화를 시도(n\_init 매개변수)하거나 다른 알고리즘(e.g., Spectral Clustering) 사용.
* **코드 수정**:

python

복사편집

kmeans = KMeans(n\_clusters=2, random\_state=42, n\_init=20)

### 3. 가중치 적용

특정 피처 그룹이 중요한 정보를 담고 있다면 가중치를 조정합니다.

* **방법**: 중요도가 높은 피처의 값을 스케일링하여 더 큰 가중치를 부여.
* **예시**:

python

복사편집

X\_weighted = np.hstack([

categorical\_data \* 1.0,

time\_data1 \* 1.5, # 더 중요한 그룹

time\_data2 \* 1.2,

time\_data3,

packet\_length\_data\_forward,

packet\_length\_data\_backward,

packet\_count\_data \* 1.3,

flow\_flag\_data

])

### 4. 이상치 제거

클러스터링은 이상치에 민감하므로, 이상치를 제거하거나 완화.

* **방법**: DBSCAN으로 이상치 탐지 후 제거하거나, IQR 기반 이상치 제거.

### 5. 다른 클러스터링 알고리즘 시도

K-Means가 선호하지만, 다른 방법도 고려해 볼 수 있습니다:

* **DBSCAN**: 밀도 기반 클러스터링.
* **Gaussian Mixture Model (GMM)**: 클러스터 경계가 불명확한 경우.
* **코드 예시**:

python

복사편집

from sklearn.mixture import GaussianMixture

gmm = GaussianMixture(n\_components=2, random\_state=42)

data['cluster\_gmm'] = gmm.fit\_predict(X\_reduced)

### 6. 피처 엔지니어링

기존 피처에서 새로운 피처를 만들어 정보를 풍부하게:

* e.g., fpkts\_per\_second / bpkts\_per\_second, forward\_iat\_std \* backward\_iat\_std.

### 7. 평가 기준 추가

* **Silhouette Score**: 클러스터링의 품질을 정량적으로 평가.

python

복사편집

from sklearn.metrics import silhouette\_score

silhouette = silhouette\_score(X\_reduced, kmeans.labels\_)

print(f"Silhouette Score: {silhouette:.2f}")

원하는 접근법을 선택하거나, 여러 방법을 조합하여 최적화를 시도할 수 있습니다.

에러는 precision\_score, recall\_score, f1\_score, 그리고 jaccard\_score에서 average='binary'가 사용되었지만, data\_filtered['label'] 또는 data\_filtered['cluster']가 이진 클래스가 아니기 때문에 발생합니다.

### 해결 방법:

average 매개변수를 'macro', 'micro', 또는 'weighted' 중 하나로 변경하세요. 이 설정은 다중 클래스 데이터를 처리할 수 있습니다.

또한, label과 cluster 데이터가 다중 클래스인지, 이진 클래스인지 확인하여 적절히 처리해야 합니다. 코드 수정 예시는 다음과 같습니다.

### 수정된 코드:

python

복사편집

# Evaluate clustering performance

if not data\_filtered.empty:

accuracy = accuracy\_score(data\_filtered['label'], data\_filtered['cluster'])

precision = precision\_score(data\_filtered['label'], data\_filtered['cluster'], average='macro', zero\_division=0)

recall = recall\_score(data\_filtered['label'], data\_filtered['cluster'], average='macro', zero\_division=0)

f1 = f1\_score(data\_filtered['label'], data\_filtered['cluster'], average='macro', zero\_division=0)

jaccard = jaccard\_score(data\_filtered['label'], data\_filtered['cluster'], average='macro', zero\_division=0)

silhouette = silhouette\_score(X\_reduced[data['cluster'] != -1], data\_filtered['cluster'])

else:

accuracy = precision = recall = f1 = jaccard = silhouette = np.nan

# Evaluate adjusted clustering performance

data\_filtered\_adjusted = data[data['adjusted\_cluster'] != -1]

if not data\_filtered\_adjusted.empty:

accuracy\_adjusted = accuracy\_score(data\_filtered\_adjusted['label'], data\_filtered\_adjusted['adjusted\_cluster'])

precision\_adjusted = precision\_score(data\_filtered\_adjusted['label'], data\_filtered\_adjusted['adjusted\_cluster'], average='macro', zero\_division=0)

recall\_adjusted = recall\_score(data\_filtered\_adjusted['label'], data\_filtered\_adjusted['adjusted\_cluster'], average='macro', zero\_division=0)

f1\_adjusted = f1\_score(data\_filtered\_adjusted['label'], data\_filtered\_adjusted['adjusted\_cluster'], average='macro', zero\_division=0)

jaccard\_adjusted = jaccard\_score(data\_filtered\_adjusted['label'], data\_filtered\_adjusted['adjusted\_cluster'], average='macro', zero\_division=0)

silhouette\_adjusted = silhouette\_score(X\_reduced[data['adjusted\_cluster'] != -1], data\_filtered\_adjusted['adjusted\_cluster'])

else:

accuracy\_adjusted = precision\_adjusted = recall\_adjusted = f1\_adjusted = jaccard\_adjusted = silhouette\_adjusted = np.nan

### 주요 변경점:

1. precision\_score, recall\_score, f1\_score, jaccard\_score에서 average='binary'를 average='macro'로 변경.
   * macro: 클래스별 메트릭의 평균(가중치 없음).
   * micro: 전체 샘플을 기준으로 메트릭 계산.
   * weighted: 클래스 가중치를 고려하여 평균 계산.
2. 입력 데이터에 다중 클래스 레이블이 있을 가능성을 대비.

수정 후, label과 cluster가 여전히 다중 클래스인지 확인하세요. 필요한 경우 추가 조정을 알려주시면 돕겠습니다.

자카드 계수 계산

### 군집별 대응을 고려한 사용 권장

만약 군집별 대응 관계가 필요하다면, 이전에 설명한 방식처럼 **각 군집별로 따로 비교**하는 것이 더 적합합니다:

python

복사편집

# 군집별로 명확히 대응 관계를 확인하고 각 군집의 재커드 점수를 계산

jaccard\_0 = jaccard\_score(data['label'] == 0, data['mapped\_cluster'] == 0)

jaccard\_1 = jaccard\_score(data['label'] == 1, data['mapped\_cluster'] == 1)

average\_jaccard = (jaccard\_0 + jaccard\_1) / 2

이 방식은 군집별로 세부적인 평가를 진행하고, 평균값을 통해 전체 평가 점수를 제공합니다.

jaccard\_score(data['label'], data['mapped\_cluster'])는 기본적으로 두 이진 배열의 **1값**에 대해 Jaccard 계수를 계산합니다.

그러나 label과 mapped\_cluster가 \*\*2개 이상의 클래스 (multi-class)\*\*로 구성되어 있으면, **multi-class Jaccard 계수**를 계산합니다. 이 경우, average 매개변수의 값에 따라 계산 방법이 달라집니다.

### 1. average='binary' (기본값)

average='binary'는 **이진 분류**의 경우에만 사용 가능하며, 1값이 있는 위치를 기준으로 계산합니다.

* 각 요소에서 A=1A = 1A=1과 B=1B = 1B=1인 교집합 (∣A∩B∣|A \cap B|∣A∩B∣).
* A=1A = 1A=1 또는 B=1B = 1B=1인 합집합 (∣A∪B∣|A \cup B|∣A∪B∣).
* 공식: J=∣A∩B∣∣A∪B∣J = \frac{|A \cap B|}{|A \cup B|}J=∣A∪B∣∣A∩B∣​

하지만 label과 mapped\_cluster가 다중 클래스이면, average='binary'는 사용할 수 없습니다.

### 2. average='macro' (기본)

jaccard\_score의 기본 average는 'macro'입니다. 각 클래스에 대해 Jaccard 계수를 계산한 다음, **모든 클래스의 평균**을 구합니다.

**계산 과정**:

1. **클래스별 Jaccard 계수 계산**:
   * 각 클래스 ccc에 대해 AAA와 BBB를 이진 배열로 변환:
     + A[i]=1A[i] = 1A[i]=1 if label[i]=clabel[i] = clabel[i]=c, otherwise 0.
     + B[i]=1B[i] = 1B[i]=1 if mapped\_cluster[i]=cmapped\\_cluster[i] = cmapped\_cluster[i]=c, otherwise 0.
   * 클래스 ccc에 대한 Jaccard 계수: Jc=∣A∩B∣∣A∪B∣J\_c = \frac{|A \cap B|}{|A \cup B|}Jc​=∣A∪B∣∣A∩B∣​
2. **모든 클래스 평균 계산**:

Jmacro=1C∑c=1CJcJ\_{macro} = \frac{1}{C} \sum\_{c=1}^C J\_cJmacro​=C1​c=1∑C​Jc​

### 3. average='micro'

micro 평균은 모든 클래스에 대해 교집합과 합집합을 **전체적으로 합산**한 후 계산합니다.

**계산 과정**:

1. **모든 클래스의 교집합 합**:

intersection=∑i=1n1[label[i]=mapped\_cluster[i]≠−1]\text{intersection} = \sum\_{i=1}^n \mathbb{1}[label[i] = mapped\\_cluster[i] \neq -1]intersection=i=1∑n​1[label[i]=mapped\_cluster[i]=−1]

(여기서 −1-1−1은 클러스터링에서 일반적으로 **노이즈**로 간주됩니다.)

1. **모든 클래스의 합집합 합**:

union=∑i=1n1[label[i]≠−1 or mapped\_cluster[i]≠−1]\text{union} = \sum\_{i=1}^n \mathbb{1}[label[i] \neq -1 \text{ or } mapped\\_cluster[i] \neq -1]union=i=1∑n​1[label[i]=−1 or mapped\_cluster[i]=−1]

1. **최종 Jaccard 점수**:

Jmicro=intersectionunionJ\_{micro} = \frac{\text{intersection}}{\text{union}}Jmicro​=unionintersection​

### 예시

#### 데이터

python

복사편집

label = [0, 0, 1, 1, 0, 1]

mapped\_cluster = [0, 1, 1, 1, 0, 0]

#### 1. average='macro'

클래스 0과 클래스 1에 대해 개별적으로 계산:

* 클래스 0 (A=[1,1,0,0,1,0],B=[1,0,0,0,1,0]A = [1, 1, 0, 0, 1, 0], B = [1, 0, 0, 0, 1, 0]A=[1,1,0,0,1,0],B=[1,0,0,0,1,0]): J0=∣A∩B∣∣A∪B∣=23J\_0 = \frac{|A \cap B|}{|A \cup B|} = \frac{2}{3}J0​=∣A∪B∣∣A∩B∣​=32​
* 클래스 1 (A=[0,0,1,1,0,1],B=[0,1,1,1,0,0]A = [0, 0, 1, 1, 0, 1], B = [0, 1, 1, 1, 0, 0]A=[0,0,1,1,0,1],B=[0,1,1,1,0,0]): J1=∣A∩B∣∣A∪B∣=25J\_1 = \frac{|A \cap B|}{|A \cup B|} = \frac{2}{5}J1​=∣A∪B∣∣A∩B∣​=52​

평균:

Jmacro=J0+J12=23+252≈0.53J\_{macro} = \frac{J\_0 + J\_1}{2} = \frac{\frac{2}{3} + \frac{2}{5}}{2} \approx 0.53Jmacro​=2J0​+J1​​=232​+52​​≈0.53

#### 2. average='micro'

* 교집합 (∣A∩B∣|A \cap B|∣A∩B∣): 1+1+1+0+1+0=31 + 1 + 1 + 0 + 1 + 0 = 31+1+1+0+1+0=3
* 합집합 (∣A∪B∣|A \cup B|∣A∪B∣): 1+1+1+1+1+1=61 + 1 + 1 + 1 + 1 + 1 = 61+1+1+1+1+1=6

Jmicro=36=0.5J\_{micro} = \frac{3}{6} = 0.5Jmicro​=63​=0.5

### 결론

* \*\*jaccard\_score(data['label'], data['mapped\_cluster'])\*\*는 다중 클래스 전체의 유사성을 계산할 때 사용됩니다.
* **계산 방식**은 average 매개변수에 따라 달라지며, 일반적으로 macro나 micro로 지정됩니다.
* 군집별로 개별 Jaccard 계수를 계산하고 싶다면 binary 방식처럼 각 클래스별로 비교를 해야 합니다.