|  |  |  |  |
| --- | --- | --- | --- |
| Datasets | Max Norm  (Yang et al.) | Max Norm  (Python/Rowan) | L21 Norm |
| Using The Embedded filter | | | |
| Leukemia (k=2) | 100 ± 0.00 | **0.98 ± 0.00** | 0.86 ± 0.02 |
| Leukemia (K=3) | 100 ± 0.00 | **0.98 ± 0.00** | 0.94 ± 0.00 |
| CNS | 96.88 ± 0.07 | 0.91 ± 0.01 | 0.80 ± 0.00 |
| Medulloblastoma | 61.76 ± 0 | 0.61 ± 0.00 | **0.62 ± 0.00** |
| GSE4913 | 62.48 ± 0.28 | 0.57 ± 0.00  Threshold= 0.5 | **0.60 ± 0.00 (T=0.5)** |
| GSE35896 | 88.00 ± 0.32 | **0.79 ± 0.00**  Threshold= 0.9 | 0.61 ± 0.01 (T=0.5)   * 1. 0.02 (T=0.9) |

|  |  |  |  |
| --- | --- | --- | --- |
| Datasets  **With noise\*** | Max Norm  (Yang et al.) | Max Norm  (Python/Rowan) | L21 Norm |
| Using The Embedded filter | | | |
| Leukemia (k=2) |  | 0.91 ± 0.00 | 0.95 ± 0.00 |
| Leukemia (K=3) |  | 0.89 ± 0.01 | 0.95 ± 0.00 |
| CNS |  | 0.77 ± 0.01 | 0.81 ± 0.01 |
| Medulloblastoma |  | 0.61 ± 0.00 | 0.62 ± 0.00 |

**NMF problem in Molecular Cancer Class Discovery:**

* Decomposition is not unique allowing different clustering results to be obtained.

Various normalisation techniques can be used to help alleviate this problem. Normalisation is a preprocessing technique which can scale data to a range of [0, 1], it can reduce the scale difference between different features, boosting performance.

**L21 Norm:**

L21 norm calculates the sum of the L2 norms of the columns, basically measuring the reconstruction error for each sample and then summing them. L21 doesn’t square the error for each data point making it more resilient to outliers and noises than the standard NMF

Hypothesis:

May be most effective if the dataset contains sample specific outliers

**Controlled uniform noise injection**

A scaling factor is used which controls the maximum amplitude of the noise, it takes the largest data entry in the matrix and sets the maximum at a set fraction of that value. For example, if the fraction is set at 0.05 and the max value is 1000, then the lambda is 50.

Questions:

Why can’t I recreate the same accuracies as in original paper?

What to use with L21 norm e.g. what filters, preprocessing etc.

What datasets to test it on?

Approach for organising code:

You've hit on a common concern with object-oriented programming: it seems like you're writing a lot of repetitive code. However, the class-based approach is designed to solve this exact problem through a principle called **inheritance**.

**How Inheritance Helps**

Instead of writing a new class from scratch for each NMF variant (L21, KL, Frobenius), you can create a single **base class** that contains all the common functionality, and then create specific subclasses for each algorithm.

1. **Create a Base Class**: You'd make a class called BaseNMF that contains all the functions that are the same for all NMF types.
   * \_\_init\_\_: For setting common parameters like n\_components, n\_iter, seed.
   * initialise\_matrices(): For randomly initializing W and H.
   * fit(): For running the iterative process.
   * cluster\_assignment(): For assigning samples to clusters.
2. **Create Subclasses**: You would then create specific classes that **inherit** from BaseNMF. These subclasses only need to implement the parts that are unique to them.
   * L21normNMF(BaseNMF): This class would override the compute() method to implement the L21-specific update rules.
   * KLNMF(BaseNMF): This class would override the compute() method to implement the KL-specific update rules.

This way, you're not writing the same code over and over. You're only writing the unique parts for each algorithm, while reusing the shared logic from the base class.

**Why This Is Better for You**

This approach is perfect for your situation because you're comparing multiple algorithms that share a common structure. By using inheritance, you can:

* **Avoid Code Duplication**: All the common setup and evaluation logic is in one place.
* **Easily Add New Algorithms**: If you want to try another NMF variant, you just create a new subclass and implement its unique update rules.
* **Ensure Consistency**: All your algorithms will share the same fit and evaluation methods, guaranteeing that you're comparing them fairly.

What this looks like: A great way to visualize this class-based approach with inheritance is to think of a family tree or a blueprint. You have a general blueprint for NMF, and then you create specific, specialized blueprints for each version.

**The Blueprint: BaseNMF**

First, you'd create a base class that holds all the shared functionality. This is your general **BaseNMF** blueprint. It defines what every NMF model should have in common.

* **Attributes**: n\_components, n\_iter, seed
* **Methods**: initialise\_matrices() and fit(). The fit() method is the main loop that calls the compute() method. It's the same for every algorithm.

Python

class BaseNMF:

def \_\_init\_\_(self, n\_components, n\_iter, seed):

self.k = n\_components

self.n\_iter = n\_iter

self.seed = seed

self.W = None

self.H = None

def initialise\_matrices(self, A):

# Common initialization logic

self.g, self.n = A.shape

rng = np.random.RandomState(self.seed)

self.W = rng.rand(self.g, self.k)

self.H = rng.rand(self.k, self.n)

def fit(self, A):

self.initialise\_matrices(A)

for step in range(self.n\_iter):

self.W, self.H = self.compute(A, self.W, self.H) # Calls the specific compute method

return self.W, self.H

def compute(self, A, W, H):

# This method is designed to be overridden by subclasses

raise NotImplementedError("Subclass must implement the compute method.")

**The Specialized Blueprints: Subclasses**

Next, you create specialized blueprints that inherit from BaseNMF. They automatically get all the common methods and attributes, and you only need to write the code that makes them unique—in this case, the compute() method with the specific update rules.

**1. L21 NMF Subclass**

The L21normNMF subclass inherits all the functionality from BaseNMF and just provides its own compute() method.

Python

class L21normNMF(BaseNMF):

def compute(self, A, W, H):

# L21-specific update rules

D = np.diag(1 / np.sqrt(np.sum(np.square(A - W @ H), axis=0)))

Wu = W \* (A @ D @ H.T) / (W @ H @ D @ H.T + 1e-10)

Hu = H \* (Wu.T @ A @ D) / (Wu.T @ Wu @ H @ D + 1e-10)

return Wu, Hu

**2. KL NMF Subclass**

Similarly, the KLNMF subclass inherits from BaseNMF and provides its own compute() method with KL-specific updates.

Python

class KLNMF(BaseNMF):

def compute(self, A, W, H):

# KL-specific update rules

# You could also add normalization here if needed

Wu = W \* (A / (W @ H + 1e-10)) @ H.T / (np.sum(H, axis=1) + 1e-10)

Hu = H \* (W.T @ (A / (W @ H + 1e-10))) / (W.T @ np.ones(W.shape) + 1e-10)

return Wu, Hu

**The Main Run Script**

Your main script would then be very clean and simple. You would just create instances of these specific classes and run their fit methods.

Python

if \_\_name\_\_ == '\_\_main\_\_':

# Create an L21 NMF model

l21\_model = L21normNMF(n\_components=2, n\_iter=500, seed=23)

W\_l21, H\_l21 = l21\_model.fit(A)

# Create a KL NMF model

kl\_model = KLNMF(n\_components=2, n\_iter=500, seed=23)

W\_kl, H\_kl = kl\_model.fit(A)

# Now you can compare the results

# ...