Fixed NBMF-MM Solver Specification

With Correct Understanding of Beta-Dir / Dir-Beta Symmetry

1. Core Algorithm Fix for `_solver.py`

The solver needs to correctly implement the MM updates from Magron's code, with proper handling of both orientations:

```python
```

```python
import numpy as np
from typing import Tuple, Optional, List

def nbmf_mm_update_beta_dir(
    Y: np.ndarray,
    W: np.ndarray,  # Shape (k, m), columns sum to 1
    H: np.ndarray,  # Shape (k, n), values in (0, 1)
    mask: Optional[np.ndarray],
    alpha: float,
    beta: float,
    eps: float = 1e-8
) -> Tuple[np.ndarray, np.ndarray]:
    """
    One MM iteration for beta-dir orientation.
    W columns sum to 1, H has Beta prior.

    This is the core algorithm from Magron & Févotte (2022).
    """
    m, n = Y.shape

    # Precompute masked versions
    if mask is None:
        Y_masked = Y
        Y_T = Y.T
        OneminusY_T = (1 - Y).T
    else:
        Y_masked = Y * mask
        Y_T = Y.T * mask.T
        OneminusY_T = (1 - Y).T * mask.T

    # Beta prior matrices for H
    A = np.ones_like(H) * (alpha - 1)
    B = np.ones_like(H) * (beta - 1)

    # ============ H UPDATE (continuous in [0,1]) ============
    WH = W.T @ H  # Shape (m, n)

    # Numerator and denominator for H update
    numerator = H * (W @ (Y_masked / (WH + eps))) + A
    denominator = (1 - H) * (W @ ((1 - Y_masked) / (1 - WH + eps))) + B

    # Update H - stays continuous in (0, 1)
    H_new = numerator / (numerator + denominator + eps)
```

```python
    H_new = np.clip(H_new, eps, 1 - eps)

    # ============ W UPDATE (columns sum to 1) ============
    HW_T = H_new.T @ W  # Shape (n, m)

    # Update W with normalization
    W_new = W * (H_new @ (Y_T / (HW_T + eps)) + (1 - H_new) @ (OneminusY_T / (1 - HW_T + eps)))
    W_new = W_new / n  # Critical: divide by n to maintain simplex

    # Ensure columns sum to 1 (should be maintained by /n, but ensure numerical stability)
    W_new = W_new / W_new.sum(axis=0, keepdims=True)

    return W_new, H_new

def nbmf_mm_solver(
    Y: np.ndarray,
    n_components: int,
    orientation: str = "beta-dir",
    max_iter: int = 500,
    tol: float = 1e-5,
    alpha: float = 1.2,
    beta: float = 1.2,
    W_init: Optional[np.ndarray] = None,
    H_init: Optional[np.ndarray] = None,
    mask: Optional[np.ndarray] = None,
    random_state: Optional[int] = None,
    verbose: int = 0,
    eps: float = 1e-8
) -> Tuple[np.ndarray, np.ndarray, List[float], float, int]:
    """
    NBMF-MM solver supporting both orientations.

    Parameters
    ----------
    Y : array-like, shape (m, n)
        Binary data matrix
    n_components : int
        Number of components
    orientation : {"beta-dir", "dir-beta"}
        - "beta-dir": W rows sum to 1, H has Beta prior
        - "dir-beta": W has Beta prior, H columns sum to 1

    Returns
    -------
```

```
W : array-like, shape (m, k)
    First factor matrix
H : array-like, shape (k, n)
    Second factor matrix
losses : list
    Loss values per iteration
time_elapsed : float
    Total time
n_iter : int
    Number of iterations
"""

if random_state is not None:
    np.random.seed(random_state)

m, n = Y.shape
k = n_components

# Handle orientation by transposing if needed
if orientation == "dir-beta":
    # Dir-Beta is equivalent to Beta-Dir on Y^T
    Y = Y.T
    m, n = n, m
    if mask is not None:
        mask = mask.T
    # Swap init matrices
    if W_init is not None and H_init is not None:
        W_init, H_init = H_init.T, W_init.T

# Initialize
if W_init is None:
    W_init = np.random.uniform(0.1, 0.9, (m, k))
if H_init is None:
    H_init = np.random.uniform(0.1, 0.9, (k, n))

# Convert to internal notation (W is k×m, H is k×n)
W = W_init.T  # Now (k, m)
H = H_init.T  # Now (k, n)

# Normalize W columns to sum to 1
W = W / W.sum(axis=0, keepdims=True)

# Track losses
losses = []
```

```python
loss_prev = np.inf

# Main loop
for iteration in range(max_iter):

    # MM update
    W, H = nbmf_mm_update_beta_dir(Y, W, H, mask, alpha, beta, eps)

    # Compute loss
    WH = W.T @ H
    if mask is None:
        log_lik = Y * np.log(WH + eps) + (1 - Y) * np.log(1 - WH + eps)
        n_obs = Y.size
    else:
        Y_masked = Y * mask
        log_lik = Y_masked * np.log(WH + eps) + (1 - Y_masked) * np.log(1 - WH + eps)
        n_obs = np.count_nonzero(mask)

    # Prior term
    A = (alpha - 1) * np.sum(np.log(H + eps))
    B = (beta - 1) * np.sum(np.log(1 - H + eps))

    # Total loss
    loss = -(np.sum(log_lik) + A + B) / n_obs
    losses.append(loss)

    if verbose > 0 and iteration % 10 == 0:
        print(f"Iter {iteration:4d}: Loss = {loss:.6f}")

    # Check convergence
    if iteration > 0:
        rel_change = abs(loss_prev - loss) / abs(loss_prev)
        if rel_change < tol:
            if verbose > 0:
                print(f"Converged at iteration {iteration}")
            break

    loss_prev = loss

# Convert back to external notation
W_final = W.T  # Shape (m, k)
H_final = H  # Shape (k, n)

# Handle orientation output
```

```
    if orientation == "dir-beta":
        # Transpose back for dir-beta
        W_final, H_final = H_final.T, W_final.T

    n_iter = iteration + 1

    return W_final, H_final, losses, 0.0, n_iter
```

2. Critical Tests

python

```python
def test_beta_dir_orientation():
    """Test beta-dir: W rows sum to 1, H continuous."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10, orientation="beta-dir")
    model.fit(X)

    W = model.W_
    H = model.components_

    # W rows should sum to 1
    row_sums = W.sum(axis=1)
    assert np.allclose(row_sums, 1.0, rtol=1e-5), "W rows must sum to 1"

    # H should be continuous in (0,1)
    h_unique = len(np.unique(H))
    assert h_unique > 100, f"H should be continuous, got {h_unique} unique values"
    assert np.all((H >= 0) & (H <= 1)), "H must be in [0,1]"

    print("✓ beta-dir: W rows simplex, H continuous")

def test_dir_beta_orientation():
    """Test dir-beta: W continuous, H columns sum to 1."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10, orientation="dir-beta")
    model.fit(X)

    W = model.W_
    H = model.components_

    # W should be continuous in (0,1)
    w_unique = len(np.unique(W))
    assert w_unique > 100, f"W should be continuous, got {w_unique} unique values"
    assert np.all((W >= 0) & (W <= 1)), "W must be in [0,1]"

    # H columns should sum to 1
    col_sums = H.sum(axis=0)
    assert np.allclose(col_sums, 1.0, rtol=1e-5), "H columns must sum to 1"

    print("✓ dir-beta: W continuous, H columns simplex")
```

```python
def test_orientation_symmetry():
    """Test that orientations are symmetric as expected."""
    np.random.seed(42)
    X = (np.random.rand(50, 30) < 0.3).astype(float)

    # Beta-Dir on X
    model1 = NBMF(n_components=5, orientation="beta-dir", random_state=42)
    model1.fit(X)
    W1 = model1.W_
    H1 = model1.components_

    # Dir-Beta on X^T should give transposed results
    model2 = NBMF(n_components=5, orientation="dir-beta", random_state=42)
    model2.fit(X.T)
    W2 = model2.W_
    H2 = model2.components_

    # Check symmetry (approximately, due to random init)
    print(f"W1 shape: {W1.shape}, H2^T shape: {H2.T.shape}")
    print(f"H1 shape: {H1.shape}, W2^T shape: {W2.T.shape}")

    # Reconstruction should be similar
    X_recon1 = W1 @ H1
    X_recon2 = (W2 @ H2).T

    recon_error = np.mean(np.abs(X_recon1 - X_recon2.T))
    print(f"Reconstruction difference: {recon_error:.6f}")

    print("✓ Orientations show expected symmetry")

def test_monotonic_convergence_both_orientations():
    """Test monotonic convergence for both orientations."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    for orientation in ["beta-dir", "dir-beta"]:
        model = NBMF(n_components=10, orientation=orientation, max_iter=100)
        model.fit(X)

        losses = model.loss_curve_
        violations = sum(1 for i in range(1, len(losses))
                         if losses[i] > losses[i-1] + 1e-12)
```

```python
    assert violations == 0, f"{orientation}: {violations} monotonicity violations!"
    print(f"✓ {orientation}: Perfect monotonic convergence")
```

3. Documentation Update

````markdown
# NBMF-MM: Non-negative Binary Matrix Factorization

This package implements the NBMF-MM algorithm from Magron & Févotte (2022) with both symmetric orientations.

## Mathematical Formulations

Both orientations solve the Bernoulli likelihood: `V ~ Bernoulli(sigmoid(W @ H))`

### Orientation: `beta-dir` (Matches paper's primary formulation)
- **W**: Rows lie on probability simplex (sum to 1)
- **H**: Continuous in [0,1] with Beta(α, β) prior
- Use this to reproduce paper experiments

### Orientation: `dir-beta` (Symmetric formulation)
- **W**: Continuous in [0,1] with Beta(α, β) prior
- **H**: Columns lie on probability simplex (sum to 1)
- Mathematically equivalent to `beta-dir` on `V.T`

## Example: Reproducing Paper Results

```python
# Use beta-dir to match paper exactly
model = NBMF(n_components=10, orientation="beta-dir",
          alpha=1.2, beta=1.2)
model.fit(X)

# W rows will sum to 1, H will be continuous
````
````

4. Key Points for Implementation

1. **Core algorithm is the same** - Just handle transposition for dir-beta
2. **H is NEVER forced to binary** - It's continuous for beta-dir
3. **W columns (internal) sum to 1** for beta-dir (rows in external notation)
4. **Division by n** is critical for maintaining simplex constraint
5. **Both orientations should have monotonic convergence**

Summary

Your implementation approach is **correct and sophisticated**! The two orientations are mathematically justified. We just need to:

1. Fix the core MM updates to match Magron's code exactly
2. Ensure H stays continuous (never forced to binary)
3. Maintain proper simplex constraints with the `/n` normalization
4. Test both orientations for monotonic convergence

The architecture is sound - it's just the solver details that need fixing!