NBMF-MM Solver Implementation Specification

Based on Magron & Févotte (2022) Reference Implementation

🚨 CRITICAL UNDERSTANDING

The KEY misunderstanding we had:

- **H is NOT binary during optimization!** It's continuous in [0,1]
- **W rows sum to 1** (simplex constraint)
- The "binary" in NBMF refers to the **binary input data Y**, not the factors!
- H becomes "effectively binary" at convergence due to the Beta prior, but is NOT forced to {0,1} during updates

1. Mathematical Formulation

Objective Function

Minimize negative log-likelihood with Beta prior:

```
L = -∑[Y * log(Θ) + (1-Y) * log(1-Θ)] - ∑[(α-1) * log(H) + (β-1) * log(1-H)]
```

where:

- `Y ∈ {0,1}^(m×n)` is binary data
- `Θ = W^T @ H ∈ (0,1)^(m×n)` is the Bernoulli parameter
- `W ∈ ℝ+^(k×m)` with columns summing to 1 (probability simplex)
- `H ∈ (0,1)^(k×n)` is continuous (NOT binary during optimization!)
- `α, β` are Beta prior parameters for H

2. Core Algorithm Implementation

File: `src/nbmf_mm/_solver.py`

```python
```

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

def nbmf_mm_solver(
    Y: np.ndarray,
    n_components: int,
    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 implementation following Magron & Févotte (2022).

    CRITICAL: H is continuous in [0,1], NOT forced to binary!

    Parameters
    ----------
    Y : array-like, shape (m, n)
        Binary data matrix {0, 1}
    n_components : int
        Number of latent components k
    max_iter : int
        Maximum iterations
    tol : float
        Convergence tolerance on relative loss change
    alpha, beta : float
        Beta prior parameters for H
    W_init, H_init : array-like, optional
        Initial matrices
    mask : array-like, optional
        Binary mask for observed entries
    random_state : int, optional
        Random seed
    verbose : int
        Verbosity level
```

```
eps : float
    Small constant for numerical stability

Returns
-------
W : array-like, shape (m, k)
    Factor matrix with rows on simplex
H : array-like, shape (k, n)
    Factor matrix in (0, 1)
losses : list
    Loss values per iteration
time_elapsed : float
    Total computation time
n_iter : int
    Number of iterations run
"""

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

# Get dimensions
m, n = Y.shape
k = n_components

# Initialize mask if not provided
if mask is None:
    mask = np.ones_like(Y)

# Initialize factors
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))

# CRITICAL: Transpose to match Magron's notation
# In Magron's code: W is (k, m) and 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 (simplex constraint)
W = W / W.sum(axis=0, keepdims=True)

# Precompute masked versions for efficiency
```

```
Y_masked = Y * mask
Y_T = Y.T * mask.T  # Transposed masked Y
OneminusY_T = (1 - Y.T) * mask.T  # Transposed masked (1-Y)

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

# Initialize tracking
losses = []
loss_prev = np.inf
start_time = time.time()

# Main optimization loop
for iteration in range(max_iter):

    # ============ H UPDATE ============
    # CRITICAL: H is NOT forced to binary!
    # H stays continuous in (0, 1)

    # Compute W^T @ H
    WH = W.T @ H  # Shape (m, n)

    # Numerator: H * W @ (Y / (WH + eps)) + A
    numerator = H * (W @ (Y_masked / (WH + eps))) + A

    # Denominator: (1 - H) * W @ ((1 - Y) / (1 - WH + eps)) + B
    denominator = (1 - H) * (W @ ((1 - Y_masked) / (1 - WH + eps))) + B

    # Update H (stays in (0, 1) naturally)
    H = numerator / (numerator + denominator + eps)

    # Clip to avoid numerical issues at boundaries
    H = np.clip(H, eps, 1 - eps)

    # ============ W UPDATE ============
    # W columns must stay on simplex

    # Compute H @ W^T (for the transpose computation)
    HW_T = H.T @ W  # Shape (n, m)

    # Update W with normalization by n (maintains simplex)
    W_numerator = W * (H @ (Y_T / (HW_T + eps)) + (1 - H) @ (OneminusY_T / (1 - HW_T + eps)))
    W = W_numerator / n
```

```
        # Ensure W stays normalized (columns sum to 1)
        # This should be maintained by the /n term, but we ensure it for numerical stability
        W = W / W.sum(axis=0, keepdims=True)

        # ============ COMPUTE LOSS ============
        WH = W.T @ H  # Recompute after updates

        # Log-likelihood term
        log_lik = Y_masked * np.log(WH + eps) + (1 - Y_masked) * np.log(1 - WH + eps)

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

        # Total loss (negative log posterior)
        loss = -(np.sum(log_lik) + np.sum(prior)) / np.count_nonzero(mask)
        losses.append(loss)

        if verbose > 0 and iteration % 10 == 0:
            print(f"Iteration {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} (rel_change: {rel_change:.2e})")
                break

        loss_prev = loss

    # Transpose back to our convention
    W_final = W.T  # Shape (m, k)
    H_final = H  # Shape (k, n)

    time_elapsed = time.time() - start_time
    n_iter = iteration + 1

    return W_final, H_final, losses, time_elapsed, n_iter
```

3. Integration with NBMF Class

File: `src/nbmf_mm/nbmf.py` (updates needed)

python

```python
```

```python
from ._solver import nbmf_mm_solver

class NBMF(BaseEstimator, TransformerMixin):
    """
    Non-negative Binary Matrix Factorization via Majorization-Minimization.

    IMPORTANT: Despite the name "binary", the factor H is continuous in [0,1]
    during optimization. The "binary" refers to the input data Y.
    """

    def fit(self, X, y=None, mask=None):
        """Fit NBMF model to binary data X."""

        # Validate input
        X = check_array(X, accept_sparse='csr', dtype=np.float64)

        # Check if data is binary or in [0,1]
        if not np.all((X >= 0) & (X <= 1)):
            raise ValueError("X must be in [0,1]")

        # Handle sparse matrices
        if sparse.issparse(X):
            X = X.toarray()

        # Call the solver with paper-correct implementation
        W, H, losses, time_elapsed, n_iter = nbmf_mm_solver(
            Y=X,
            n_components=self.n_components,
            max_iter=self.max_iter,
            tol=self.tol,
            alpha=self.alpha,
            beta=self.beta,
            W_init=self.W_init,
            H_init=self.H_init,
            mask=mask,
            random_state=self.random_state,
            verbose=self.verbose
        )

        # Store results
        self.W_ = W  # Shape (n_samples, n_components), rows sum to 1
        self.components_ = H  # Shape (n_components, n_features), values in (0,1)
        self.loss_curve_ = losses
```

```
        self.n_iter_ = n_iter
        self.reconstruction_err_ = losses[-1] if losses else np.inf

        return self

    def transform(self, X, mask=None):
        """Transform X by finding W given fixed H."""
        check_is_fitted(self, ['components_'])
        X = check_array(X, accept_sparse='csr', dtype=np.float64)

        if sparse.issparse(X):
            X = X.toarray()

        m = X.shape[0]
        k = self.n_components
        H = self.components_

        # Initialize W randomly
        W = np.random.uniform(0.1, 0.9, (m, k))

        # Run a few iterations to find W given fixed H
        for _ in range(50):
            # Same W update as in fit, but with fixed H
            W_T = W.T
            HW_T = H.T @ W_T

            if mask is None:
                Y_T = X.T
                OneminusY_T = (1 - X).T
            else:
                Y_T = X.T * mask.T
                OneminusY_T = (1 - X).T * mask.T

            W_T = W_T * (H @ (Y_T / (HW_T + 1e-8)) + (1 - H) @ (OneminusY_T / (1 - HW_T + 1e-8)))
            W_T = W_T / X.shape[1]
            W_T = W_T / W_T.sum(axis=0, keepdims=True)
            W = W_T.T

        return W

    def inverse_transform(self, W):
        """Transform W back to data space."""
        check_is_fitted(self, ['components_'])
        W = check_array(W, dtype=np.float64)
```

```
    # Compute reconstruction
    # Note: W has rows summing to 1, H is in (0,1)
    return W @ self.components_  # Returns probabilities in (0,1)
```

4. Critical Tests to Verify Correctness

File: `tests/test_algorithm_correctness.py`

```
python
```

```python
import numpy as np
import pytest
from nbmf_mm import NBMF

def test_h_continuous_not_binary():
    """Verify H stays continuous, NOT binary during optimization."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10, max_iter=50)
    model.fit(X)

    H = model.components_

    # H should be continuous in (0, 1), NOT binary
    unique_values = np.unique(H)
    assert len(unique_values) > 2, "H should be continuous, not binary!"
    assert np.all((H >= 0) & (H <= 1)), "H should be in [0, 1]"

    # Check that H has many distinct values (continuous)
    assert len(unique_values) > 100, f"H has only {len(unique_values)} unique values, should be continuous"

    print(f"✓ H is continuous with {len(unique_values)} unique values")

def test_w_simplex_constraint():
    """Verify W rows sum to 1 (simplex constraint)."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10, max_iter=50)
    model.fit(X)

    W = model.W_
    row_sums = W.sum(axis=1)

    np.testing.assert_allclose(row_sums, 1.0, rtol=1e-5,
                               err_msg="W rows must sum to 1 (simplex constraint)")

    print(f"✓ W rows sum to 1: min={row_sums.min():.6f}, max={row_sums.max():.6f}")

def test_monotonic_convergence():
    """Test that loss decreases monotonically (MM property)."""
    np.random.seed(42)
```

```python
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10, max_iter=100, tol=1e-8)
    model.fit(X)

    losses = model.loss_curve_

    # Check strict monotonicity
    violations = []
    for i in range(1, len(losses)):
        if losses[i] > losses[i-1] + 1e-12:
            violations.append(i)
            print(f"  Violation at iter {i}: {losses[i-1]:.10f} -> {losses[i]:.10f}")

    assert len(violations) == 0, f"Found {len(violations)} monotonicity violations!"

    print(f"✓ Perfect monotonic convergence over {len(losses)} iterations")

def test_reconstruction_probabilities():
    """Test that reconstruction gives valid probabilities."""
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)

    model = NBMF(n_components=10)
    model.fit(X)

    X_reconstructed = model.inverse_transform(model.W_)

    # Should be probabilities in (0, 1)
    assert np.all((X_reconstructed >= 0) & (X_reconstructed <= 1)), \
        "Reconstructed values should be probabilities in [0, 1]"

    # Should NOT be binary
    unique_recon = np.unique(X_reconstructed)
    assert len(unique_recon) > 100, \
        f"Reconstruction should be continuous probabilities, got {len(unique_recon)} unique values"

    print(f"✓ Reconstruction gives continuous probabilities")

def test_beta_prior_effect():
    """Test that Beta prior parameters affect the solution."""
    np.random.seed(42)
    X = (np.random.rand(50, 30) < 0.3).astype(float)
```

```
# Model with symmetric prior (no preference)
model1 = NBMF(n_components=5, alpha=1.0, beta=1.0, max_iter=100)
model1.fit(X)
H1 = model1.components_

# Model with prior favoring values near 0
model2 = NBMF(n_components=5, alpha=0.5, beta=2.0, max_iter=100, random_state=42)
model2.fit(X)
H2 = model2.components_

# Model with prior favoring values near 1
model3 = NBMF(n_components=5, alpha=2.0, beta=0.5, max_iter=100, random_state=42)
model3.fit(X)
H3 = model3.components_

# Check that priors have expected effect on H
assert H2.mean() < H1.mean(), "Beta(0.5, 2) should push H toward 0"
assert H3.mean() > H1.mean(), "Beta(2, 0.5) should push H toward 1"

print(f"✓ Beta prior affects solution: H means = {H1.mean():.3f}, {H2.mean():.3f}, {H3.mean():.3f}")
```

5. Key Implementation Notes

CRITICAL POINTS:

1. **H is CONTINUOUS in [0,1]**, not binary during optimization
2. **W rows sum to 1** (probability simplex)
3. **Transpose convention**: Internally use Magron's notation (W is k×m, H is k×n), then transpose for sklearn API
4. **Normalization by n**: The W update includes division by n_features to maintain simplex
5. **Numerical stability**: Clip H to [eps, 1-eps] to avoid log(0)

What We Were Doing Wrong:

- ❌ Forcing H to be binary {0,1} after each update
- ❌ Not properly maintaining W simplex constraint
- ❌ Using wrong update equations

What Magron Does Right:

- ✅ H stays continuous in (0,1)

- ✅ W normalized to simplex via /n term
- ✅ Proper MM updates that guarantee monotonicity
- ✅ Numerical stability with eps additions

6. Validation Script

Create `examples/validate_magron_implementation.py`:

```python
```

```python
#!/usr/bin/env python3
"""Validate our implementation matches Magron's behavior."""

import numpy as np
from nbmf_mm import NBMF
import matplotlib.pyplot as plt

def main():
    print("="*60)
    print("VALIDATING NBMF-MM IMPLEMENTATION")
    print("="*60)

    # Generate test data
    np.random.seed(42)
    X = (np.random.rand(100, 50) < 0.3).astype(float)
    print(f"\nData shape: {X.shape}")
    print(f"Data sparsity: {X.mean():.3f}")

    # Fit model
    model = NBMF(
        n_components=10,
        alpha=1.2,
        beta=1.2,
        max_iter=200,
        tol=1e-6,
        verbose=1
    )
    model.fit(X)

    # Validate results
    print("\n" + "="*60)
    print("VALIDATION RESULTS")
    print("="*60)

    W = model.W_
    H = model.components_
    losses = model.loss_curve_

    # 1. Check H is continuous
    H_unique = np.unique(H)
    print(f"\n1. H Continuity:")
    print(f"   Unique values in H: {len(H_unique)}")
    print(f"   H range: [{H.min():.4f}, {H.max():.4f}]")
```

```
    print(f"   H mean: {H.mean():.4f}")
    is_continuous = len(H_unique) > 100
    print(f"   ✓ H is continuous" if is_continuous else "   ✗ H is not continuous!")

    # 2. Check W simplex constraint
    print(f"\n2. W Simplex Constraint:")
    row_sums = W.sum(axis=1)
    print(f"   W row sums: min={row_sums.min():.6f}, max={row_sums.max():.6f}")
    simplex_ok = np.allclose(row_sums, 1.0, rtol=1e-5)
    print(f"   ✓ W rows sum to 1" if simplex_ok else "   ✗ W simplex constraint violated!")

    # 3. Check monotonic convergence
    print(f"\n3. Monotonic Convergence:")
    violations = sum(1 for i in range(1, len(losses)) if losses[i] > losses[i-1] + 1e-12)
    print(f"   Iterations: {len(losses)}")
    print(f"   Final loss: {losses[-1]:.6f}")
    print(f"   Monotonicity violations: {violations}")
    print(f"   ✓ Perfect monotonic convergence" if violations == 0 else f"   ✗ {violations} violations!")

    # 4. Plot convergence
    plt.figure(figsize=(10, 6))
    plt.semilogy(losses, 'b-', linewidth=2)
    plt.xlabel('Iteration')
    plt.ylabel('Loss (log scale)')
    plt.title(f'NBMF-MM Convergence (Violations: {violations})')
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    plt.savefig('nbmf_convergence_validation.png')
    plt.show()

    # Overall result
    print("\n" + "="*60)
    if is_continuous and simplex_ok and violations == 0:
        print("🎉 ALL VALIDATIONS PASSED!")
        print("Implementation correctly follows Magron & Févotte (2022)")
    else:
        print("⚠️ SOME VALIDATIONS FAILED")
        print("Check implementation against reference")
    print("="*60)

if __name__ == "__main__":
    main()
```

7. Summary of Changes Needed

1. **Update `_solver.py`**: Implement the exact algorithm above with H continuous
2. **Update `nbmf.py`**: Remove any code that forces H to binary
3. **Update tests**: Test for H continuity, not binary constraint
4. **Update documentation**: Clarify that H is continuous, not binary

The key insight is that **H is continuous in [0,1]** throughout optimization. The Beta prior naturally encourages H toward 0 or 1 at convergence, making it "effectively binary" for interpretation, but it's never forced to be exactly {0,1} during the algorithm!