

Final NBMF-MM Fixes and Test Cleanup Plan

Priority 1: Fix Orientation Default (CRITICAL)

The Problem

The current implementation has the orientations backwards from the Magron & Févotte (2022) paper!

The Fix

File: `src/nbmf_mm/nbmf.py`

Change the default orientation:

```
python

class NBMF(BaseEstimator, TransformerMixin):
    def __init__(
        self,
        n_components: int = 10,
        orientation: str = "beta-dir", # CHANGE THIS TO MATCH PAPER!
        # ... rest of parameters
    ):

```

File: `README.md`

Update the documentation to reflect correct default:

```
markdown

## Orientations

- `orientation="beta-dir"` (default; Original Paper Setting)
  - H is binary {0,1}
  - W is non-negative with rows on simplex
  - This matches Magron & Févotte (2022)

- `orientation="dir-beta"` (Alternative)
  - H has columns on simplex
  - W is binary {0,1}
  - This is the symmetric formulation

```

Verification Test

python

```
def test_paper_default_orientation():
    """Verify default orientation matches Magron & Févotte 2022."""
    X = np.random.rand(50, 30)
    model = NBMF(n_components=5) # Using defaults
    model.fit(X)

    # With paper default (beta-dir):
    # - H should be binary
    # - W should be non-negative with simplex rows
    H = model.components_
    W = model.W_

    # Check H is binary
    assert np.all((H == 0) | (H == 1)), "H must be binary with default orientation"

    # Check W rows sum to 1 (simplex constraint)
    row_sums = W.sum(axis=1)
    np.testing.assert_allclose(row_sums, 1.0, rtol=1e-5)

    print("✓ Default orientation matches paper")
```

Priority 2: Clean Up Test Suite

Remove Implementation Detail Dependencies

Step 1: Identify Problem Tests

bash

```
# Find all tests importing from internal modules
grep -r "from nbmf_mm_" tests/
grep -r "import.*_mm_exact" tests/
grep -r "import.*estimator" tests/
```

Step 2: Refactor Tests to Use Public API Only

Create `tests/test_public_api.py`:

python

```
"""Test only the public API, no internal imports."""
```

```
import numpy as np
import pytest
from nbmf_mm import NBMF, NBMFMM # Only public imports!
```

```
class TestPublicAPI:
```

```
    """Test suite using only public API."""
```

```
    def test_basic_fit(self):
```

```
        """Test basic fitting works."""
```

```
        X = np.random.rand(100, 50)
```

```
        model = NBMF(n_components=10)
```

```
        model.fit(X)
```

```
        assert hasattr(model, 'W_')
```

```
        assert hasattr(model, 'components_')
```

```
        assert model.W_.shape == (100, 10)
```

```
        assert model.components_.shape == (10, 50)
```

```
    def test_transform(self):
```

```
        """Test transform method."""
```

```
        X_train = np.random.rand(100, 50)
```

```
        X_test = np.random.rand(20, 50)
```

```
        model = NBMF(n_components=10)
```

```
        model.fit(X_train)
```

```
        W_test = model.transform(X_test)
```

```
        assert W_test.shape == (20, 10)
```

```
    def test_fit_transform(self):
```

```
        """Test fit_transform method."""
```

```
        X = np.random.rand(100, 50)
```

```
        model = NBMF(n_components=10)
```

```
        W = model.fit_transform(X)
```

```
        assert W.shape == (100, 10)
```

```
        np.testing.assert_allclose(W, model.W_)
```

```
    def test_inverse_transform(self):
```

```
        """Test inverse_transform method."""
```

```
        X = np.random.rand(100, 50)
```

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

```
X_reconstructed = model.inverse_transform(model.W_)
assert X_reconstructed.shape == X.shape
assert np.all((X_reconstructed >= 0) & (X_reconstructed <= 1))
```

```
def test_score(self):
    """Test score method."""
    X = np.random.rand(100, 50)
    model = NBMF(n_components=10)
    model.fit(X)
```

```
score = model.score(X)
assert isinstance(score, float)
assert not np.isnan(score)
```

```
def test_perplexity(self):
    """Test perplexity method."""
    X = np.random.rand(100, 50)
    model = NBMF(n_components=10)
    model.fit(X)
```

```
perp = model.perplexity(X)
assert isinstance(perp, float)
assert perp > 0
```

```
def test_nbmfm_alias(self):
    """Test NBMFMM alias works."""
    X = np.random.rand(100, 50)
    model = NBMFMM(n_components=10) # Using alias
    model.fit(X)
```

```
assert hasattr(model, 'W_')
assert hasattr(model, 'components_')
```

```
def test_orientations(self):
    """Test both orientations work."""
    X = np.random.rand(100, 50)

    # Test beta-dir (paper default)
    model1 = NBMF(n_components=10, orientation="beta-dir")
    model1.fit(X)
    H1 = model1.components_
```

```
W1 = model1.W_
```

```
# H should be binary
```

```
assert np.all((H1 == 0) | (H1 == 1)), "H must be binary for beta-dir"
```

```
# W rows should sum to 1
```

```
np.testing.assert_allclose(W1.sum(axis=1), 1.0, rtol=1e-5)
```

```
# Test dir-beta (alternative)
```

```
model2 = NBMF(n_components=10, orientation="dir-beta")
```

```
model2.fit(X)
```

```
H2 = model2.components_
```

```
W2 = model2.W_
```

```
# H columns should sum to 1
```

```
np.testing.assert_allclose(H2.sum(axis=0), 1.0, rtol=1e-5)
```

```
# W should be binary
```

```
assert np.all((W2 == 0) | (W2 == 1)), "W must be binary for dir-beta"
```

```
def test_sparse_input(self):
```

```
    """Test sparse matrix input."""
```

```
    from scipy import sparse
```

```
    X_dense = np.random.rand(100, 50)
```

```
    X_sparse = sparse.csr_matrix(X_dense)
```

```
    model = NBMF(n_components=10)
```

```
    model.fit(X_sparse)
```

```
    assert hasattr(model, 'W_')
```

```
    assert hasattr(model, 'components_')
```

```
def test_masked_training(self):
```

```
    """Test masked training."""
```

```
    X = np.random.rand(100, 50)
```

```
    mask = np.random.rand(100, 50) > 0.1 # 90% observed
```

```
    model = NBMF(n_components=10)
```

```
    model.fit(X, mask=mask)
```

```
    score = model.score(X, mask=mask)
```

```
    assert isinstance(score, float)
```

```
def test_reproducibility(self):
```

```
    """Test random_state gives reproducible results."""
```

```
X = np.random.rand(100, 50)

model1 = NBMF(n_components=10, random_state=42)
model1.fit(X)

model2 = NBMF(n_components=10, random_state=42)
model2.fit(X)

np.testing.assert_allclose(model1.W_, model2.W_)
np.testing.assert_array_equal(model1.components_, model2.components_)
```

Step 3: Move Internal Tests to Separate File

Create `tests/test_internals.py` (if we must keep them):

```
python

"""Tests for internal implementation details.
These are separate from public API tests and can be skipped if internals change."""

import pytest

# Mark all tests as internal
pytestmark = pytest.mark.internal

def test_mm_exact_implementation():
    """Test the exact MM implementation if needed."""
    pytest.skip("Internal tests should be refactored to test public API")
```

Priority 3: Improve Reproduction Scripts

Fix Orientation in Reproduction Script

File: `examples/reproduce_magron2022.py`

```
python
```

CRITICAL: Use the correct orientation for paper reproduction

```
model = NBMF(  
    n_components=k,  
    orientation="beta-dir", # MUST be beta-dir for paper setting!  
    alpha=1.2, # From paper  
    beta=1.2, # From paper  
    max_iter=500,  
    tol=1e-5,  
    random_state=42,  
    verbose=0  
)
```

Add Validation Script

Create `examples/validate_implementation.py`:

```
python
```

```
#!/usr/bin/env python3
```

```
"""
```

Validate that our implementation matches the paper's mathematical specification.

```
"""
```

```
import numpy as np
```

```
from nbmf_mm import NBMF
```

```
import matplotlib.pyplot as plt
```

```
def validate_monotonicity():
```

```
    """Ensure MM algorithm has monotonic convergence."""
```

```
    print("Testing Monotonic Convergence...")
```

```
    # Generate test data
```

```
    np.random.seed(42)
```

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

```
    # Fit with paper settings
```

```
    model = NBMF(
```

```
        n_components=10,
```

```
        orientation="beta-dir", # Paper setting
```

```
        alpha=1.2,
```

```
        beta=1.2,
```

```
        max_iter=200,
```

```
        verbose=1
```

```
)
```

```
    model.fit(X)
```

```
    # Check monotonicity
```

```
    losses = model.loss_curve_
```

```
    violations = 0
```

```
    for i in range(1, len(losses)):
```

```
        if losses[i] > losses[i-1] + 1e-10:
```

```
            violations += 1
```

```
            print(f" Violation at iteration {i}: {losses[i-1]:.10f} -> {losses[i]:.10f}")
```

```
    if violations == 0:
```

```
        print("✅ Perfect monotonic convergence!")
```

```
    else:
```

```
        print(f"❌ {violations} monotonicity violations found!")
```

```
    # Plot
```

```
    plt.figure(figsize=(10, 6))
```



```

plt.semilogy(losses, 'b-', linewidth=2)
plt.xlabel('Iteration')
plt.ylabel('Negative Log-Likelihood')
plt.title(f'Monotonic Convergence Test (Violations: {violations})')
plt.grid(True, alpha=0.3)
plt.savefig('outputs/monotonicity_test.png')
plt.show()

```

```

return violations == 0

```

```

def validate_constraints():

```

```

    """Ensure constraints are satisfied."""

```

```

    print("\nTesting Constraints...")

```

```

    np.random.seed(42)

```

```

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

```

```

    # Test beta-dir (paper setting)

```

```

    print("\n1. Testing beta-dir (paper setting):")

```

```

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

```

```

    model.fit(X)

```

```

    H = model.components_

```

```

    W = model.W_

```

```

    # Check H is binary

```

```

    is_binary = np.all((H == 0) | (H == 1))

```

```

    print(f"  H is binary: {is_binary}")

```

```

    print(f"  H unique values: {np.unique(H)}")

```

```

    # Check W rows sum to 1

```

```

    row_sums = W.sum(axis=1)

```

```

    simplex_ok = np.allclose(row_sums, 1.0, rtol=1e-5)

```

```

    print(f"  W rows on simplex: {simplex_ok}")

```

```

    print(f"  W row sums range: [{row_sums.min():.6f}, {row_sums.max():.6f}]")

```

```

    # Test dir-beta (alternative)

```

```

    print("\n2. Testing dir-beta (alternative):")

```

```

    model2 = NBMF(n_components=10, orientation="dir-beta")

```

```

    model2.fit(X)

```

```

    H2 = model2.components_

```

```

    W2 = model2.W_

```

```

# Check H columns sum to 1
col_sums = H2.sum(axis=0)
simplex_ok2 = np.allclose(col_sums, 1.0, rtol=1e-5)
print(f"  H columns on simplex: {simplex_ok2}")
print(f"  H column sums range: [{col_sums.min():.6f}, {col_sums.max():.6f}]")

# Check W is binary
is_binary2 = np.all((W2 == 0) | (W2 == 1))
print(f"  W is binary: {is_binary2}")
print(f"  W unique values: {np.unique(W2)}")

return is_binary and simplex_ok and simplex_ok2 and is_binary2

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

    # Run all validations
    mono_ok = validate_monotonicity()
    const_ok = validate_constraints()

    print("\n" + "="*60)
    print("VALIDATION SUMMARY")
    print("="*60)
    print(f"Monotonic Convergence: {'✅ PASS' if mono_ok else '❌ FAIL'}")
    print(f"Constraint Satisfaction: {'✅ PASS' if const_ok else '❌ FAIL'}")

    if mono_ok and const_ok:
        print("\n🎉 All validations passed! Implementation is correct.")
    else:
        print("\n⚠️ Some validations failed. Check implementation.")

if __name__ == "__main__":
    main()

```

Priority 4: Update Documentation

Update README.md

Add a clear section about the paper reproduction:

markdown

Reproducing Magron & Févotte (2022)

To reproduce the results from the original paper, use these settings:

```
```python
from nbmf_mm import NBMF

Paper setting: H binary, W non-negative with simplex rows
model = NBMF(
 n_components=10,
 orientation="beta-dir", # THIS IS CRITICAL!
 alpha=1.2,
 beta=1.2,
 max_iter=500,
 tol=1e-5
)
model.fit(X)

H will be binary {0,1}
W will have rows that sum to 1
```

Run the reproduction scripts:

```
bash

python examples/reproduce_magron2022.py
python examples/display_figures.py
```

## ## Priority 5: Clean Up Legacy Code

### ### Decision Tree for \_mm\_exact.py

1. **\*\*If tests can be refactored to not need it\*\***: DELETE IT
2. **\*\*If it provides unique functionality\*\***: DOCUMENT why it's needed
3. **\*\*If it's just for backwards compatibility\*\***: Mark as DEPRECATED

```
```python
# src/nbmf_mm/_mm_exact.py
"""
DEPRECATED: Legacy implementation kept for backwards compatibility.
Will be removed in v1.0.0. Use NBMF class instead.
"""
import warnings

def any_legacy_function():
    warnings.warn(
        "This function is deprecated. Use NBMF class instead.",
        DeprecationWarning,
        stacklevel=2
    )
    # ... legacy code
```

Testing Checklist

After implementing these fixes:

```
bash
```

1. Run validation script

```
python examples/validate_implementation.py
```

2. Run public API tests

```
pytest tests/test_public_api.py -v
```

3. Run reproduction

```
python examples/reproduce_magron2022.py
```

4. Check orientation default

```
python -c "from nbmf_mm import NBMF; m = NBMF(); print(f'Default orientation: {m.orientation}')"
# Should print: Default orientation: beta-dir
```

5. Quick paper setting test

```
python -c """
import numpy as np
from nbmf_mm import NBMF
X = np.random.rand(50, 30)
m = NBMF(n_components=5).fit(X)
H = m.components_
print(f'H is binary: {np.all((H == 0) | (H == 1))}')
print(f'W rows sum to 1: {np.allclose(m.W_.sum(axis=1), 1.0)}')
"""
```

Should print: H is binary: True, W rows sum to 1: True

Success Criteria

✅ **Orientation:** Default is `beta-dir` matching paper (H binary, W simplex rows) ✅ **Tests:** All public API tests pass without importing internals ✅ **Monotonicity:** Zero violations in convergence ✅

Reproduction: Scripts run and produce comparable results ✅ **Documentation:** Clear explanation of paper settings

Final Note

The core algorithm is WORKING! These are mostly cosmetic/organizational fixes. The most critical issue is fixing the orientation default to match the paper. Everything else is cleanup to make the package more maintainable.