LSA methods comparison

Python

Anton Antonov
PythonForPrediction at WordPress
SimplifiedMachineLearningWorkflows-book at GitHub

December 2021
February 2022

Get Python mandalas collection

Setup

In[123]:=

```
# "Standard" packages
import pandas
import numpy
import random
import io
import time

# SSparseMatrix, SMR, and LSA packages
from SSparseMatrix import *
from SparseMatrixRecommender import *
from LatentSemanticAnalyzer import LatentSemanticAnalyzer

# Plotting packages
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

# Image processing package(s)
from PIL import Image, ImageOps

# Random mandalas packages
from RandomMandala import random_mandala, figure_to_image
```

In[124]:=

```
pythonSession = First@Pick[ExternalSessions[], #["System"] == "Python" & /@ ExternalSessions[]]
```

Out[•]=

```
ExternalSessionObject[ System: Python  Version: 3.10.2
                       Name: DefaultPythonSession ]
```

In[•]:=

```
from SSparseMatrix import *
from wolframclient.language import wl
from wolframclient.serializers import export, wolfram_encoder

# Provide definition of the deferred to_wl() method
def _my_to_wl(self):
```

```
    return wl.SSparseMatrix.ToSSparseMatrix(self.to_dict())
SSparseMatrix.to_wl = _my_to_wl

@wolfram_encoder.dispatch(SSparseMatrix)
def encode_s_sparse_matrix(serializer, ssmat):
    # encode the class as a symbol called SSparseMatrix
    return serializer.encode(ssmat.to_wl())
```

Out[ ]= ExternalFunction[System: Python Arguments: {serializer, ssmat} Command: encode_s_sparse_matrix]

Mandala collection

In[125]:=

```
# A list to accumulate random mandala images
mandala_images = []

# Generation loop
random.seed(443)
tstart = time.time()
for i in range(512):

    # Generate one random mandala figure
    fig2 = random_mandala(n_rows=None,
                          n_columns=None,
                          radius=[8, 6, 3],
                          rotational_symmetry_order=6,
                          symmetric_seed=True,
                          connecting_function='bezier_fill',
                          face_color="0.",
                          alpha = 1.0)
    fig2.tight_layout()

    # Convert the figure into an image and add it to the list
    mandala_images = mandala_images + [figure_to_image(fig2)]

    # Close figure to save memory
    plt.close(fig2)

# Invert image colors
mandala_images2 = [ImageOps.invert(img) for img in mandala_images]

# Binarize images
mandala_images3 = [im.convert('1') for im in mandala_images2]

# Resize images
width, height = mandala_images3[0].size
print([width, height])
ratio = height / width
new_width = 200
mandala_images4 = [img.resize((new_width, round(new_width * ratio)), Image.ANTIALIAS) for img in mandala_images3]

width, height = mandala_images4[0].size
print([width, height])
print("Process time: " + str(time.time()-tstart))
```

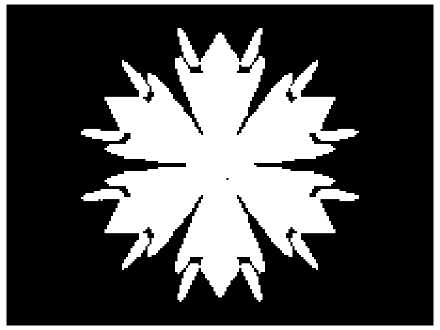
```
[640, 480]
[200, 150]
Process time: 21.96627712249756
```

Process random mandalas collection

Show one of the random mandalas:

In[∘]:=
```
mandala_images4[11]
```

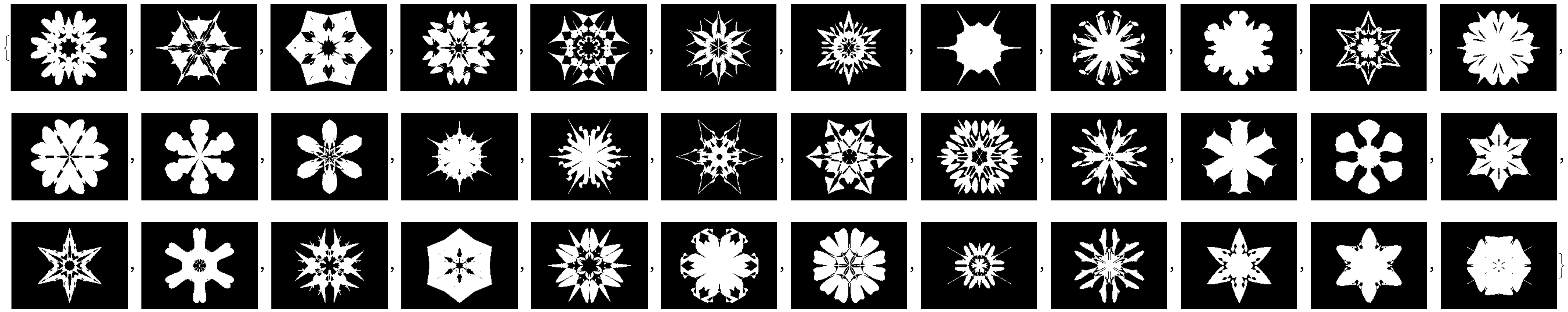
Out[∘]=



Show an array of inverted random mandalas:

In[∘]:=
```
random.sample(mandala_images4, 36)
```

Out[∘]=



Convert each image into array and flatten that array:

In[∘]:=
```
mandala_arrays = [numpy.asarray(x, dtype="int32") for x in mandala_images4]
len(mandala_arrays)
```

Out[∘]= 512

Make a matrix of flattened mandalas:

In[∘]:=
```
mandalaMat = [x.reshape(x.shape[0] * x.shape[1]) for x in mandala_arrays]
mandalaMat = numpy.array(mandalaMat)
mandalaMat.shape
```

Out[∘]= {512, 30 000}

Make the corresponding `SSparseMatrix` object:

In[∘]:=
```
mandalaSMat = SSparseMatrix(mandalaMat, row_names="", column_names="")
print(repr(mandalaSMat))
```

```
<512x30000 SSparseMatrix (sparse matrix with named rows and columns) of type '<class 'numpy.int32'>'
    with 3436475 stored elements in Compressed Sparse Row format, and fill-in 0.22372884114583333>
```

LSAMon object creation

Assign the number of topics for the computations below:

In[∘]:=
```
numberOfTopics = 40
```

Create the LSAMon object:

In[∘]:=
```
tStart = time.time()
lsaObj = LatentSemanticAnalyzer().set_document_term_matrix(mandalaSMat).apply_term_weight_functions("None", "None", "None")
tEnd = time.time()
print("\n\t\tCreation time : ", tEnd-tStart)
```

```
        Creation time :  0.06528496742248535
```

SVD

Here we extract image topics using Singular Value Decomposition (SVD):

In[∘]:=
```
tStart = time.time()

lsaSVDObj = lsaObj.extract_topics(number_of_topics=numberOfTopics, min_number_of_documents_per_term=0, method="SVD", max_steps=50)

tEnd = time.time()

print("Topic extraction time SVD : ", tEnd-tStart)
```

```
Topic extraction time SVD :  1.428023099899292
```

Get the SVD topics matrix:

In[∘]:=
```
svdH = lsaSVDObj.normalize_matrix_product(normalize_left=False).take_H().copy()
```

NNMF

Here we extract image topics using Non-Negative Matrix Factorization (NNFM):

In[∘]:=
```
tStart = time.time()
```

```
lsaNNMFObj=(LatentSemanticAnalyzer()
            .set_document_term_matrix(mandalaSMat)
            .apply_term_weight_functions("None", "None", "None")
            .extract_topics(number_of_topics=numberOfTopics, min_number_of_documents_per_term=0, method="NNMF", max_steps=16))

tEnd = time.time()

print("Topic extraction time NNMF : ", tEnd-tStart)
```

```
Topic extraction time NNMF :  609.1903309822083
```

Get the NNMF topics matrix:

In[∘]:=
```
nnmfH = lsaNNMFObj.normalize_matrix_product(normalize_left=False).take_H().copy()
```

ICA

Not implemented in Python yet.

Show topics interpretation

SVD

Show the importance coefficients of the topics (if SVD was used the plot would show the singular values):

Get the topics matrix from the LSA object and convert it into a dense array:

In[∘]:=
```
svdTopicsArray = svdH.sparse_matrix().todense()
svdTopicsArray.shape
```

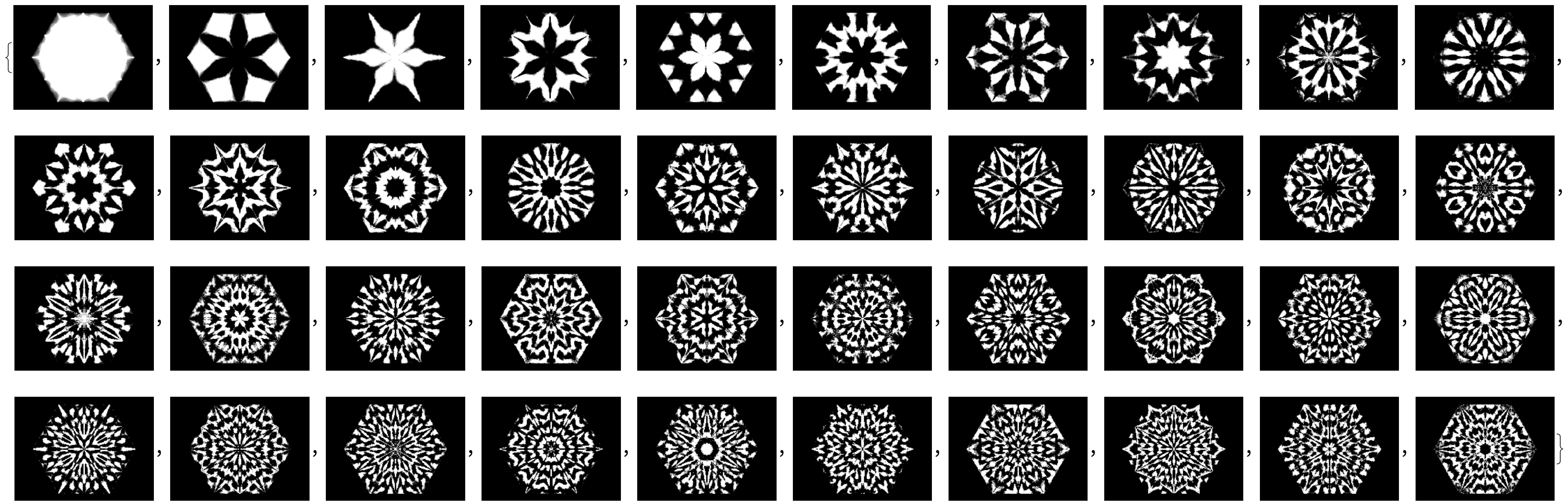
Out[∘]= {40, 30 000}

Show the interpretation of the extracted image topics:

In[∘]:=
```
imageSizeX, imageSizeY = mandala_images4[0].size
svdTopicImages = [(svdTopicsArray[i,:]*255).reshape(imageSizeY, imageSizeX) for i in range(svdTopicsArray.shape[0])]
svdTopicImages2 = [Image.fromarray(x) for x in svdTopicImages]
```

In[∘]:=
```
svdTopicImages2
```

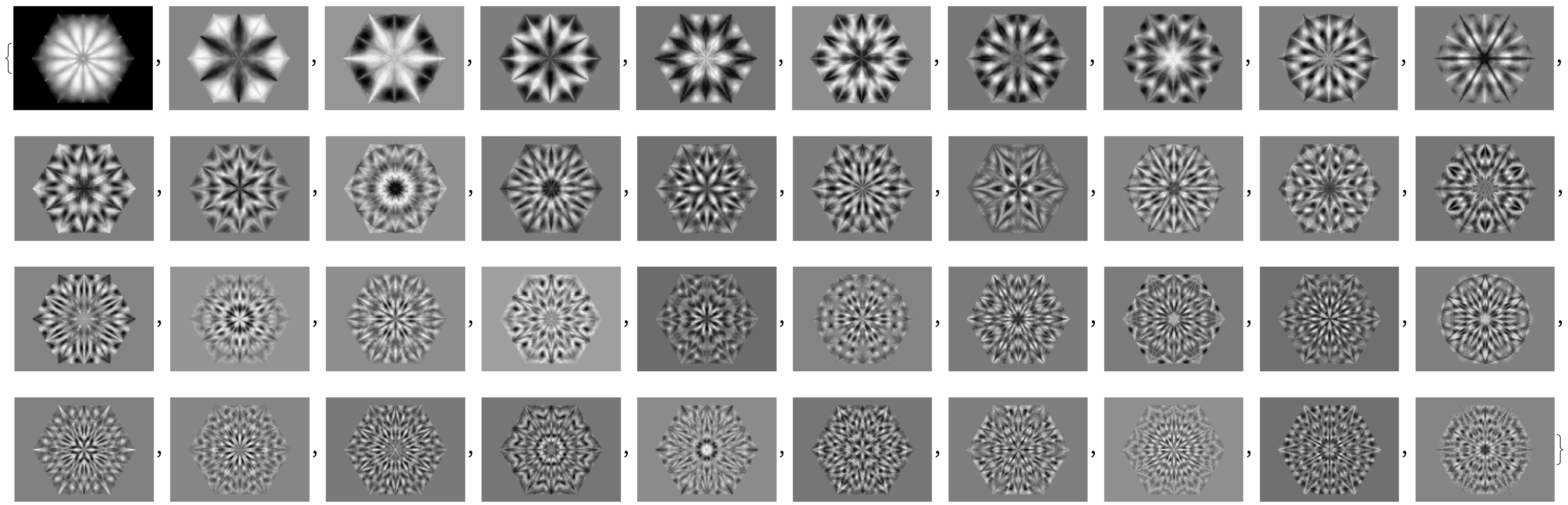
Out[ ]=



"Image adjusted" basis:

In[ ]:=
```
ImageAdjust@*Image /@ ExternalEvaluate[pythonSession, "svdTopicImages"]
```

Out[ ]=



We can see after the image adjustment we get values in [0, 1].

```
In[ ]:= Block[{imgs = ExternalEvaluate[pythonSession, "svdTopicImages"]},
  ResourceFunction["GridTableForm"][{{
     ResourceFunction["RecordsSummary"][Flatten[Normal /@ imgs]],
     ResourceFunction["RecordsSummary"][Flatten[ImageData@*ImageAdjust@*Image /@ imgs]]}}, TableHeadings → {"Obtained", "Normalized"}]
 ]
```

Out[ ]=

#	Obtained	Normalized
1	1 column 1 Min -9.47429 1st Qu 0. 3rd Qu 0. Median 0. Mean 0.0100618 Max 8.77935	1 column 1 Min 0. 1st Qu 0.449813 Mean 0.495668 Median 0.500848 3rd Qu 0.555578 Max 1.

NNMF

Show the importance coefficients of the topics :

```
In[ ]:= nnmfTopicsArray = nnmfH.sparse_matrix().todense()
nnmfTopicsArray.shape
```

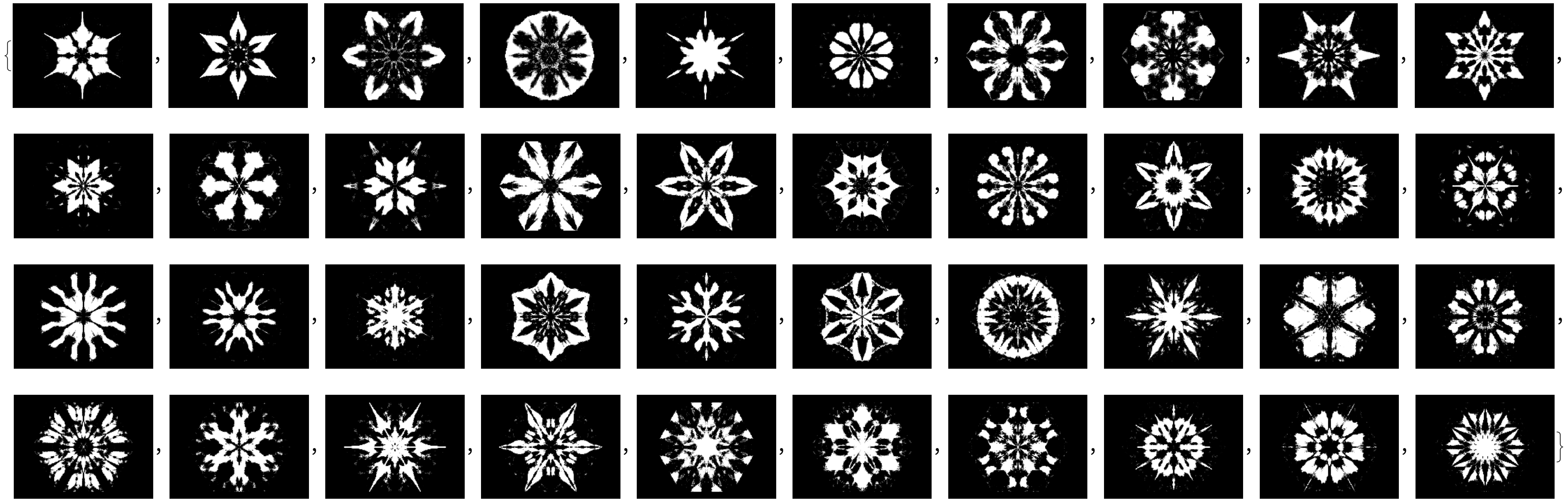
Out[ ]= {40, 30 000}

Show the interpretation of the extracted image topics:

```
In[ ]:= imageSizeX, imageSizeY = mandala_images4[0].size
nnmfTopicImages = [(nnmfTopicsArray[i,:]*255).reshape(imageSizeY, imageSizeX) for i in range(nnmfTopicsArray.shape[0])]
nnmfTopicImages2 = [Image.fromarray(x) for x in nnmfTopicImages]
```

```
In[ ]:= nnmfTopicImages2
```

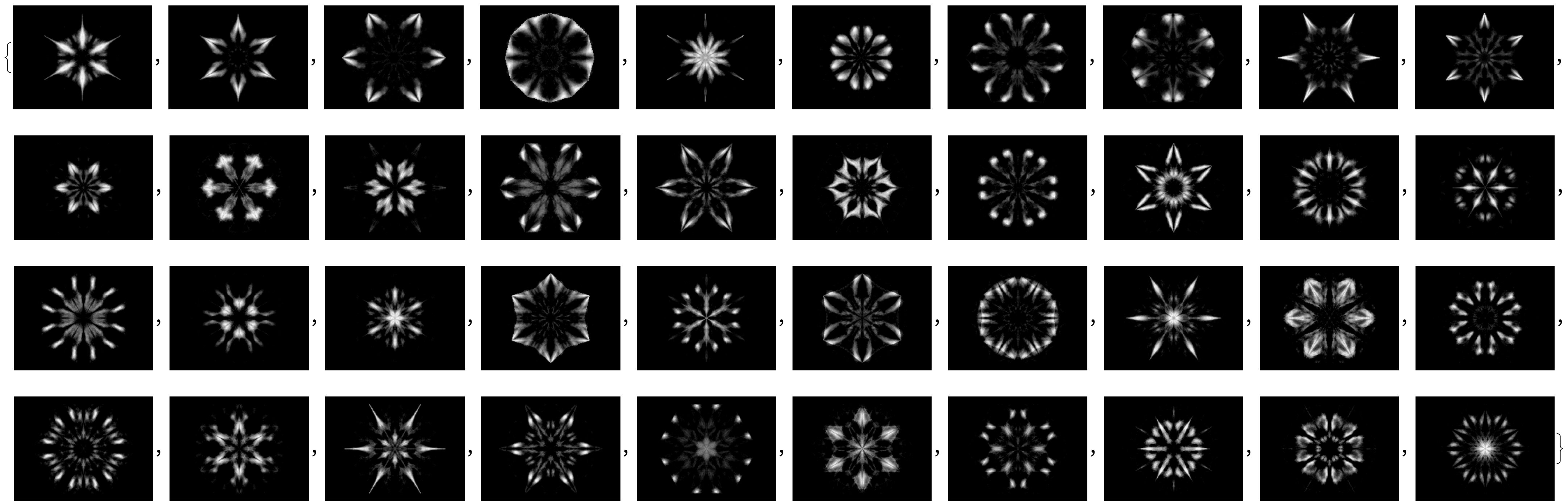
Out[∘]=



"Image adjusted" basis:

In[∘]:=
```
ImageAdjust@*Image /@ ExternalEvaluate[pythonSession, "nnmfTopicImages2"]
```

Out[∘]=

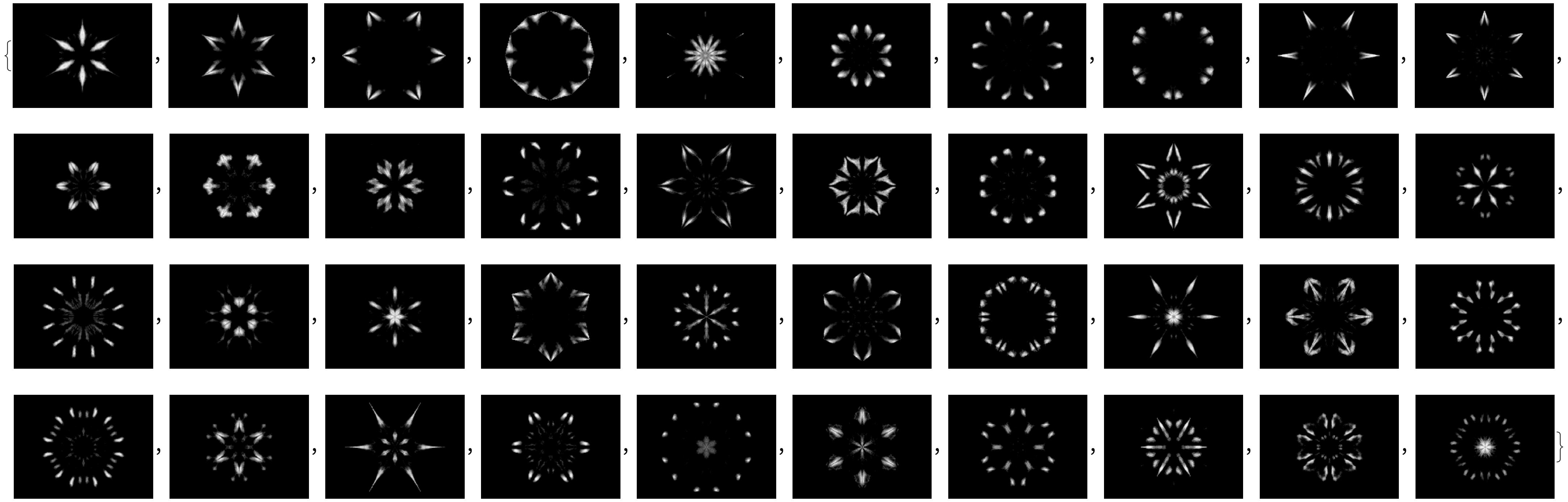


In[∘]:=
```
nnmfTopicImages3 = [(numpy.clip(a = nnmfTopicsArray[i,:], a_min = 0.01, a_max = 1) * 255).reshape(imageSizeY, imageSizeX) for i in range(nnmfTopicsArray.shape[0])]
nnmfTopicImages4 = [Image.fromarray(x) for x in nnmfTopicImages3]
```

"Image adjusted" basis clipped:

In[∘]:=
```
ImageAdjust@*Image /@ ExternalEvaluate[pythonSession, "nnmfTopicImages4"]
```

Out[∘]=



We can see after the image adjustment we get values in [0, 1].

In[∘]:=
```
Block[{imgs = ExternalEvaluate[pythonSession, "nnmfTopicImages"]},
 ResourceFunction["GridTableForm"][{{
    ResourceFunction["RecordsSummary"][Flatten[Normal /@ imgs]],
    ResourceFunction["RecordsSummary"][Flatten[ImageData@*ImageAdjust@*Image /@ imgs]]}}, TableHeadings → {"Obtained", "Normalized"}]
]
```

Out[∘]=

#	Obtained	Normalized
1	1 column 1 1st Qu 0. 3rd Qu 0. Median 0. Min 0. Mean 0.482169 Max 13.3194	1 column 1 1st Qu 0. 3rd Qu 0. Median 0. Min 0. Mean 0.0542883 Max 1.

Clip the right, topics factor of NNMF LSA object:

In[∘]:=
```
nnmfH.sparse_matrix() > 0.01
```

Out[∘]=
```
ExternalObject[ Command: csr_matrix  System: Python
                Type: PythonObject   IsModule: False ]
```

ICA

Not implemented in Python yet.

Approximation

Get a new, unseen mandala:

In[◦]:=

```
# Consider using radius = [8, 6, 3]
fig = random_mandala(radius=[8, 6, 3],
                     rotational_symmetry_order=6,
                     symmetric_seed=True,
                     connecting_function='bezier_fill',
                     face_color="0.",
                     alpha = 1.0)

# Convert the mandala figure to image
testMandala = figure_to_image(fig)
testMandala
```

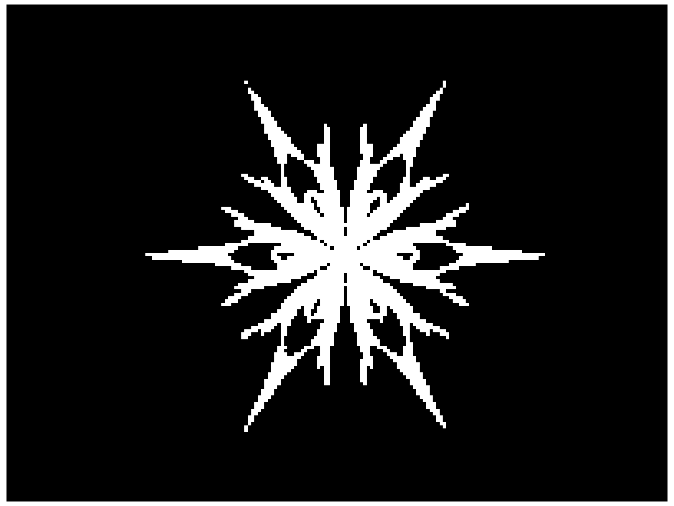
Out[◦]=

In[◦]:=

```
# Resize image
testMandala1 = testMandala.resize((new_width, round(new_width * ratio)), Image.ANTIALIAS)

# Invert image colors
testMandala2 = ImageOps.invert(testMandala)

# Binarize image
testMandala3 = testMandala2.convert('1')
```

```
# Resize image
testMandala4 = testMandala3.resize((new_width, round(new_width * ratio)), Image.ANTIALIAS)

# The image
testMandala4
```

Out[•]=



Get the corresponding vector:

In[•]:=

```
# Flatten array:
testMandalaArray = numpy.asarray(testMandala4, dtype="int32")
```

Make a query matrix:

In[•]:=

```
testMandalaMat = testMandalaArray.reshape(1, testMandalaArray.shape[0] * testMandalaArray.shape[1])
testMandalaMat = numpy.array(testMandalaMat)
matQuery = SSparseMatrix(testMandalaMat, row_names="", column_names="")
print(repr(matQuery))
```

```
<1x30000 SSparseMatrix (sparse matrix with named rows and columns) of type '<class 'numpy.int32'>'
    with 2780 stored elements in Compressed Sparse Row format, and fill-in 0.09266666666666666>
```

SVD

Represent by topics:

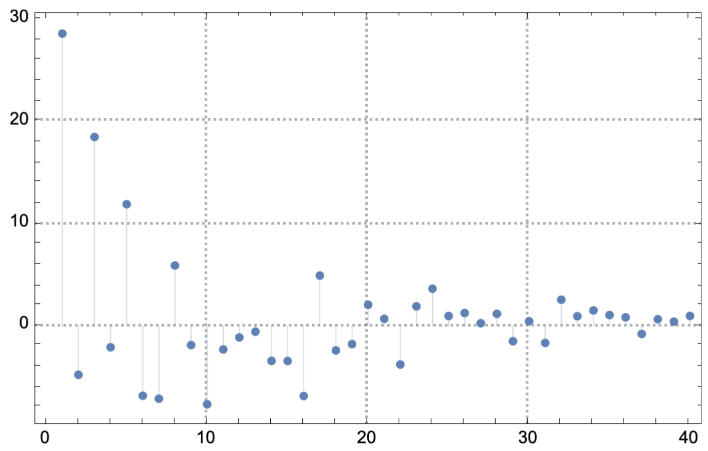
In[•]:=

```
resMat = lsaSVDObj.represent_by_topics(matQuery, method="recommendation").take_value()
resMat
```

Out[•]= SparseArray[Specified elements: 40 Dimensions: {1, 40}]

In[•]:=
```
ListPlot[SparseArray[ExternalEvaluate[pythonSession, "resMat"]][[1]], ...]
```

Out[•]=

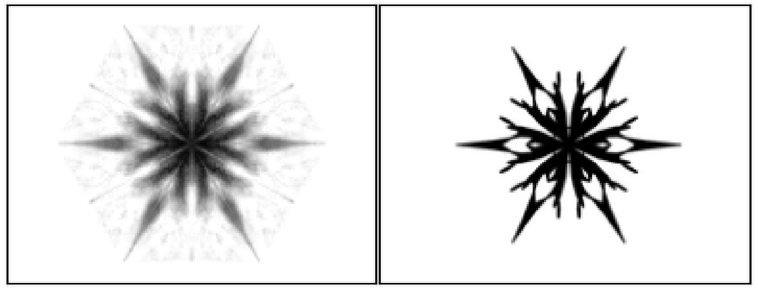


In[•]:=
```python
approx = resMat.dot(svdH)
approx2 = apply_term_weight_functions(approx, "None", "None", "AbsMax")
approx2.data
```

Out[•]= NumericArray[Type: Real64 Dimensions: {14 274}]

In[•]:=
```python
approxImg = (255 - approx2.todense().reshape(imageSizeY, imageSizeX)*255)
approxImg2 = Image.fromarray(approxImg)

fig = plt.figure(figsize=(5., 5.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(1,2),
                 axes_pad=0.02,
                 )

for ax, img in zip(grid, [approxImg2, testMandala1]):
    ax.imshow(img)
    ax.set(xticks=[], yticks=[])

figure_to_image(fig)
```

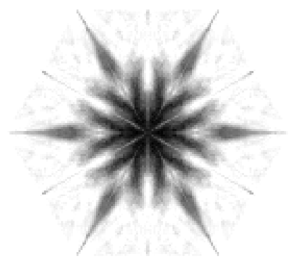
Out[◦]=



Direct manipulation:

In[◦]:=
```
ColorNegate[Image[ExternalEvaluate[pythonSession, "approx2.todense().reshape(imageSizeY, imageSizeX)"]]]
```

Out[◦]=



NNMF

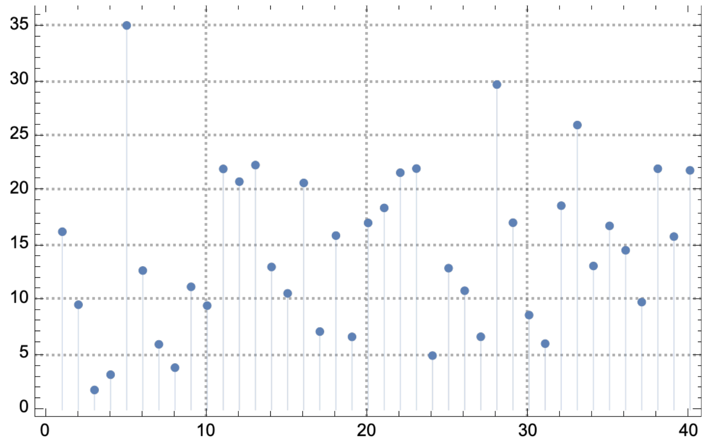
Represent by topics:

In[◦]:=
```
resMat = lsaNNMFObj.represent_by_topics(matQuery, method="recommendation").take_value()
resMat
```

Out[◦]= SparseArray[Specified elements: 40 Dimensions: {1, 40}]

In[∘]:= ListPlot[SparseArray[ExternalEvaluate[pythonSession, "resMat"]][[1]], ⋯]

Out[∘]=



In[∘]:=

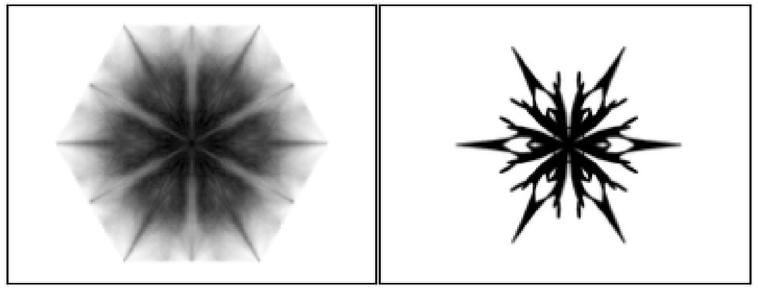
```
approx = resMat.dot(nnmfH)
approx2 = apply_term_weight_functions(approx, "None", "None", "AbsMax")
approx2.data
```

Out[∘]= NumericArray[Type: Real64 Dimensions: {14 274}]

In[∘]:=

```
approxImg = (255 - approx2.todense().reshape(imageSizeY, imageSizeX)*255)
approxImg2 = Image.fromarray(approxImg)

fig = plt.figure(figsize=(5., 5.))
grid = ImageGrid(fig, 111,
                 nrows_ncols=(1,2),
                 axes_pad=0.02,
                 )

for ax, img in zip(grid, [approxImg2, testMandala1]):
    ax.imshow(img)
    ax.set(xticks=[], yticks=[])

figure_to_image(fig)
```

Out[ ]=

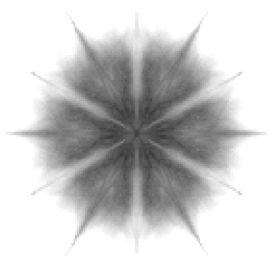


Direct manipulation (sometimes produces better results):

In[ ]:=

In[ ]:=
```
Block[{imgArr = ExternalEvaluate[pythonSession, "approx2.todense().reshape(imageSizeY, imageSizeX)"], m},
 m = Mean@Flatten@Normal@imgArr;
 ColorNegate[Image[Normal[imgArr] - m]]
]
```

Out[ ]=



ICA

Not implemented yet in Python.