PyTransKit

Release 0.1

Abu Hasnat Mohammad Rubaiyat, Xuwang Yin, Liam Cattell, Sohe

Aug 24, 2023

CONTENTS

Python Transport Based Signal Processing Toolkit

CHAPTER

ONE

INSTALLATION

The library could be installed through pip

```
pip install pytranskit
```

Alternately, you could clone/download the repository and add the `pytranskit` directory to your Python path

```
import sys
sys.path.append('path/to/pytranskit')

from pytranskit.optrans.continuous.cdt import CDT
```

CHAPTER

TWO

TUTORIALS

2.1 Cumulative Distribution Transform (CDT)

This tutorial will demonstrate: how to use the forward and inverse operations of the CDT in the the *PyTransKit* package.

2.1.1 Class:: CDT

Functions:

1. Forward transform: sig1_hat, sig1_hat_old, xtilde = forward(x0, sig0, x1, sig1)

```
Inputs:
----------------
x0 : 1d array
    Independent axis variable of reference signal (sig0).
sig0 : 1d array
    Reference signal.
x1 : 1d array
    Independent axis variable of the signal to transform (sig1).
sig1 : 1d array
    Signal to transform.

Outputs:
----------------
sig1_hat : 1d array
    CDT of input signal sig1.
sig1_hat_old : 1d array
    CDT of input signal sig1 using the old definition.
xtilde : 1d array
    Independent axis variable of sig1_hat.
```

2. Inverse transform: sig1_recon = inverse(sig1_hat, sig0, x1)

```
Inputs:
----------------
sig1_hat : 1d array
    CDT of a signal sig1.
sig0 : 1d array
    Reference signal.
x1 : 1d array
```

(continues on next page)

(continued from previous page)

```
    Independent axis variable of the signal to reconstruct.

Outputs:
----------------
sig1_recon : 1d array
    Reconstructed signal.
```

2.1.2 Definition

Forward Transform

Let $s(x), x \in \Omega_s \subset \mathbb{R}$ be a positive density function (PDF). The CDT of the PDF $s(x)$ with respect to a reference PDF $s_0(x), x \in \Omega_{s_0} \subset \mathbb{R}$ is given by the mass preserving function $\widehat{s}(x)$ that satisfies:

$$\int_{inf(\Omega_s)}^{\widehat{s}(x)} s(u)du = \int_{inf(\Omega_{s_0})}^{x} s_0(u)du \tag{2.1}$$

which yields:

$$\widehat{s}(x) = S^{-1}(S_0(x)), \tag{2.2}$$

where $S(x) = \int_{inf(\Omega_s)}^{x} s(u)du$, and $S_0(x) = \int_{inf(\Omega_{s_0})}^{x} s_0(u)du$. For continuous positive PDFs, $\widehat{s}$ is a continuous and monotonically increasing function. If $\widehat{s}$ is differentiable, the above equation can be rewritten as:

$$s_0(x) = \widehat{s}'(x)s(\widehat{s}(x)). \tag{2.3}$$

Inverse Transform

The inverse transform of the CDT $\widehat{s}(x)$ is given by:

$$s(x) = (\widehat{s}^{-1}(x))'s_0(\widehat{s}^{-1}(x)), \tag{2.4}$$

where $\widehat{s}^{-1}$ is the inverse of $\widehat{s}$, i.e. $\widehat{s}^{-1}(\widehat{s}(x)) = x$.

2.1.3 CDT Demo

The examples will cover the following operations: * Forward and inverse operations of the CDT * CDT Properties - translation, scaling, and linear separability in CDT space

2.1.4 Forward CDT

Import necessary python packages

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Create reference I_0 (uniform distribution) and a demo signal I_1

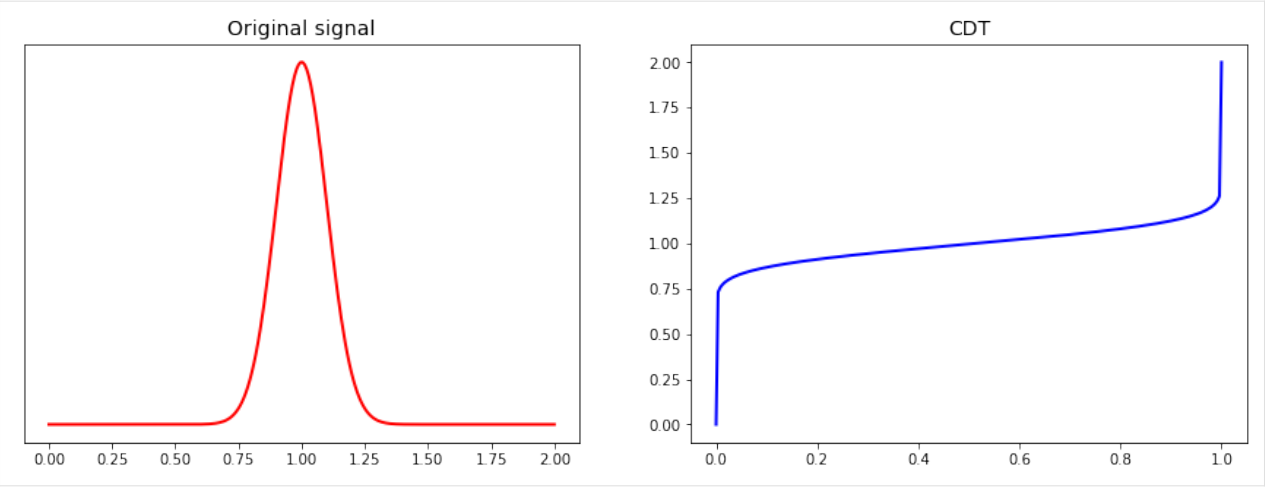
[2]:
```
N=250
x0 = np.linspace(0, 1, N)
I0= np.ones(x0.size)
x=np.linspace(0, 2, N)
mu=x[len(x)-1]/2.
sigma=0.1
I1=1/(sigma*np.sqrt(2*np.pi))*np.exp(-((x-mu)**2)/(2*sigma**2))
```

Convert signals to strictly positive PDFs

[3]:
```
epsilon = 1e-7
I0 = abs(I0) + epsilon
I0 = I0/I0.sum()
I1 = abs(I1) + epsilon
I1 = I1/I1.sum()
```

Compute forward CDT transform of I_1

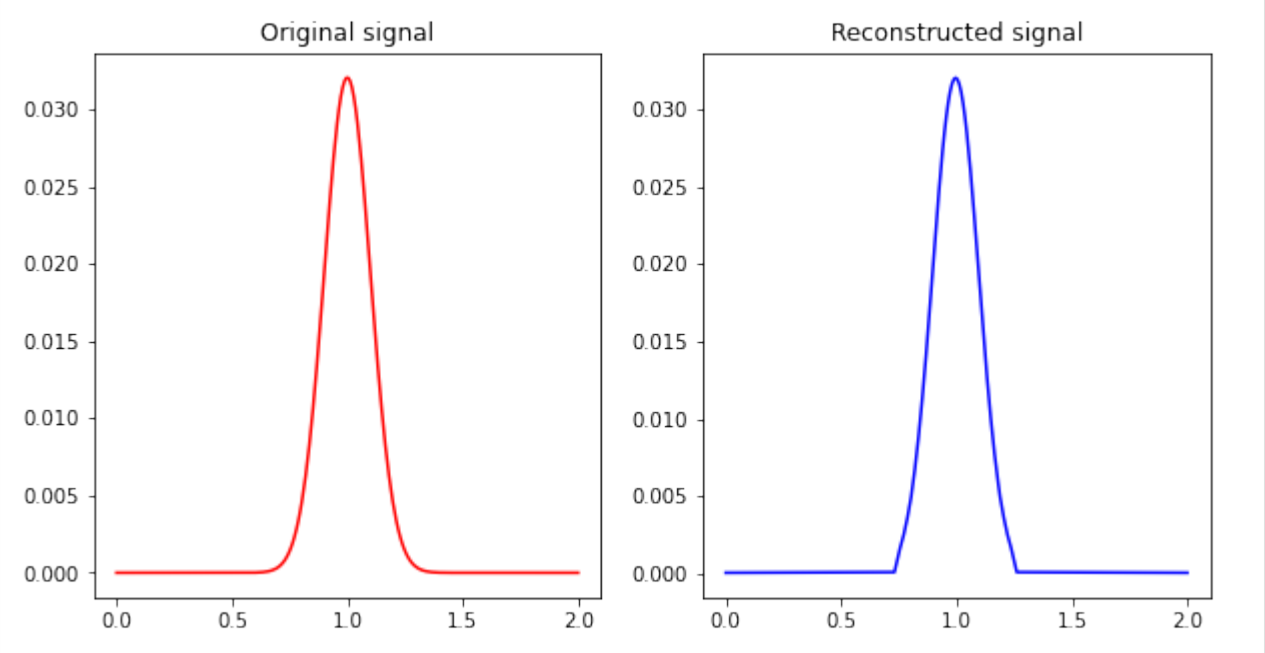
[4]:
```
import sys
sys.path.append('../')

from pytranskit.optrans.continuous.cdt import CDT

# Create a CDT object
cdt = CDT()

# Compute the forward transform
I1_hat, I1_hat_old, xtilde = cdt.forward(x0, I0, x, I1)

# Plot I1 and I1_hat
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=False, sharey=False, figsize=(15,5))
ax[0].plot(x, I1, 'r-',linewidth=2)
ax[0].set_title('Original signal',fontsize=fontSize)
ax[0].set_yticks([])

ax[1].plot(xtilde, I1_hat, 'b-',linewidth=2)
ax[1].set_title('CDT',fontsize=fontSize)
#ax[1].set_yticks([])

#ax[2].plot(xtilde, I1_hat_old, 'g-',linewidth=2)
#ax[2].set_title('CDT (old definition)',fontsize=fontSize)
#ax[2].set_yticks([])

plt.show()
```



2.1.5 Inverse CDT

Reconstruct the original signal I_1 from $\hat{I}_1$ using inverse CDT

[5]:
```
I1_hat[0] = 0
I1_recon = cdt.inverse(I1_hat, I0, x)

# Plot I1 and I1_recon
fig, ax = plt.subplots(1, 2, sharex=True, sharey=False, figsize=(10,5))
ax[0].plot(x, I1, 'r-')
ax[0].set_title('Original signal')

ax[1].plot(x, I1_recon, 'b-')
ax[1].set_title('Reconstructed signal')
#ax[1].set_yticks([])

plt.show()
```



2.1.6 Translation

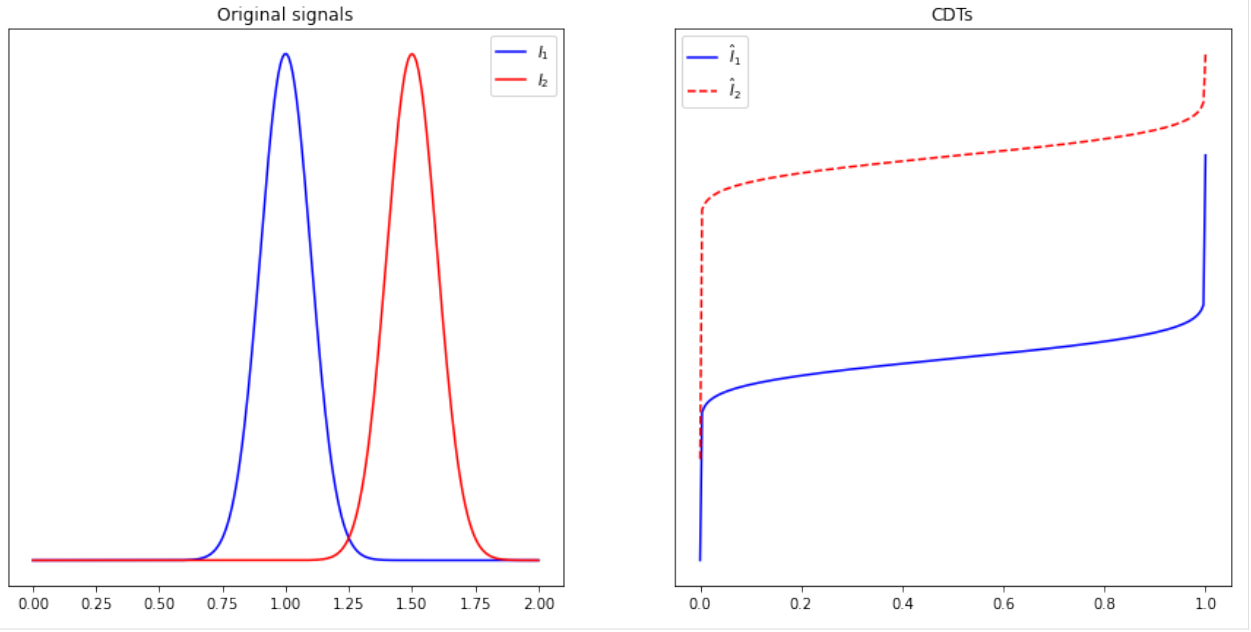
Generate a second signal I_2 which is a shifted version I_1, i.e. $I_2 = I_1(t - \tau)$. Then convert the signals into PDFs and compute CDT for both signals

[6]:
```
from scipy.ndimage.interpolation import shift
tau = 0.5
I1=1/(sigma*np.sqrt(2*np.pi))*np.exp(-((x-mu)**2)/(2*sigma**2))

I2 = np.interp(x-tau, x, I1)

# Convert signals to strictly positive PDFs
I1 = abs(I1) + epsilon
I1 = I1/I1.sum()
I2 = abs(I2) + epsilon
I2 = I2/I2.sum()

# Create a CDT object
cdt1 = CDT()

# Compute the forward transform
I1_hat, I1_hat_old, xtilde = cdt1.forward(x0, I0, x, I1, rm_edge=False)
I2_hat, I2_hat_old, xtilde = cdt1.forward(x0, I0, x+tau, I2, rm_edge=False)

#Plot the signals and CDTs
fig, ax = plt.subplots(1, 2, sharex=False, sharey=False, figsize=(15,7))
ax[0].plot(x, I1, 'b-')
ax[0].plot(x, I2, 'r-')
ax[0].set_title('Original signals')
ax[0].set_yticks([])
```

(continues on next page)

(continued from previous page)

```
ax[0].legend([r'$I_1$',r'$I_2$'])

ax[1].plot(xtilde, I1_hat, 'b-')
ax[1].plot(xtilde, I2_hat, 'r--')
ax[1].set_yticks([])
ax[1].set_title('CDTs')
ax[1].legend([r'$\hat{I}_1$',r'$\hat{I}_2$'])

#ax[2].plot(xtilde, f1_hat, 'b-')
#ax[2].plot(xtilde, f2_hat, 'r--')
#ax[2].set_yticks([])
#ax[2].set_title('Transport map')
#ax[2].legend([r'$f_1$',r'$f_2$'])

plt.show()
```



2.1.7 Scaling

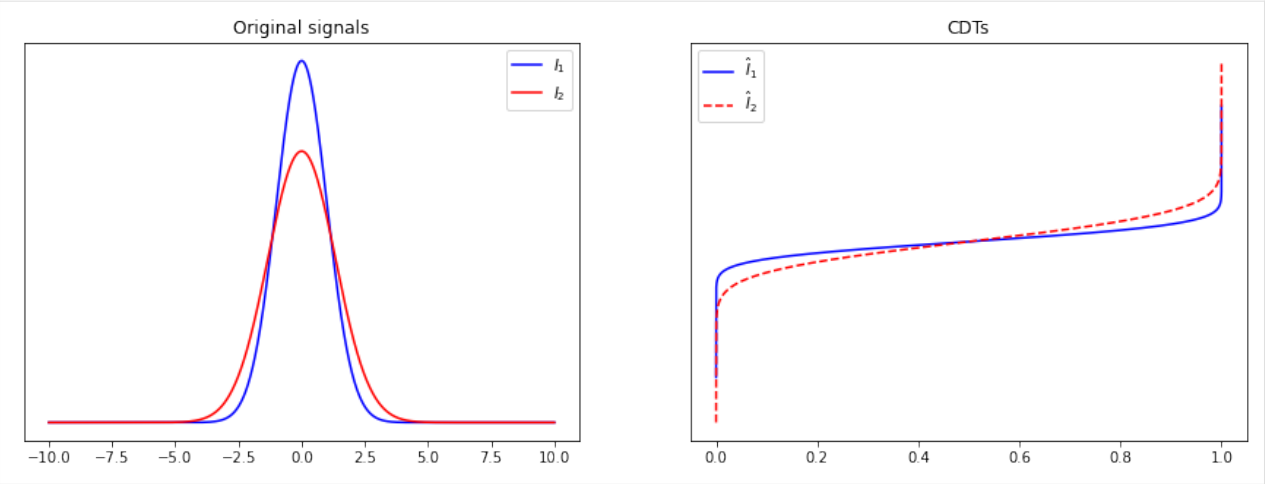
Generating a scaled version of I_1, i.e. $I_2 = I_1(\alpha t)$, and computing CDTs of both signals.

[7]:
```
from scipy.stats import norm

x = np.arange(-10,10,0.001)

N=len(x)

x0 = np.linspace(0, 1, round(N/2))
I0= np.ones(x0.size)

I1 = norm.pdf(x,0,1)
```

(continues on next page)

(continued from previous page)

```
# create a scaled version of I1; i.e. I2(t) = I1(alpha*t)
alpha = 0.75
I2 = np.interp(alpha*x, x, I1)

# Convert signals to strictly positive PDFs
I0 = abs(I0) + epsilon
I0 = I0/I0.sum()
I1 = abs(I1) + epsilon
I1 = I1/I1.sum()
I2 = abs(I2) + epsilon
I2 = I2/I2.sum()

# Create a CDT object
cdt2 = CDT()

# Compute the forward transform
I1_hat, I1_hat_old, xtilde = cdt2.forward(x0, I0, x, I1, rm_edge=False)
I2_hat, I2_hat_old, xtilde = cdt2.forward(x0, I0, x/alpha, I2, rm_edge=False)

#Plot the signals and CDTs
fig, ax = plt.subplots(1, 2, sharex=False, sharey=False, figsize=(15,5))
ax[0].plot(x, I1, 'b-')
ax[0].plot(x, I2, 'r-')
ax[0].set_title('Original signals')
ax[0].set_yticks([])
ax[0].legend([r'$I_1$',r'$I_2$'])

ax[1].plot(xtilde, I1_hat, 'b-')
ax[1].plot(xtilde, I2_hat, 'r--')
ax[1].set_yticks([])
ax[1].set_title('CDTs')
ax[1].legend([r'$\hat{I}_1$',r'$\hat{I}_2$'])

#ax[2].plot(xtilde, f1_hat, 'b-')
#ax[2].plot(xtilde, f2_hat, 'r--')
#ax[2].set_yticks([])
#ax[2].set_title('Transport map')
#ax[2].legend([r'$f_1$',r'$f_2$'])

plt.show()
```



2.1.8 Linear Separability

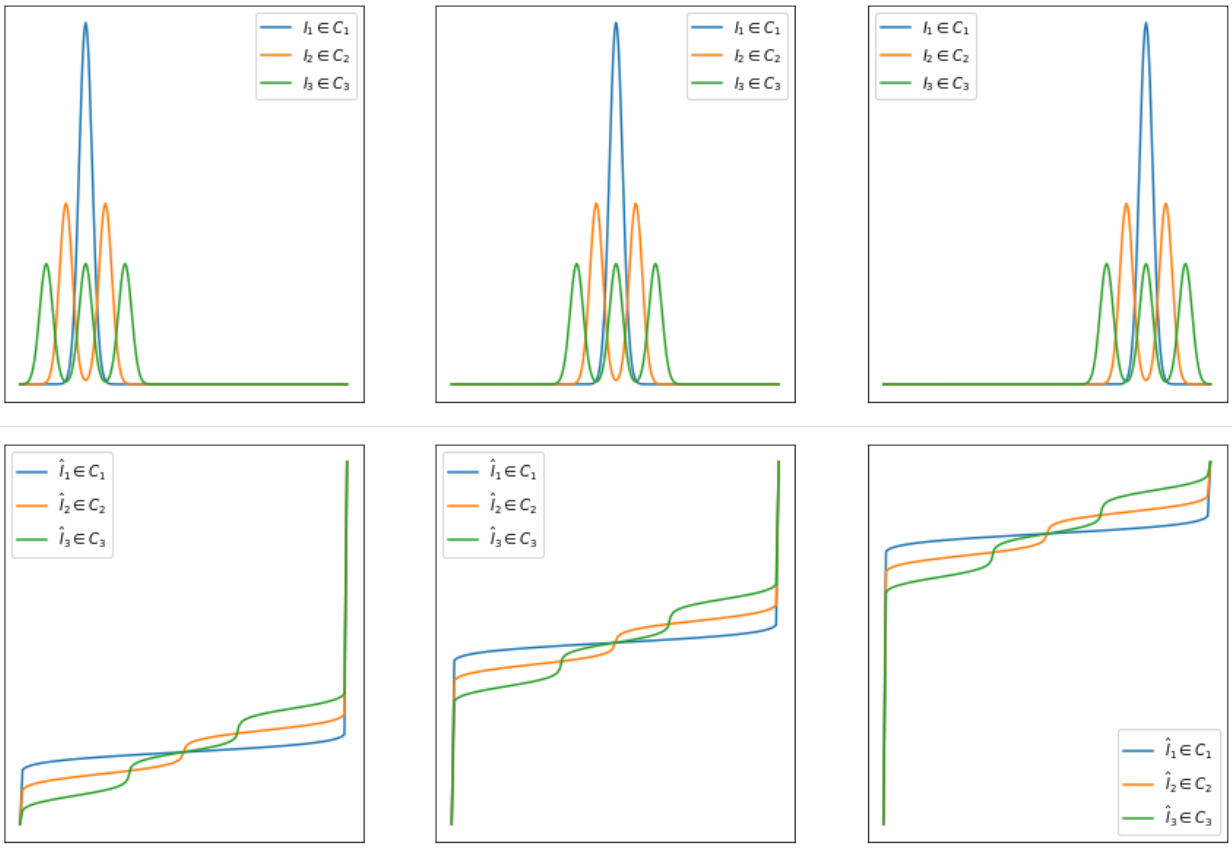
Here we are defining three classes of signal, i.e. class $k \in \{1, 2, 3\}$. Each class k consists of translated versions of a k-modal Gaussian distribution.

[8]:
```
N=250
x=np.arange(N)

# Creating reference signal I0
x0 = np.linspace(0, 1, round(N/2))
I0= np.ones(x0.size)
epsilon = 1e-7
I0 = abs(I0) + epsilon
I0 = I0/I0.sum()

# Creating three classes of translated versions of k-modal Gaussian distribution --␣
↪skolouri
K=3 # Number of classes
L=500 # Number of datapoints per class

rm_edge = False
if rm_edge:
    rml = 2
else:
    rml = 0

I=np.zeros((K,L,N))
Ihat=np.zeros((K,L,x0.size-rml))
kmodal_shift=[]
kmodal_shift.append(np.array([0]))
kmodal_shift.append(np.array([-15,15]))
kmodal_shift.append(np.array([-30,0,30]))
sigma=5
for k in range(K):
    for i,mu in enumerate(np.linspace(50,200,L)):
        for j in range(k+1):
```

(continues on next page)

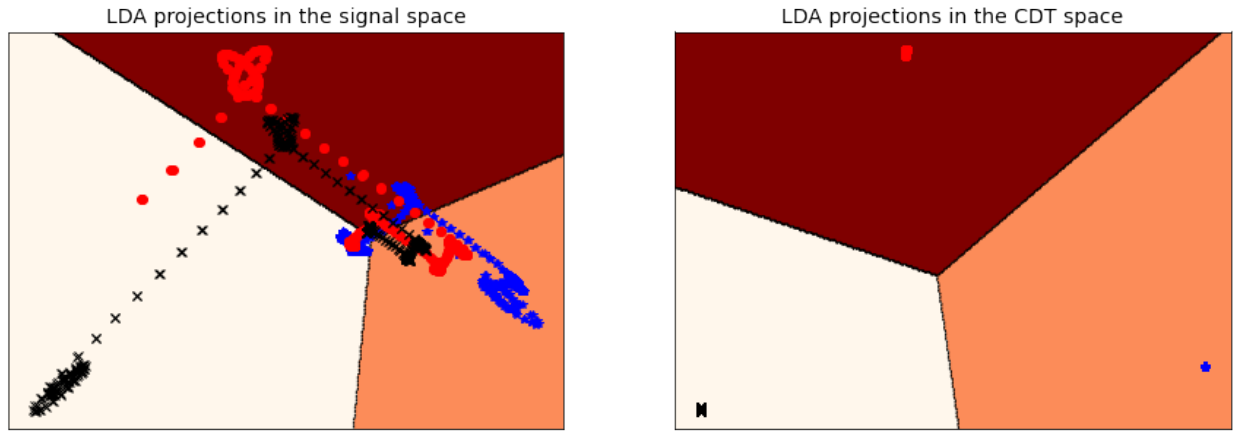
(continued from previous page)

```
            I[k,i,:]+=1/((k+1)*sigma*np.sqrt(2*np.pi))*np.exp(-((x-mu-kmodal_
↪shift[k][j])**2)/(2*sigma**2))
        I[k,i,:] = abs(I[k,i,:]) + epsilon
        I[k,i,:] = I[k,i,:]/I[k,i,:].sum()
        Ihat[k,i,:], _,xtilde = cdt.forward(x0, I0, x+mu, I[k,i,:], rm_edge)
        #Ihat[k,i,:]=cdt.transform(I[k,i,:])
fig,ax=plt.subplots(1,3,figsize=(15,5))
for count,index in enumerate([0,int(L/2),L-1]):
    for k in range(K):
        ax[count].plot(x, I[k,index,:]) #template
        ax[count].set_xticks([])
        ax[count].set_yticks([])
        ax[count].legend([r'$I_1\in C_1$',r'$I_2\in C_2$',r'$I_3\in C_3$'])
plt.show()

fig,ax=plt.subplots(1,3,figsize=(15,5))
for count,index in enumerate([0,int(L/2),L-1]):
    for k in range(K):
        ax[count].plot(xtilde, Ihat[k,index,:]) #template
        ax[count].set_xticks([])
        ax[count].set_yticks([])
        ax[count].legend([r'$\hat{I}_1\in C_1$',r'$\hat{I}_2\in C_2$',r'$\hat{I}_3\in C_3
↪$'])
plt.show()
```



Now we'll do linear classification in Signal and CDT spaces. LDA is used for visualization.

[9]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.svm import LinearSVC

import warnings
warnings.filterwarnings('ignore')

X=np.reshape(I,(K*L,N))      #Combine the signals into a features vector X
Xhat=np.reshape(Ihat,(K*L,x0.size-rml))     #Combine the transformed signals into a␣
↪features vector Xhat
data=[X,Xhat]
label=np.concatenate((np.zeros(L,),np.ones(L,),-1*np.ones(L,))) # Define the labels as -
↪1,0,1 for the three classes
lda=LinearDiscriminantAnalysis(n_components=2)
svm=LinearSVC()
title_=['LDA projections in the signal space','LDA projections in the CDT space']
fig,ax=plt.subplots(1,2,figsize=(15,5))
for i in range(2):
    dataLDA=lda.fit_transform(data[i],label)
    ax[i].plot(dataLDA[:L,0],dataLDA[:L,1],'b*')
    ax[i].plot(dataLDA[L:2*L,0],dataLDA[L:2*L,1],'ro')
    ax[i].plot(dataLDA[2*L:,0],dataLDA[2*L:,1],'kx')
    x_min, x_max = ax[i].get_xlim()
    y_min, y_max = ax[i].get_ylim()
    nx, ny = 400, 200
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),np.linspace(y_min, y_max, ny))
    svm.fit(dataLDA,label)
    Z = svm.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z[:].reshape(xx.shape)
    ax[i].pcolormesh(xx, yy, Z,cmap='OrRd')
    ax[i].contour(xx, yy, Z, linewidths=.5, colors='k')
    ax[i].set_xticks([])
    ax[i].set_yticks([])
    ax[i].set_title(title_[i],fontsize=fontSize)
plt.show()
```



[ ]:

2.2 Radon-Cumulative Distribution Transform (R-CDT)

This tutorial will demonstrate: how to use the forward and inverse operations of the R-CDT in the *PyTransKit* package.

2.2.1 Class:: RadonCDT

Functions:

1. Forward transform: sig1_hat = forward(x0_range, sig0, x1_range, sig1)

```
Inputs:
----------------
x0_range : 1x2 array
    contains lower and upper limits for independent variable of reference signal␣
↪(sig0). Example: [0,1].
sig0 : 2d array
    Reference signal.
x1_range : 1x2 array
    contains lower and upper limits for independent variable of input signal (sig1).
↪ Example: [0,1].
sig1 : 2d array
    Signal to transform.

Outputs:
----------------
sig1_hat : 2d array
    R-CDT of input signal sig1.
```

2. Inverse transform: sig1_recon = inverse(sig1_hat, sig0, x1_range)

```
Inputs:
----------------
sig1_hat : 2d array
    R-CDT of a signal sig1.
sig0 : 2d array
    Reference signal.
x1_range : 1x2 array
    contains lower and upper limits for independent variable of input signal (sig1).
↪ Example: [0,1].

Outputs:
----------------
sig1_recon : 2d array
    Reconstructed signal.
```

2.2.2 Definition

Forward Transform

Let $s(\mathbf{x}), \mathbf{x} \in \Omega_s \subset \mathbb{R}^2$ and $s_0(\mathbf{x}), \mathbf{x} \in \Omega_{s_0} \subset \mathbb{R}^2$ define a given image and a reference image, respectively, which we consider to be appropriately normalized. The forward R-CDT of $s(\mathbf{x})$ is given by the measure preserving function $\widehat{s}(t, \theta)$ that satisfies:

$$\int_{inf(\Omega_s)}^{\widehat{s}(t,\theta)} \widetilde{s}(u, \theta) du = \int_{inf(\Omega_{s_0})}^{t} \widetilde{s}_0(u) du, \quad \forall \theta \in [0, \pi] \tag{2.5}$$

where, the Radon transform of the image $s(\mathbf{x})$ is given by

$$\widetilde{s}(t, \theta) = \int_{\Omega_s} s(\mathbf{x}) \delta(t - \mathbf{x}.\xi_\theta) d\mathbf{x}, \quad \xi_\theta = [\cos(\theta) \quad \sin(\theta)]^T \tag{2.6}$$

Inverse Transform

The transformed signal in R-CDT space can be recovered via the following inverse formula:

$$s(\mathbf{x}) = \mathcal{R}^{-1} \left(\frac{\partial \widehat{s}^{-1}(t, \theta)}{\partial t} \widetilde{s}_0 \left(\widehat{s}^{-1}(t, \theta), \theta \right) \right), \tag{2.7}$$

where $\mathcal{R}^{-1}(.)$ denotes the inverse Radon transform.

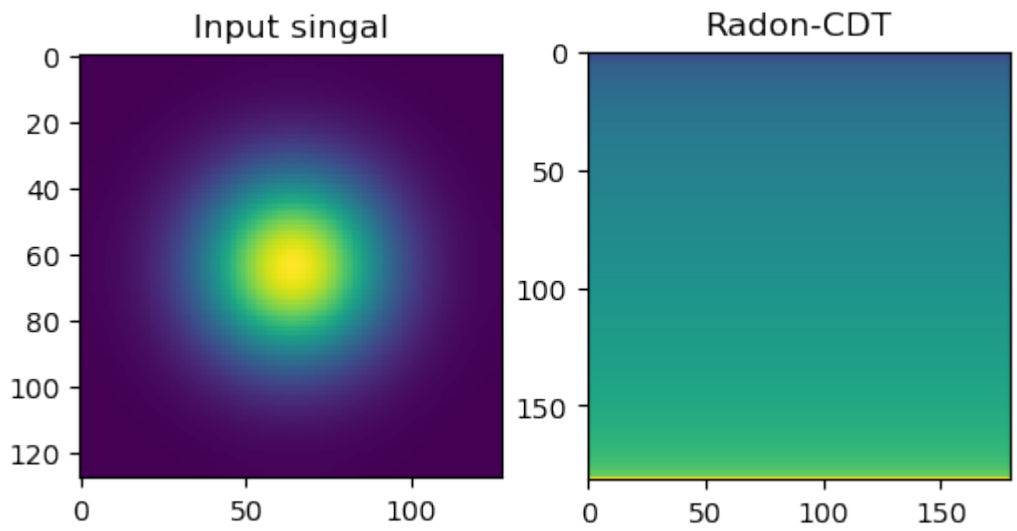
2.2.3 R-CDT Demo

The examples will cover the following operations: * Forward and inverse operations of the R-CDT * Classification using R-CDT

[1]:
```
import sys
sys.path.append('../')

import numpy as np
from skimage.io import imread
from skimage.color import rgb2gray
from skimage.transform import radon, iradon
from pytranskit.optrans.continuous.radoncdt import RadonCDT
from pytranskit.optrans.utils import signal_to_pdf
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from scipy.ndimage import filters
from sklearn.svm import LinearSVC
import matplotlib.pyplot as plt
%matplotlib inline
```

Forward Radon-CDT

[2]:
```
# Create the signal
I = np.zeros([128, 128])
I[64, 64] = 1
I = filters.gaussian_filter(I, sigma=20)

# Perform transform
x0_range = [0, 1]
x_range = [0, 1]
template=np.ones([128, 128])
radoncdt = RadonCDT()
Ihat = radoncdt.forward(x0_range, template, x_range, I)

# Plot results
fig, (ax0, ax1) = plt.subplots(ncols=2, dpi=100)
ax0.set_title('Input singal')
ax1.set_title('Radon-CDT')
ax0.imshow(I)
ax1.imshow(Ihat)
plt.show()
```



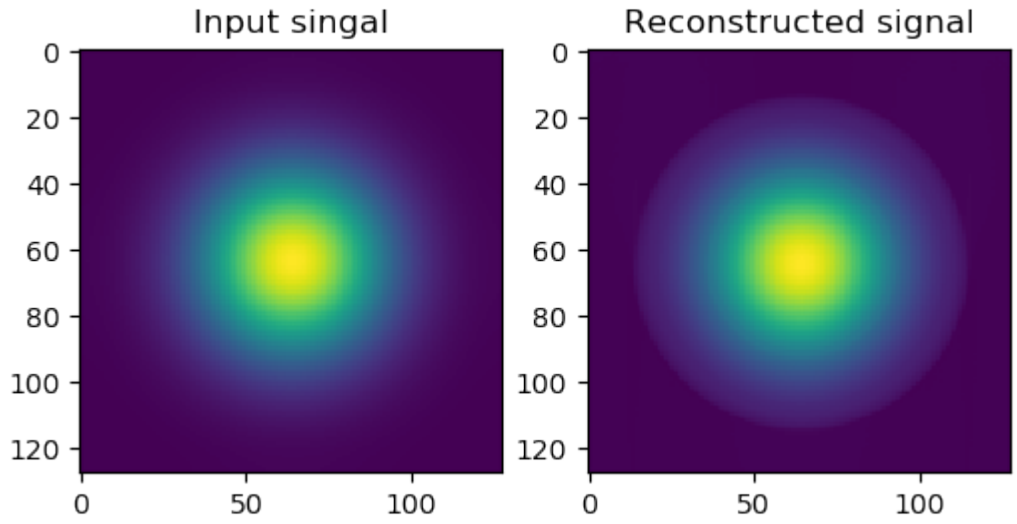
Inverse Radon-CDT

[3]:
```
Irecon = radoncdt.inverse(Ihat, template, x_range)

# Plot results
fig, (ax0, ax1) = plt.subplots(ncols=2, dpi=100)
ax0.set_title('Input singal')
ax1.set_title('Reconstructed signal')
ax0.imshow(I)
ax1.imshow(Irecon)
plt.show()
```

(continues on next page)

(continued from previous page)

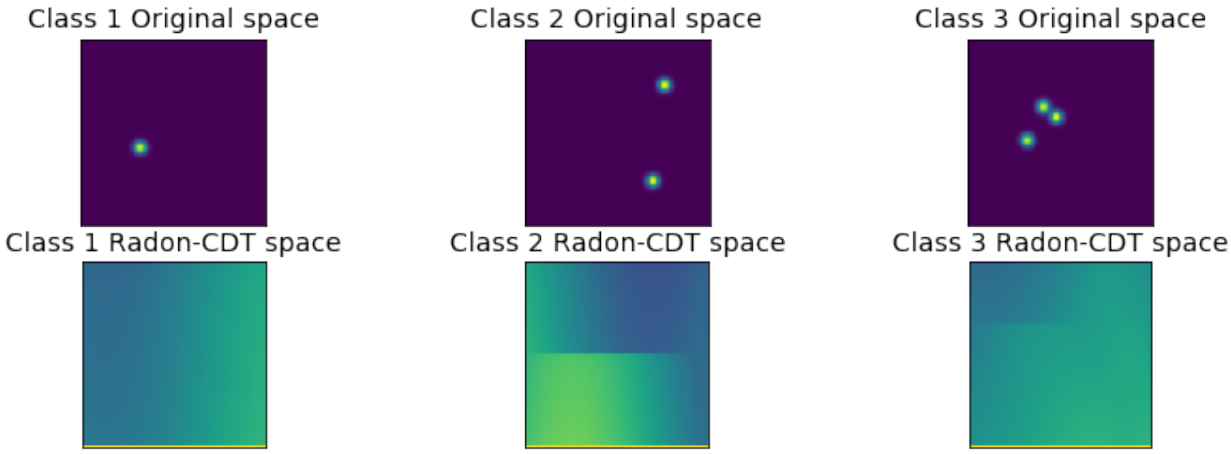
```
plt.show()
```



Classification in Radon-CDT space

[4]:
```
np.random.seed(123)
# Initialize RadonCDT object
radoncdt = RadonCDT()

# Generate 3 classes of k-modal two-dimensional Gaussians
N = 100  # Number of datapoints per class
sigma = 3  # Standard deviation of each Gaussian
# Initialize
x0_range = [0, 1]
x_range = [0, 1]
I = np.zeros((3, N, 128, 128))
template = np.ones([128, 128])
#K, L = radoncdt.forward(template, I[0, 0, :, :]+1e-8).shape
K, L = radoncdt.forward(x0_range, template, x_range, I[0, 0, :, :]+1e-8).shape
Ihat = np.zeros((3, N, K, L))
Ihat = np.random.randn(3, N, K, L)
# Generate dataset
for c in range(3):
  for i in range(N):
    for _ in range(c + 1):
      x, y = np.random.uniform(30, 98, (2,)).astype('int')
      I[c, i, x, y] = 1
    I[c, i, :, :] = I[c, i, :, :] / I[c, i, :, :].sum()
    I[c, i, :, :] = filters.gaussian_filter(I[c, i, :, :], sigma=sigma)
    #Ihat[c, i, :, :] = radoncdt.forward(template, I[c, i, :, :])
    Ihat[c, i, :, :] = radoncdt.forward(x0_range, template, x_range, I[c, i, :, :])
```

[5]:
```
fig, ax = plt.subplots(2, 3, figsize=(12, 4))
title = ['Class 1', 'Class 2', 'Class 3']
for c in range(3):
  ind = np.random.randint(low=0, high=N)
  ax[0, c].imshow(I[c, ind, :, :])
  ax[0, c].set_xticks([])
  ax[0, c].set_yticks([])
  ax[0, c].set_title(title[c] + ' Original space', fontsize=14)
  ax[1, c].imshow(Ihat[c, ind, :, :])
  ax[1, c].set_xticks([])
  ax[1, c].set_yticks([])
  ax[1, c].set_title(title[c] + ' Radon-CDT space', fontsize=14)
plt.show()
```



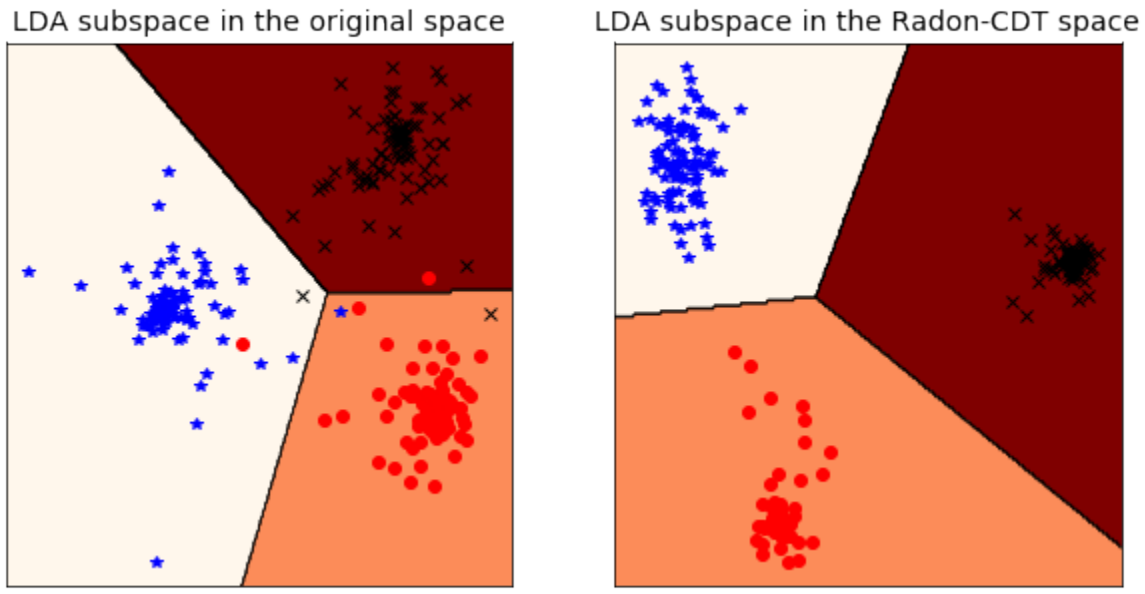
[6]:
```
X=np.reshape(I,(3*N,128*128))
Xhat=np.reshape(Ihat,(3*N,K*L))
label=np.concatenate([np.ones(N), np.ones(N)+1, np.ones(N)+2])
lda=LDA(solver='svd', n_components=2)
n=N
# Apply LDA  in Signal Space
Xlda=lda.fit_transform(X,label)
svm=LinearSVC()
svm.fit(Xlda,label)
# Apply LDA in transform space
Xhatlda=lda.fit_transform(Xhat,label)
svmhat=LinearSVC()
svmhat.fit(Xhatlda,label)
# Show classification boundaries in both Spaces
svmClassifier=[svm,svmhat]
Xdata=[Xlda,Xhatlda]
title=['LDA subspace in the original space','LDA subspace in the Radon-CDT space']
fig,ax=plt.subplots(1,2,figsize=(10,5))
for i in range(2):
    ax[i].plot(Xdata[i][:n,0],Xdata[i][:n,1],'b*')
    ax[i].plot(Xdata[i][n:2*n,0],Xdata[i][n:2*n,1],'ro')
    ax[i].plot(Xdata[i][2*n:,0],Xdata[i][2*n:,1],'kx')
```

(continues on next page)

(continued from previous page)

```
    x_min, x_max = ax[i].get_xlim()
    y_min, y_max = ax[i].get_ylim()
    nx, ny = 400, 200
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, nx),
                         np.linspace(y_min, y_max, ny))
    Z = svmClassifier[i].predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z[:].reshape(xx.shape)

    ax[i].pcolormesh(xx, yy, Z,cmap='OrRd')
    ax[i].contour(xx, yy, Z, linewidths=.5, colors='k')
    ax[i].set_title(title[i],fontsize=14)
    ax[i].set_xticks([])
    ax[i].set_yticks([])
```

```
/Users/ar3fx/anaconda3/anaconda3/lib/python3.7/site-packages/sklearn/discriminant_
↪analysis.py:388: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
/Users/ar3fx/anaconda3/anaconda3/lib/python3.7/site-packages/sklearn/svm/base.py:929:␣
↪ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  "the number of iterations.", ConvergenceWarning)
/Users/ar3fx/anaconda3/anaconda3/lib/python3.7/site-packages/sklearn/discriminant_
↪analysis.py:388: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
```



[ ]:

2.3 3D Radon-Cumulative Distribution Transform (3D R-CDT)

This tutorial will demonstrate: how to use the forward and inverse operations of the 3D R-CDT in the *PyTransKit* package.

2.3.1 Class:: RadonCDT3D

Parameters:

```
Npoints : scaler, number of radon projections (default = 1024)
use_gpu : boolean, use GPU if True (default = False)
```

Functions:

1. Forward transform: sig1_hat = forward(x0_range, sig0, x1_range, sig1)

```
Inputs:
----------------
x0_range : 1x2 array
    contains lower and upper limits for independent variable of reference signal␣
↪(sig0). Example: [0,1].
sig0 : 3d array, shape (L, L, L)
    Reference signal.
x1_range : 1x2 array
    contains lower and upper limits for independent variable of input signal (sig1).
↪ Example: [0,1].
sig1 : 3d array, shape (L, L, L)
    Signal to transform.

Outputs:
----------------
sig1_hat : 2d array, shape(L, Npoints)
    R-CDT of 3D input signal sig1.
```

2. Inverse transform: sig1_recon = inverse(sig1_hat, sig0, x1_range)

```
Inputs:
----------------
sig1_hat : 2d array, shape(L, Npoints)
    R-CDT of 3D signal sig1.
sig0 : 3d array, shape (L, L, L)
    Reference signal.
x1_range : 1x2 array
    contains lower and upper limits for independent variable of input signal (sig1).
↪ Example: [0,1].

Outputs:
----------------
sig1_recon : 3d array, shape (L, L, L)
    Reconstructed signal.
```

2.3.2 Example

The example will cover the following operations: * Forward and inverse operations of the 3D R-CDT

[1]:
```
import sys
sys.path.append('../')
from pytranskit.optrans.continuous.radoncdt3D import RadonCDT3D

import time
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
%matplotlib inline
```

Forward 3D Radon-CDT

Load 3D phantom data from 'images/' direcotry

[2]:
```
datafile = 'images/phantom3D.mat'
img3D = loadmat(datafile)['phantom3D']
i03D = np.ones(img3D.shape)
```

Run forward 3D R-CDT on the 3D phantom data

[3]:
```
use_gpu = False  # Set it True if want GPU support
x0_range = [0,1]
x_range = [0,1]
Npoints = 1000

# Create an instance of 3D R-CDT
rcdt3D = RadonCDT3D(Npoints, use_gpu)

tic=time.time()
# Forward function
img3Dhat=rcdt3D.forward(x0_range, i03D/np.sum(img3D), x_range, img3D/np.sum(img3D), rm_
↪edge=False)

toc=time.time()
Run_time = toc - tic
print("Forward 3D RCDT is done in {} seconds".format(Run_time))
```

```
Forward 3D RCDT is done in 19.50796389579773 seconds
```

Inverse 3D Radon-CDT

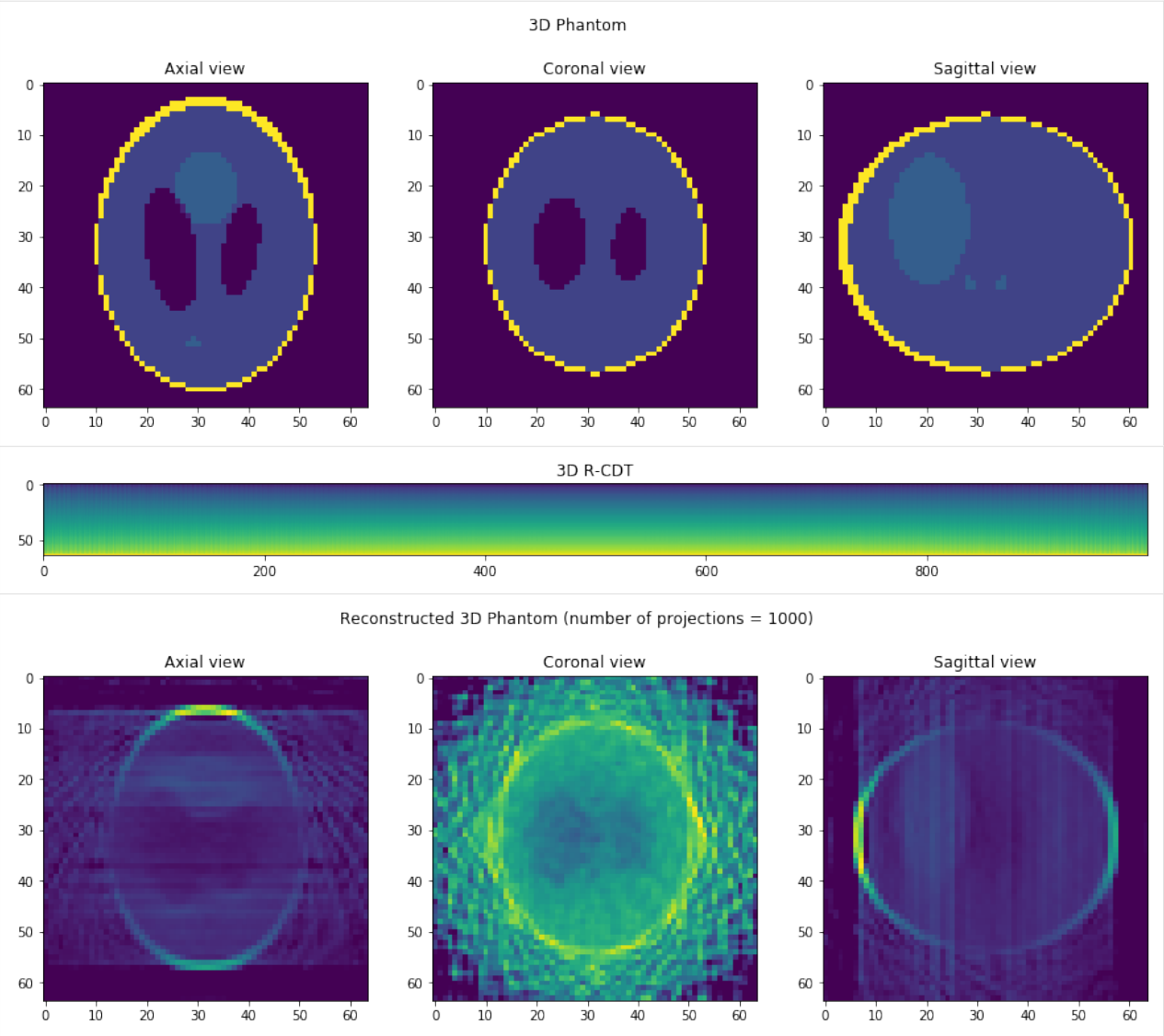
[4]:
```
tic=time.time()
# Inverse function
img3D_recon = rcdt3D.inverse(img3Dhat, i03D, x_range)
toc=time.time()
Run_time = toc - tic
print("Inverse 3D RCDT is done in {} seconds".format(Run_time))
```

```
Inverse 3D RCDT is done in 20.88492465019226 seconds
```

Show results

Show specific slices from the input 3D phantom image and reconstructed image

[5]:
```
sliceSel = int(0.5*img3D.shape[0])

# plot original 3D image
plt.figure(figsize=(15,5))
plt.suptitle('3D Phantom')
plt.subplot(131)
plt.imshow(img3D[sliceSel,:,:])
plt.title('Axial view')

plt.subplot(132)
plt.imshow(img3D[:,sliceSel,:])
plt.title('Coronal view')

plt.subplot(133)
plt.imshow(img3D[:,:,sliceSel])
plt.title('Sagittal view')
plt.show()

# plot 3D Radon transform
fig=plt.figure(figsize=(15,5))
plt.imshow(img3Dhat)
plt.title('3D R-CDT')
plt.show()

# plot 3D reconstruction
plt.figure(figsize=(15,5))
plt.suptitle('Reconstructed 3D Phantom (number of projections = {})'.format(Npoints))
plt.subplot(131)
plt.imshow(np.log10(.1+img3D_recon[sliceSel,:,:]))
plt.title('Axial view')

plt.subplot(132)
plt.imshow(np.log10(.1+img3D_recon[:,sliceSel,:]))
plt.title('Coronal view')

plt.subplot(133)
plt.imshow(np.log10(.1+img3D_recon[:,:,sliceSel]))
plt.title('Sagittal view')
plt.show()
```



[ ]:

2.4 Continuous Linear Optimal Transport Transform (CLOT)

This tutorial will demonstrate: how to use the forward and inverse operations of the CLOT in the the *PyTransKit* package.

2.4.1 Class:: CLOT

```
Continuous Linear Optimal Transport Transform.

Parameters
----------
lr : float (default=0.01)
    Learning rate.
momentum : float (default=0.)
    Nesterov accelerated gradient descent momentum.
decay : float (default=0.)
    Learning rate decay over each update.
max_iter : int (default=300)
    Maximum number of iterations.
tol : float (default=0.001)
    Stop iterating when change in cost function is below this threshold.
verbose : int (default=1)
    Verbosity during optimization. 0=no output, 1=print cost,
    2=print all metrics.

Attributes
-----------
displacements_ : array, shape (2, height, width)
    Displacements u. First index denotes direction: displacements_[0] is
    y-displacements, and displacements_[1] is x-displacements.
transport_map_ : array, shape (2, height, width)
    Transport map f. First index denotes direction: transport_map_[0] is
    y-map, and transport_map_[1] is x-map.
displacements_initial_ : array, shape (2, height, width)
    Initial displacements computed using the method by Haker et al.
transport_map_initial_ : array, shape (2, height, width)
    Initial transport map computed using the method by Haker et al.
cost_ : list of float
    Value of cost function at each iteration.
curl_ : list of float
    Curl at each iteration.

References
----------
[A continuous linear optimal transport approach for pattern analysis in
image datasets]
(https://www.sciencedirect.com/science/article/pii/S0031320315003507)
[Optimal mass transport for registration and warping]
(https://link.springer.com/article/10.1023/B:VISI.0000036836.66311.97)
```

2.4.2 Functions:

1. Forward transform: lot = forward(sig0, sig1)

```
Inputs:
----------------
sig0 : array, shape (height, width)
    Reference image.
sig1 : array, shape (height, width)
    Signal to transform.

Outputs:
----------------
lot : array, shape (2, height, width)
    LOT transform of input image sig1. First index denotes direction:
    lot[0] is y-LOT, and lot[1] is x-LOT.
```

2. Apply forward transport map: sig0_recon = apply_forward_map(transport_map, sig1)

```
Inputs:
----------------
transport_map : array, shape (2, height, width)
    Forward transport map.
sig1 : array, shape (height, width)
    Signal to transform.

Outputs:
----------------
sig0_recon : array, shape (height, width)
    Reconstructed reference signal sig0.
```

3. Apply inverse transport map: sig1_recon = inverse(transport_map, sig0)

```
Inputs:
----------------
transport_map : array, shape (2, height, width)
    Forward transport map. Inverse is computed in this function.
sig0 : array, shape (height, width)
    Reference signal.

Outputs:
----------------
sig1_recon : array, shape (height, width)
    Reconstructed signal sig1.
```

2.4.3 Definition

The Continuous Linear Optimal Transport (CLOT) transform $\widehat{s}$ of a density function $s(\mathbf{x})$ is defined as the optimal transport map from a reference density $s_0(\mathbf{x})$ to $s(\mathbf{x})$. Specifically, let $s_0(\mathbf{x}), s(\mathbf{x})$ be positive functions defined on domains $\Omega_{s_0}, \Omega_s \subseteq \mathbb{R}^d$ respectively and such that

$$\int_{\Omega_{s_0}} s_0(\mathbf{x})d\mathbf{x} = \int_{\Omega_s} s(\mathbf{x})d\mathbf{x} = 1 \quad \text{(normalized)}.$$

Assuming that the density functions s_0, s have finite second moments, there is an unique solution to the Monge optimal transport problem:

$$\text{min Monge } (T) = \int_{R^d} \big|x - T(x)\big|^2 s_0(x)dx, \qquad (1) \tag{2.8}$$

$$\text{s.t.} \quad \int_B s(\mathbf{y})d\mathbf{y} = \int_{T^{-1}(B)} s_0(\mathbf{x})d\mathbf{x}, \quad \text{for all open } B \subseteq \mathbb{R}^d. \qquad (2) \tag{2.9}$$

Any map T satisfying constraint in (2) is called a transport (mass-preserving) map between s_0 and s. In particular, when T is bijective and continuously differentiable, the mass-preserving constraint in (2) becomes

$$s_0(x) = \big|\det\big(\nabla T(x)\big)\big| s\big(T(x)\big). \tag{2.10}$$

The minimizer to the above Monge problem is called an optimal transport map. Given a fixed reference density s_0, the LOT transform $\widehat{s}$ of a density function s is defined to the unique optimal transport map from s_0 to s. Moreover Brenier [1] shows that any optimal transport map can be written as the gradient of a convex function, i.e., $\widehat{s} = \nabla\phi$ where ϕ is a convex function. Following the generic approach described in [2], Kolouri et al. [3] employed an iterative algorithm minimizing (1) with constraint (2) via the gradient descent idea.

References

[1] Y. Brenier. Polar factorization and monotone rearrangement of vector-valuedfunctions.Commun. Pure Appl. Math., 44(4):375–417, 1991.1 [2] S. Haker, L. Zhu, A. Tannenbaum, and S. Angenent. Optimal mass transport forregistration and warping.Int. J. Comput. Vis., 60(4):225–240, 2004. [3] S. Kolouri, A. Tosun, J. Ozolek, and G. Rohde. A continuous linear optimal trans-port approach for pattern analysis in image datasets.Pattern Recognit., 51:453–462, 2016.

2.4.4 CLOT Demo

The examples will cover the following operations: * Forward operation of the CLOT * Apply forward map to transport I_1 to I_0 * Apply inverse map to reconstruct I_1 from I_0

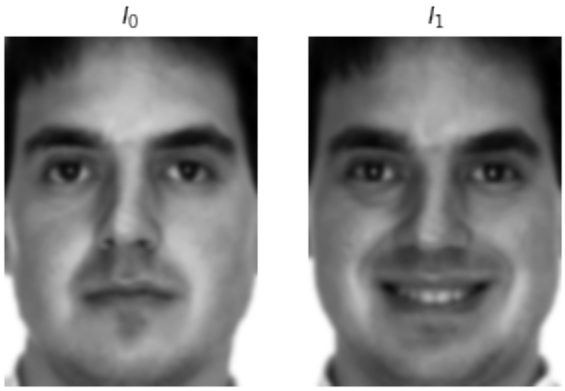
2.4.5 Forward CLOT

Import necessary python packages

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Read and normalize two images I_0 and I_1.

[2]:
```
import matplotlib.image as mpimg
import sys
sys.path.append('../')
from pytranskit.optrans.utils import signal_to_pdf

I0 = mpimg.imread('images/I0.bmp')
I1 = mpimg.imread('images/I1.bmp')

# Convert images to PDFs
img0 = signal_to_pdf(I0, sigma=1., total=100.)
img1 = signal_to_pdf(I1, sigma=1., total=100.)

fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(5,10))
ax[0].imshow(img0,cmap='gray')
ax[1].imshow(img1,cmap='gray')
ax[0].set_title('$I_0$')
ax[1].set_title('$I_1$')
ax[0].axis('off')
ax[1].axis('off')
plt.show()
```



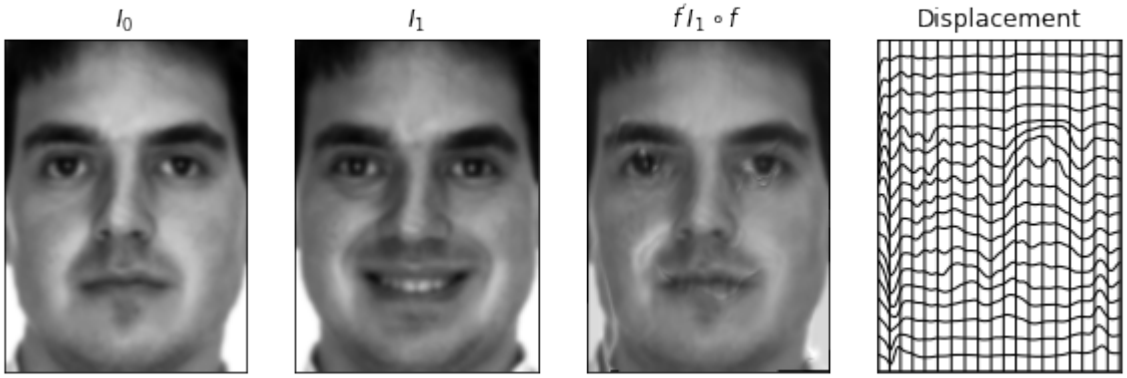
Compute CLOT and apply forward map

[7]:
```
from pytranskit.optrans.continuous.clot import CLOT
from pytranskit.optrans.utils import plot_displacements2d

clot = CLOT(max_iter=500, lr=1e-6, tol=1e-4,verbose=0)

# calculate CLOT
lot = clot.forward(img0, img1)

# transport map and displacement map from I1 to I0
tmap10 = clot.transport_map_
disp = clot.displacements_

# apply forward map to transport I1 to I0
```

(continues on next page)

(continued from previous page)

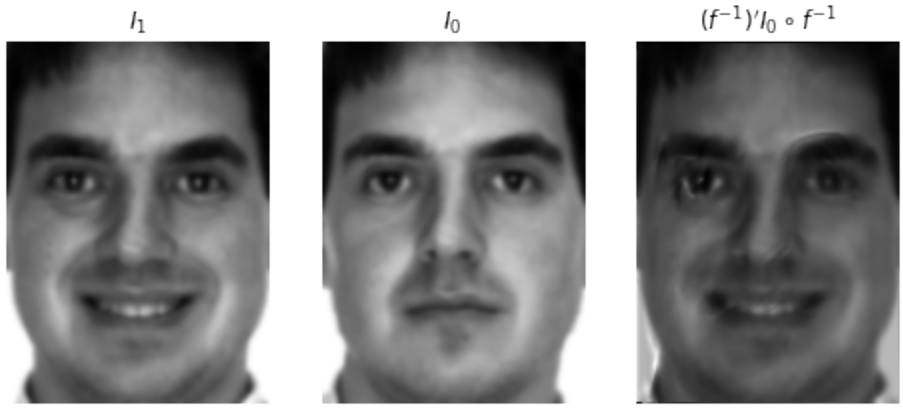
```
img0_recon = clot.apply_forward_map(tmap10, img1)

fig, ax = plt.subplots(1, 4, sharex=True, sharey=True, figsize=(10,20))
ax[0].imshow(img0, cmap='gray')
ax[0].set_title('$I_0$')

ax[1].imshow(img1, cmap='gray')
ax[1].set_title('$I_1$')

ax[2].imshow(img0_recon, cmap='gray')
ax[2].set_title('$f^{\'}I_1\circ f$')

plot_displacements2d(disp, ax=ax[3], count=20)
ax[3].set_title('Displacement')

plt.show()
```



2.4.6 Inverse CLOT

Apply inverse map on I_0 to reconstruct I_1

[4]:
```
img1_recon = clot.apply_inverse_map(tmap10, img0)

fig, ax = plt.subplots(1, 3, sharex=True, sharey=True, figsize=(8,15))
ax[0].imshow(img1, cmap='gray')
ax[0].set_title('$I_1$')

ax[1].imshow(img0, cmap='gray')
ax[1].set_title('$I_0$')

ax[2].imshow(img1_recon, cmap='gray')
ax[2].set_title('$(f^{-1})\'I_0\circ f^{-1}$')

ax[0].axis('off')
ax[1].axis('off')
ax[2].axis('off')

plt.show()
```

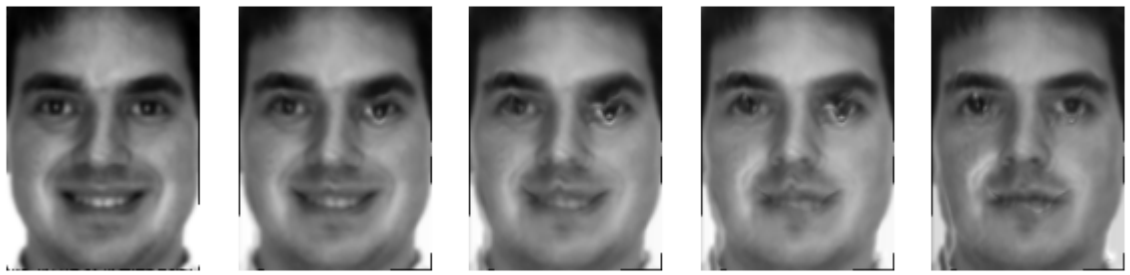


2.4.7 Geodesic

Show points on the geodesic between I_0 and I_1

[5]:
```
lot11 = clot.forward(img1, img1)
tmap11 = clot.transport_map_

alpha = np.linspace(0,1,5)
img_recon = []
fig, ax = plt.subplots(1, len(alpha), sharex=True, sharey=True, figsize=(10,
↪5*len(alpha)))
for i in range(len(alpha)):
    tmap = alpha[i]*tmap10 + (1-alpha[i])*tmap11
    img_recon.append(clot.apply_forward_map(tmap, img1))
    ax[i].imshow(img_recon[i],cmap='gray')
    ax[i].axis('off')
plt.show
```

[5]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



[ ]:

2.5 Cumulative Distribution Transform Nearest Subspace (CDT-NS) Classifier

This tutorial will demonstrate how to use the CDT-NS classifier for 1D data in the *PyTransKit* package.

2.5.1 Class:: CDT_NS

Functions:

1. Constructor function: cdt_ns_obj = CDT_NS(num_classes, rm_edge)

```
Inputs:
----------------
num_classes : integer value
    totale number of classes in the dataset.
rm_edge : boolean
    IF TRUE the first and last points of CDTs will be removed.

Outputs:
----------------
cdt_ns_obj : class object
    Instance of the class CDT_NS.
```

2. Fit function: cdt_ns_obj.fit(Xtrain, Ytrain, no_deform_model)

```
Inputs:
----------------
Xtrain : array-like, shape (n_samples, n_columns)
    1D data for training.
Ytrain : ndarray of shape (n_samples,)
    Labels of the training samples.
no_deform_model : boolean flag; IF TRUE, no deformation model will be added
    default = False.
```

3. Predict function: preds = cdt_ns_obj.predict(Xtest, use_gpu)

```
Inputs:
----------------
Xtest : array-like, shape (n_samples, n_columns)
    1D data for testing.
use_gpu: boolean flag; IF TRUE, use gpu for calculations
    default = False.

Outputs:
----------------
preds : 1d array, shape (n_samples,)
   Predicted labels for test samples.
```

2.5.2 Example

The following example will demonstrate how to: * create and initialize an instance of the class CDT_NS * train the model with training 1D samples * apply the model to predict calss labels of the test 1D samples In this example we have used a synthetic dataset (1D) stored in the *data* folder. The dataset contains two classes. Class 0: different translated versions of Gaussian signal Class 1: translated versions of summation of two Gaussian signals

Import some python libraries

[1]:
```
import numpy as np
from sklearn.metrics import accuracy_score
from pathlib import Path
import sys
sys.path.append('../')
from pytranskit.classification.utils import *

use_gpu = False
```

Import CDT-NS class from *PyTransKit* package

[2]:
```
from pytranskit.classification.cdt_ns import CDT_NS
```

Load dataset

For loading data we have used *load_data_1D* function from the *pytranskit/classifier/utils.py* script. It takes name and directory of the dataset, and total number of classes as input. Returns both train and test samples in two separate 2d arrays of shape (n_samples, n_columns), and corresponding class labels. User can use there own implementation to load data, just need to make sure that the output arrays are consistent.

[3]:
```
datadir = './data'
dataset = 'synthetic_1D'
num_classes = 2          # total number of classes in the dataset
(x_train, y_train), (x_test, y_test) = load_data_1D(dataset, num_classes, datadir)  #␣
↪load_data function from utils.py
```

```
loading data from mat files
x_train.shape (1400, 201) x_test.shape (600, 201)
saved to ./data/synthetic_1D/dataset.hdf5
```

In this example we have used 512 randomly chosen samples per class to train the model. We have used another function *take_train_samples* function from *utils.py* script for this. User can use their own script.

[4]:
```
n_samples_perclass = 512  # total number of training samples per class used in this␣
↪example
x_train_sub, y_train_sub = take_train_samples(x_train, y_train, n_samples_perclass,
                                              num_classes, repeat=0) # function from␣
↪utils.py
```

Create an instance of CDT_NS class

[5]:
```
cdt_ns_obj = CDT_NS(num_classes, rm_edge=True)
```

Training phase

This function takes the train samples and labels as input, and stores the basis vectors for corresponding classes in a *private* variable. This variable will be used in the *predict* function in the test phase

[6]:
```
print(x_train_sub.shape)
cdt_ns_obj.fit(x_train_sub, y_train_sub)
```

```
(1024, 201)

Calculating CDTs for training data ...
Generating basis vectors for each class ...
```

Testing phase

predict function takes the train samples as input and returns the predicted class labels

[7]:
```
preds = cdt_ns_obj.predict(x_test, use_gpu)
```

```

Calculating CDTs for testing samples ...
Finding nearest subspace for each test sample ...
```

[8]:
```
print('\nTest accuracy: {}%'.format(100*accuracy_score(y_test, preds)))
```

```

Test accuracy: 100.0%
```

[ ]:
```

```

2.6 Radon-Cumulative Distribution Transform Nearest Subspace (RCDT-NS) Classifier

This tutorial will demonstrate how to use the RCDT-NS classifier in the *PyTransKit* package.

2.6.1 Class:: RCDT_NS

Functions:

1. Constructor function: rcdt_ns_obj = RCDT_NS(num_classes, thetas, rm_edge)

```
Inputs:
----------------
num_classes : integer value
    totale number of classes in the dataset.
```

(continues on next page)

(continued from previous page)

```
thetas : 1d array
    angles in degrees for taking radon projections. Example: thetas=numpy.
↪linspace(0,180,45)
rm_edge : boolean
    IF TRUE the first and last points of RCDTs will be removed.

Outputs:
----------------
rcdt_ns_obj : class object
    Instance of the class RCDT_NS.
```

2. Fit function: rcdt_ns_obj.fit(Xtrain, Ytrain, no_deform_model)

```
Inputs:
----------------
Xtrain : 3d array, shape (n_samples, n_rows, n_columns)
    Image data for training.
Ytrain : 1d array, shape (n_samples,)
    Labels of the training images.
no_deform_model : boolean
    IF TRUE, no deformation model will be added
```

3. Predict function: preds = rcdt_ns_obj.predict(Xtest, use_gpu)

```
Inputs:
----------------
Xtest : 3d array, shape (n_samples, n_rows, n_columns)
    Image data for testing.
use_gpu : boolean
    IF TRUE, use gpu for calculations.

Outputs:
----------------
preds : 1d array, shape (n_samples,)
   Predicted labels for test samples.
```

2.6.2 Example

The following example will demonstrate how to: * create and initialize an instance of the class RCDT_NS * train the model with training images * apply the model to predict calss labels of the test images In this example we have used MNIST dataset stored in the *data* folder

Import some python libraries

[1]:
```
import numpy as np
from sklearn.metrics import accuracy_score
from pathlib import Path
import sys
sys.path.append('../')
from pytranskit.classification.utils import *

use_gpu = False
```

Import RCDT-NS class from *PyTransKit* package

[2]:
```
from pytranskit.classification.rcdt_ns import RCDT_NS
```

Load dataset

For loading data we have used *load_data* function from the *pytranskit/classifier/utils.py* script. It takes name and directory of the dataset, and total number of classes as input. Returns both train and test images in two separate 3d arrays of shape (n_samples, n_rows, n_columns), and corresponding class labels. User can use there own implementation to load data, just need to make sure that the output arrays are consistent.

[3]:
```
datadir = './data'
dataset = 'MNIST'
num_classes = 10          # total number of classes in the dataset
(x_train, y_train), (x_test, y_test) = load_data(dataset, num_classes, datadir)  # load_
↪data function from utils.py
```

```
loading data from mat files
x_train.shape (60000, 28, 28) x_test.shape (10000, 28, 28)
saved to ./data/MNIST/dataset.hdf5
```

In this example we have used 512 randomly chosen samples per class to train the model. We have used another function *take_train_samples* function from *utils.py* script for this. User can use their own script.

[4]:
```
n_samples_perclass = 512  # total number of training samples per class used in this␣
↪example
x_train_sub, y_train_sub = take_train_samples(x_train, y_train, n_samples_perclass,
                                              num_classes, repeat=0) # function from␣
↪utils.py
```

Create an instance of RCDT_NS class

[5]:
```
theta = np.linspace(0, 176, 45)    # choose the angles in degrees that will be used to␣
↪calculate Radon projections
rcdt_ns_obj = RCDT_NS(num_classes, theta, rm_edge=True)
```

Training phase

This function takes the train samples and labels as input, and stores the basis vectors for corresponding classes in a *private* variable. This variable will be used in the *predict* function in the test phase

[6]:
```
rcdt_ns_obj.fit(x_train_sub, y_train_sub)
```

```
Calculating RCDTs for training images ...
Generating basis vectors for each class ...
```

Testing phase

predict function takes the train samples as input and returns the predicted class labels

[7]:
```
preds = rcdt_ns_obj.predict(x_test, use_gpu)
```

```
Calculating RCDTs for testing images ...
Finding nearest subspace for each test sample ...
```

[8]:
```
print('\nTest accuracy: {}%'.format(100*accuracy_score(y_test, preds)))
```

```
Test accuracy: 95.41%
```

[ ]:
```
```

2.7 3D Radon-Cumulative Distribution Transform Nearest Subspace (3D-RCDT-NS) Classifier

This tutorial will demonstrate how to use the 3D-RCDT-NS classifier in the *PyTransKit* package.

2.7.1 Class:: RCDT_NS_3D

Functions:

1. Constructor function: rcdt_ns_obj = RCDT_NS(num_classes, thetas, rm_edge)

```
Inputs:
----------------
num_classes : integer value
    totale number of classes in the dataset.
Npoints : scalar; number of radon projections
use_gpu : boolean; IF TRUE, use GPU to calculate 3D RCDT
rm_edge : boolean
    IF TRUE, the first and last points of RCDTs will be removed.

Outputs:
----------------
```

(continues on next page)

(continued from previous page)

```
rcdt_ns_obj : class object
    Instance of the class RCDT_NS.
```

2. Fit function: rcdt_ns_obj.fit(Xtrain, Ytrain, no_deform_model)

```
Inputs:
----------------
Xtrain : 4d array, shape (n_samples, L, L, L)
    3D Image data for training. L is the dimension along X,Y, and Z axes.
Ytrain : 1d array, shape (n_samples,)
    Labels of the training images.
no_deform_model : boolean
    IF TRUE, no deformation model will be added
```

3. Predict function: preds = rcdt_ns_obj.predict(Xtest, use_gpu)

```
Inputs:
----------------
Xtest : 4d array, shape (n_samples, L, L, L)
    3D Image data for testing. L is the dimension along X,Y, and Z axes.
use_gpu : boolean
    IF TRUE, use gpu for calculations.

Outputs:
----------------
preds : 1d array, shape (n_samples,)
   Predicted labels for test samples.
```

2.7.2 Example

The following example will demonstrate how to: * create and initialize an instance of the class 3D-RCDT_NS * train the model with training images * apply the model to predict calss labels of the test images In this example we have used MNIST dataset stored in the *data* folder

Import some python libraries

[1]:
```
import numpy as np
from sklearn.metrics import accuracy_score
from pathlib import Path
import sys
sys.path.append('../')
from pytranskit.classification.utils import *

use_gpu = True
```

Import 3D-RCDT-NS class from *PyTransKit* package

[2]:
```
from pytranskit.classification.rcdt_ns_3d import RCDT_NS_3D
```

Load dataset

For loading data we have used *load_data_3D* function from the *pytranskit/classifier/utils.py* script. It takes name and directory of the dataset, and total number of classes as input. Returns both train and test images in two separate 4d arrays of shape (n_samples, n_rows, n_columns, n_columns), and corresponding class labels. User can use there own implementation to load data, just need to make sure that the output arrays are consistent. In this example, we have used a synthetic 3D dataset with two classes: class 0 contains one Gaussian blob in each image, class 1 contains two Gaussian blobs in each image. Note: The 3D RCDT implemented in PyTransKit, 3D images need to be equal shape along all three directions, i.e. *n_rows=n_columns=n_columns=L*. Therefore, if the original image does not have equal length in all axes, users need to zero pad to make all the dimensions equal.

[3]:
```
datadir = './data'
dataset = 'synthetic_3D'
num_classes = 2          # total number of classes in the dataset
(x_train, y_train), (x_test, y_test) = load_data_3D(dataset, num_classes, datadir)  #␣
↪load_data function from utils.py
```

```
loading data from mat files
split training class 0 data.shape (50, 32, 32, 32)
split training class 1 data.shape (50, 32, 32, 32)
split testing class 0 data.shape (50, 32, 32, 32)
split testing class 1 data.shape (50, 32, 32, 32)
x_train.shape (100, 32, 32, 32) x_test.shape (100, 32, 32, 32)
saved to ./data/synthetic_3D/dataset.hdf5
```

In this example we have used 32 randomly chosen samples per class to train the model. We have used another function *take_train_samples* function from *utils.py* script for this. User can use their own script.

[4]:
```
n_samples_perclass = 32  # total number of training samples per class used in this␣
↪example
x_train_sub, y_train_sub = take_train_samples(x_train, y_train, n_samples_perclass,
                                              num_classes, repeat=0) # function from␣
↪utils.py
```

Create an instance of 3D-RCDT-NS class

[5]:
```
Npoints = 500    # choose number projections 3D Radon transform
rcdt_ns_obj = RCDT_NS_3D(num_classes, Npoints, use_gpu, rm_edge=True)
```

Training phase

This function takes the train samples and labels as input, and stores the basis vectors for corresponding classes in a *private* variable. This variable will be used in the *predict* function in the test phase

[6]:
```
rcdt_ns_obj.fit(x_train_sub, y_train_sub)
```

```
Calculating RCDTs for training images ...
Generating basis vectors for each class ...
```

Testing phase

predict function takes the train samples as input and returns the predicted class labels

[7]:
```
preds = rcdt_ns_obj.predict(x_test, use_gpu)
```

```
Calculating RCDTs for testing images ...
Finding nearest subspace for each test sample ...
```

[8]:
```
print('\nTest accuracy: {}%'.format(100*accuracy_score(y_test, preds)))
```

```
Test accuracy: 98.0%
```

[ ]:
```
```

CHAPTER

THREE

EXAMPLES

3.1 Estimation of time delay

The signal of interest is

$$z(t) = Ae^{-\frac{(t-t_c)^2}{2b_w^2}} \sin(2\pi f t) \tag{3.1}$$

of width b_w and frequency f. In this example we will take $A = b_w = f = 1$ and set $t_c = 0$. The probability density function (PDF),

$$s(t) = B(z)(t) := \frac{z^2(t)}{\int_{\Omega_s} z^2(t)dt} \tag{3.2}$$

In this example, we are interested in estimating the time delay (τ), i.e. $g_p(t) = t - \tau$. Therefore $z_g(t) = z(t - \tau)$ and $s_g(t) = B(z_g)(t)$.

Normalized measured signal with noise $\eta \sim \mathcal{N}(0, \sigma^2)$ is given by,

$$r(t) = B(z_g + \eta)(t) \tag{3.3}$$

3.1.1 Parameter estimation in the CDT domain

Let, $\hat{s}$ and $\hat{r}$ be the CDTs of $s(t)$ and $r(t)$, respectively. The time delay estimate is calculated as,

$$\widetilde{\tau} = \mu_r - \mu_s \tag{3.4}$$

where μ_s and μ_r are center of mass of signals s and r, respectively.

Create $s(t)$ and $r(t)$

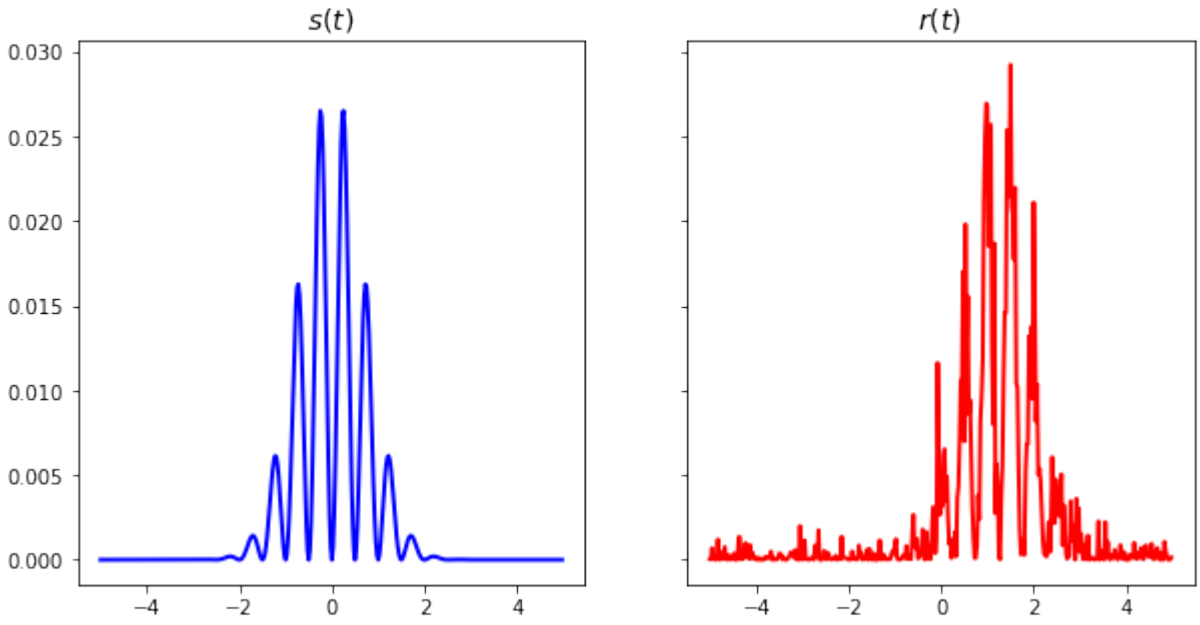
[21]:
```
import numpy as np
import matplotlib.pyplot as plt

# To use the CDT first install the PyTransKit package using:
# pip install pytranskit
from pytranskit.optrans.continuous.cdt import CDT

N = 400
dt = 0.025
t = np.linspace(-N/2*dt, (N/2-1)*dt, N)
```

(continues on next page)

(continued from previous page)

```
f = 1
epsilon = 1e-8

# Original signal
gwin = np.exp(-t**2/2)
z = gwin*np.sin(2*np.pi*f*t) + epsilon
s = z**2/np.linalg.norm(z)**2

# zero mean additive Gaussian noise
sigma = 0.1          # standard deviation
SNRdb = 10*np.log10(np.mean(z**2)/sigma**2)
print('SNR: {} dB'.format(SNRdb))
noise = np.random.normal(0,sigma,N)

# Signal after delay
tau = 50.3*dt
gwin = np.exp(-(t - tau)**2/2)
zg = gwin*np.sin(2*np.pi*f*(t - tau)) + noise + epsilon
r = zg**2/np.linalg.norm(zg)**2

# Plot s and r
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, s, 'b-',linewidth=2)
ax[0].set_title('$s(t)$',fontsize=fontSize)

ax[1].plot(t, r, 'r-',linewidth=2)
ax[1].set_title('$r(t)$',fontsize=fontSize)

plt.show()
```

```
SNR: 9.47544940682685 dB
```



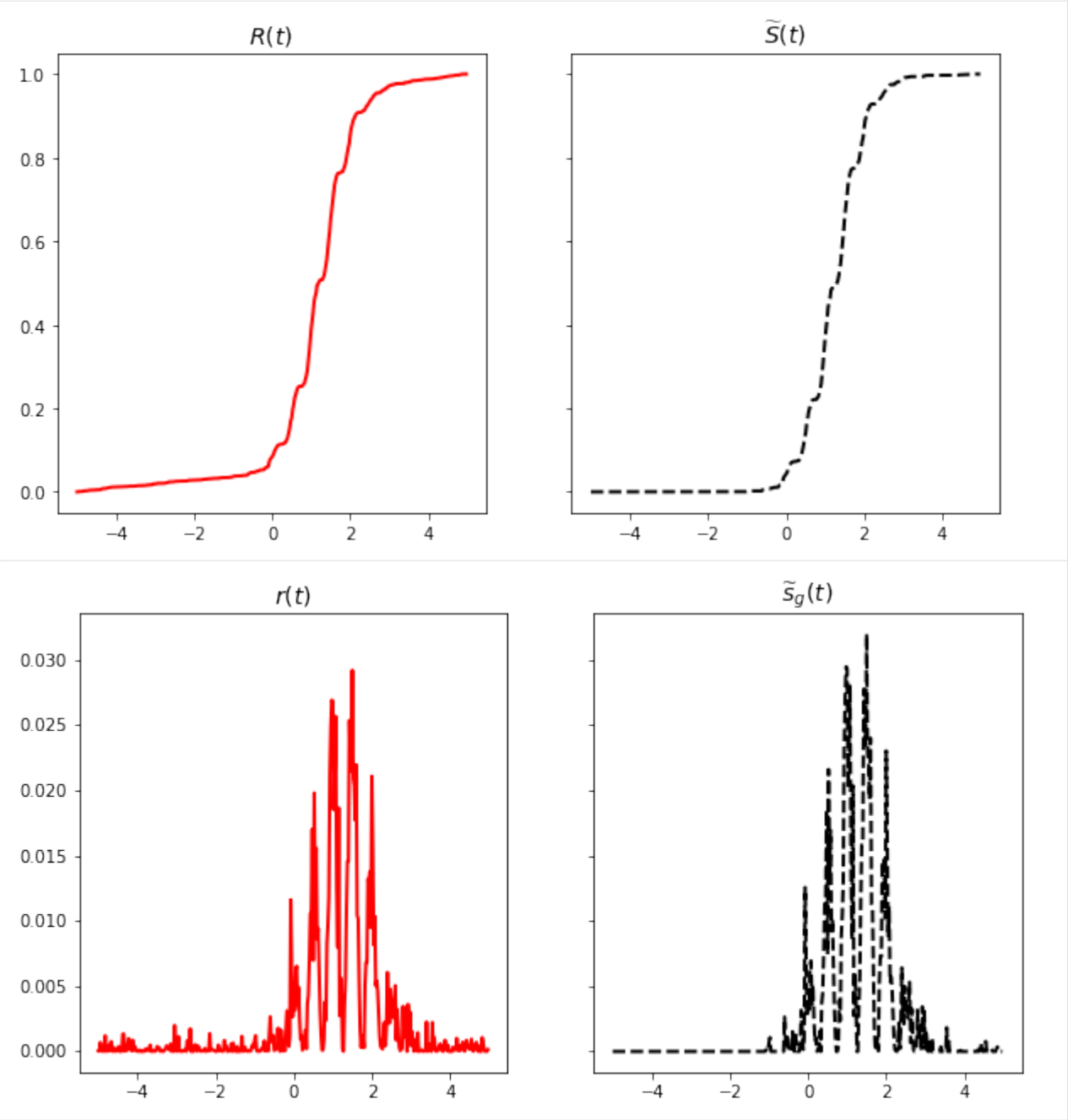
Noise correction

Expression of noise-corrected CDF is given by,

$$\tilde{S}_g(t) = \frac{E[R(t)]\{\mathcal{E}_z + \sigma^2(t_N - t_1)\} - \sigma^2(t - t_1)}{\mathcal{E}_z}, \quad t_1 \leq t \leq t_N \tag{3.5}$$

where $R(t)$ is the CDF associated with $r(t)$, σ is the standard deviation of noise, and $\mathcal{E}_z$ is the total energy of the noise free signal.

[22]:
```
# Calculate the noise-corrected CDF
R = np.cumsum(r)                    # CDF of r(t)
Ez = np.mean(z**2)*(t[-1] - t[0])  # Energy of the signal
Stilde = (R*(Ez+sigma**2*(t[-1]-t[0])) - sigma**2*(t-t[0]))/Ez

# Preserve the non-decreasing property of CDFs
mind = np.argmin(Stilde)
Mind = np.argmax(Stilde)
Stilde[0:mind] = Stilde[mind]
Stilde[Mind:] = Stilde[Mind]
for i in range(len(Stilde)-1):
    if Stilde[i+1]<=Stilde[i]:
        Stilde[i+1] = Stilde[i] + epsilon

Stilde = (Stilde - np.min(Stilde))/(np.max(Stilde) - np.min(Stilde))

# Plot R and Stilde
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, R, 'r-',linewidth=2)
ax[0].set_title('$R(t)$',fontsize=fontSize)

ax[1].plot(t, Stilde, 'k--',linewidth=2)
ax[1].set_title('$\widetilde{S}(t)$',fontsize=fontSize)

plt.show()

# Calculate the noise-corrected PDF
sg = Stilde
sg[1:] -= sg[:-1].copy()
sg += epsilon

# Plot R and Stilde
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, r, 'r-',linewidth=2)
ax[0].set_title('$r(t)$',fontsize=fontSize)

ax[1].plot(t, sg, 'k--',linewidth=2)
ax[1].set_title('$\widetilde{s}_g(t)$',fontsize=fontSize)

plt.show()
```



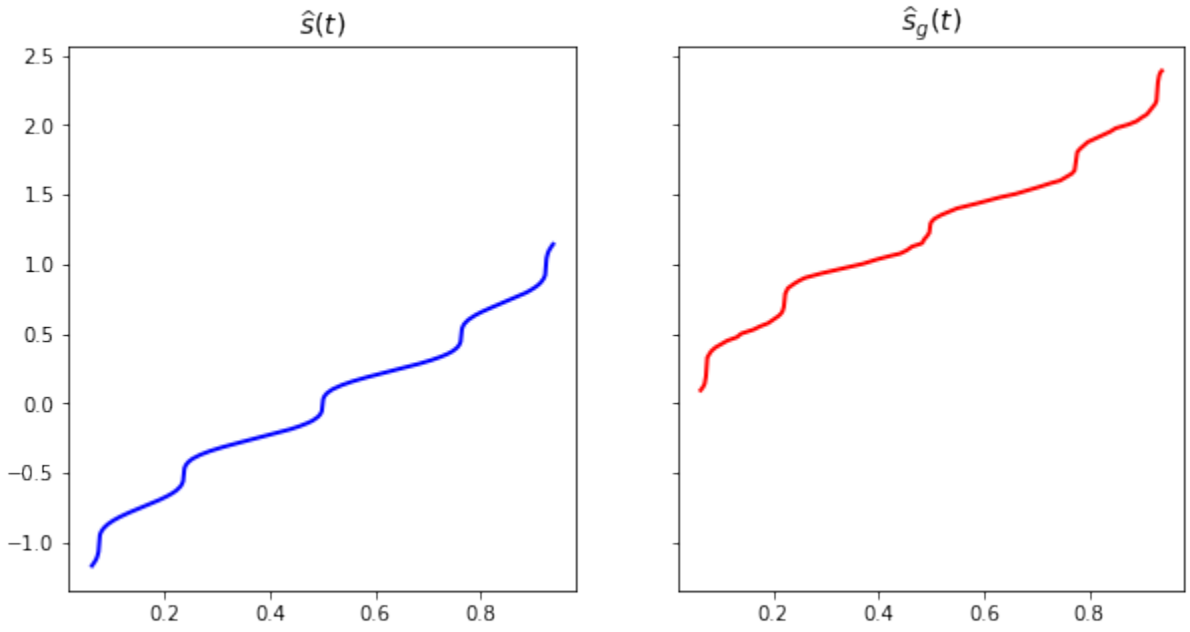
Calculate CDTs using PyTransKit package

[23]:
```
# Reference signal
t0 = np.linspace(0, 1, N)
z0= np.ones(t0.size)+epsilon
s0 = z0**2/np.linalg.norm(z0)**2

# Create a CDT object
cdt = CDT()
```

(continues on next page)

(continued from previous page)

```
# Compute the forward transform
s_hat, s_hat_old, xtilde = cdt.forward(t0, s0, t, s, rm_edge=False)
sg_hat, sg_hat_old, xtilde = cdt.forward(t0, s0, t, sg, rm_edge=False)

# remove the edges
s_hat = s_hat[25:N-25]
sg_hat = sg_hat[25:N-25]
xtilde = xtilde[25:N-25]

# Plot s_hat and sg_hat
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(xtilde, s_hat, 'b-',linewidth=2)
ax[0].set_title('$\widehat{s}(t)$',fontsize=fontSize)

ax[1].plot(xtilde, sg_hat, 'r-',linewidth=2)
ax[1].set_title('$\widehat{s}_g(t)$',fontsize=fontSize)

plt.show()
```



Calculate time delay estimate

[24]:
```
estimate = np.mean(sg_hat) - np.mean(s_hat)

print('\nTrue value of time delay: '+str(tau) + ' seconds')
print('Estimated value of time delay: '+str(estimate) + ' seconds\n')
```

```
True value of time delay: 1.2575 seconds
Estimated value of time delay: 1.2562758076649236 seconds
```

[ ]:
```
```

3.2 Estimation of time delay and linear dispersion

The signal of interest is

$$z(t) = Ae^{-\frac{(t-t_c)^2}{2b_w^2}} \sin(2\pi f t) \tag{3.6}$$

of width b_w and frequency f. In this example we will take $A = b_w = f = 1$ and set $t_c = 0$. The probability density function (PDF),

$$s(t) = B(z)(t) := \frac{z^2(t)}{\int_{\Omega_s} z^2(t)dt} \tag{3.7}$$

In this example, we are interested in estimating the time delay (τ) and linear dispersion (ω) parameters, i.e. $g_p(t) = \omega t - \tau$. Therefore $z_g(t) = z(\omega t - \tau)$ and $s_g(t) = B(z_g)(t)$.

Normalized measured signal with noise $\eta \sim \mathcal{N}(0, \sigma^2)$ is given by,

$$r(t) = B(z_g + \eta)(t) \tag{3.8}$$

3.2.1 Parameter estimation in the CDT domain

Let, $\vec{p} = [\tau, \omega]^T$, and $\hat{s}$ and $\hat{r}$ be the CDTs of $s(t)$ and $r(t)$, respectively. The estimates are calculated as,

$$\widetilde{\vec{p}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\hat{s} \tag{3.9}$$

where $\mathbf{X} \equiv \left[-\vec{\mathbf{1}}, \vec{\hat{r}}\right]$.

Create $s(t)$ and $r(t)$

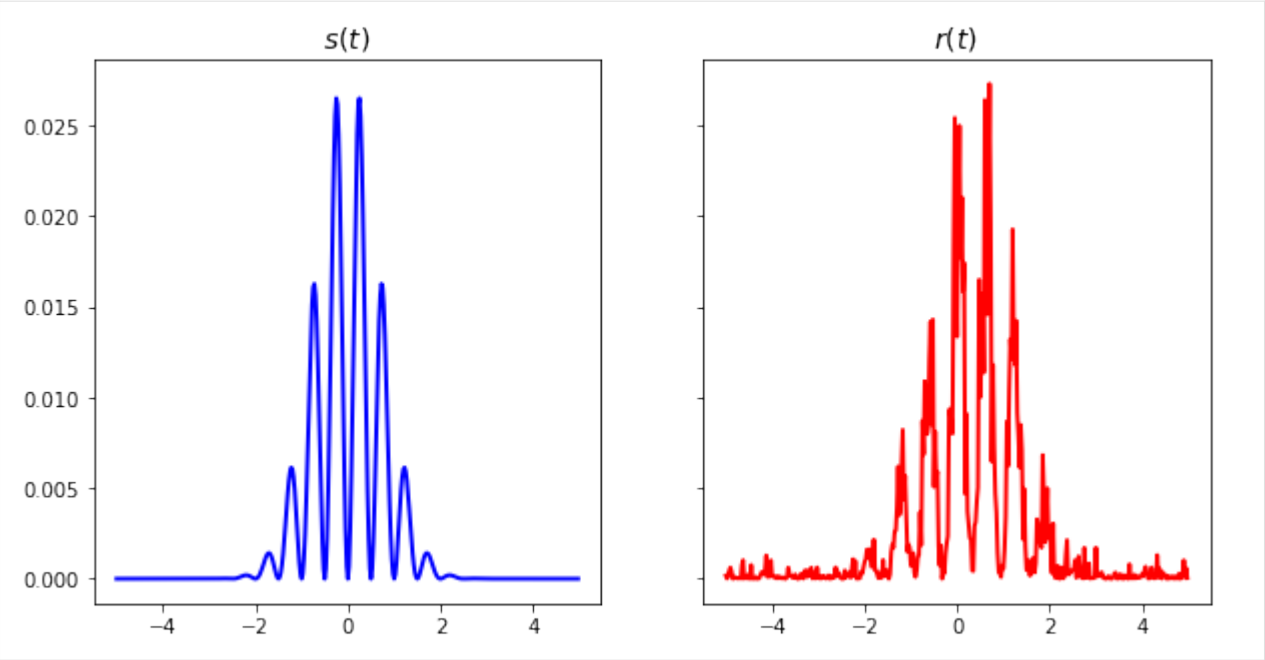
[13]:
```
import numpy as np
import matplotlib.pyplot as plt

# To use the CDT first install the PyTransKit package using:
# pip install pytranskit
```

(continues on next page)

(continued from previous page)

```
from pytranskit.optrans.continuous.cdt import CDT

N = 400
dt = 0.025
t = np.linspace(-N/2*dt, (N/2-1)*dt, N)
f = 1
epsilon = 1e-8

# Original signal
gwin = np.exp(-t**2/2)
z = gwin*np.sin(2*np.pi*f*t) + epsilon
s = z**2/np.linalg.norm(z)**2

# zero mean additive Gaussian noise
sigma = 0.1          # standard deviation
SNRdb = 10*np.log10(np.mean(z**2)/sigma**2)
print('SNR: {} dB'.format(SNRdb))
noise = np.random.normal(0,sigma,N)

# Signal after delay and dispersion
omega = 0.8
tau = 10.3*dt
gwin = np.exp(-(omega*t - tau)**2/2)
zg = gwin*np.sin(2*np.pi*f*(omega*t - tau)) + noise + epsilon
r = zg**2/np.linalg.norm(zg)**2

# Plot s and sg
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, s, 'b-',linewidth=2)
ax[0].set_title('$s(t)$',fontsize=fontSize)

ax[1].plot(t, r, 'r-',linewidth=2)
ax[1].set_title('$r(t)$',fontsize=fontSize)

plt.show()
```

```
SNR: 9.47544940682685 dB
```



Noise correction

Expression of noise-corrected CDF is given by,

$$\tilde{S}_g(t) = \frac{E[R(t)]\{\mathcal{E}_z + \sigma^2(t_N - t_1)\} - \sigma^2(t - t_1)}{\mathcal{E}_z}, \quad t_1 \leq t \leq t_N \tag{3.10}$$

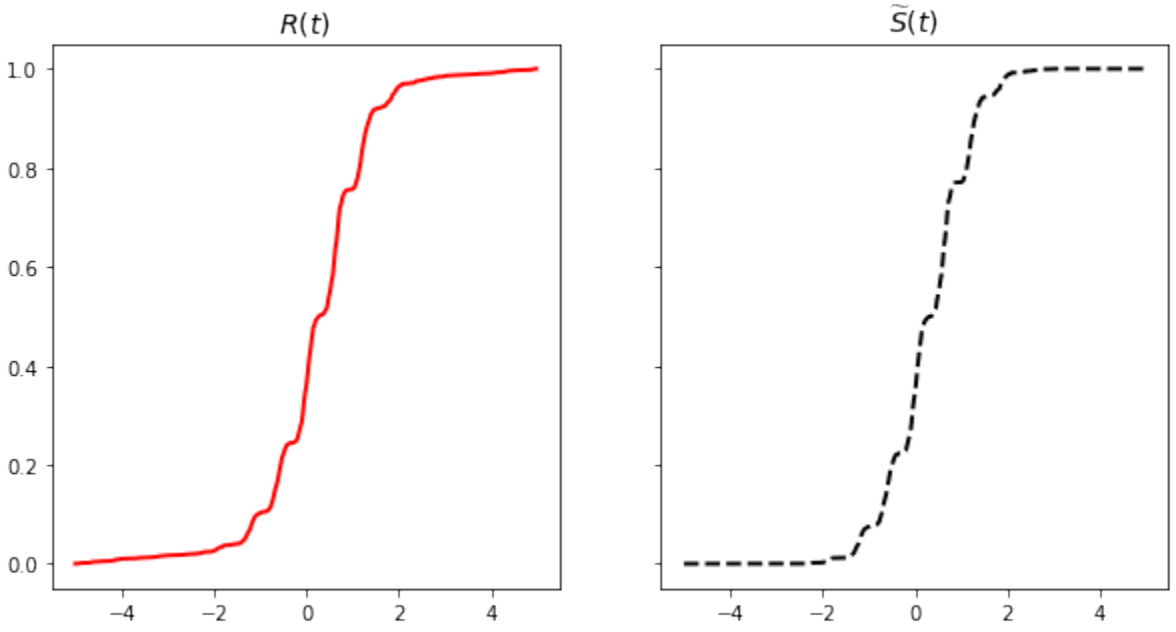
where $R(t)$ is the CDF associated with $r(t)$, σ is the standard deviation of noise, and $\mathcal{E}_z$ is the total energy of the noise free signal.

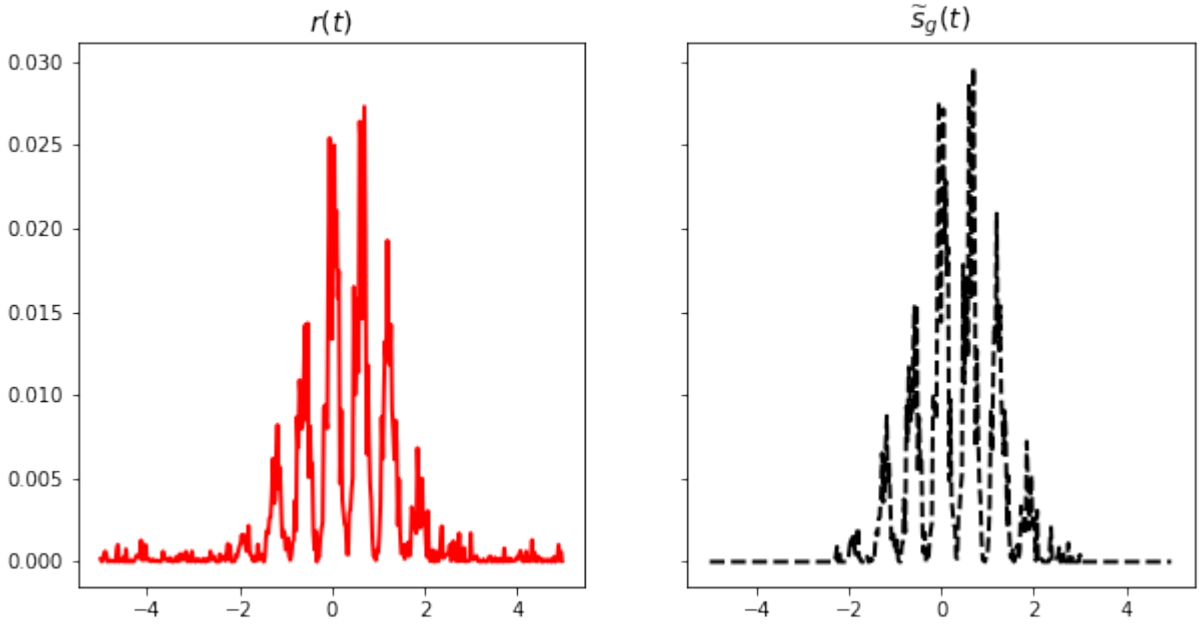
[14]:
```
# Calculate the noise-corrected CDF
R = np.cumsum(r)                    # CDF of r(t)
Ez = np.mean(z**2)*(t[-1] - t[0])   # Energy of the signal
Stilde = (R*(Ez+sigma**2*(t[-1]-t[0])) - sigma**2*(t-t[0]))/Ez

# Preserve the non-decreasing property of CDFs
mind = np.argmin(Stilde)
Mind = np.argmax(Stilde)
Stilde[0:mind] = Stilde[mind]
Stilde[Mind:] = Stilde[Mind]
for i in range(len(Stilde)-1):
    if Stilde[i+1]<=Stilde[i]:
        Stilde[i+1] = Stilde[i] + epsilon

Stilde = (Stilde - np.min(Stilde))/(np.max(Stilde) - np.min(Stilde))

# Plot R and Stilde
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, R, 'r-',linewidth=2)
ax[0].set_title('$R(t)$',fontsize=fontSize)
```

(continues on next page)

(continued from previous page)

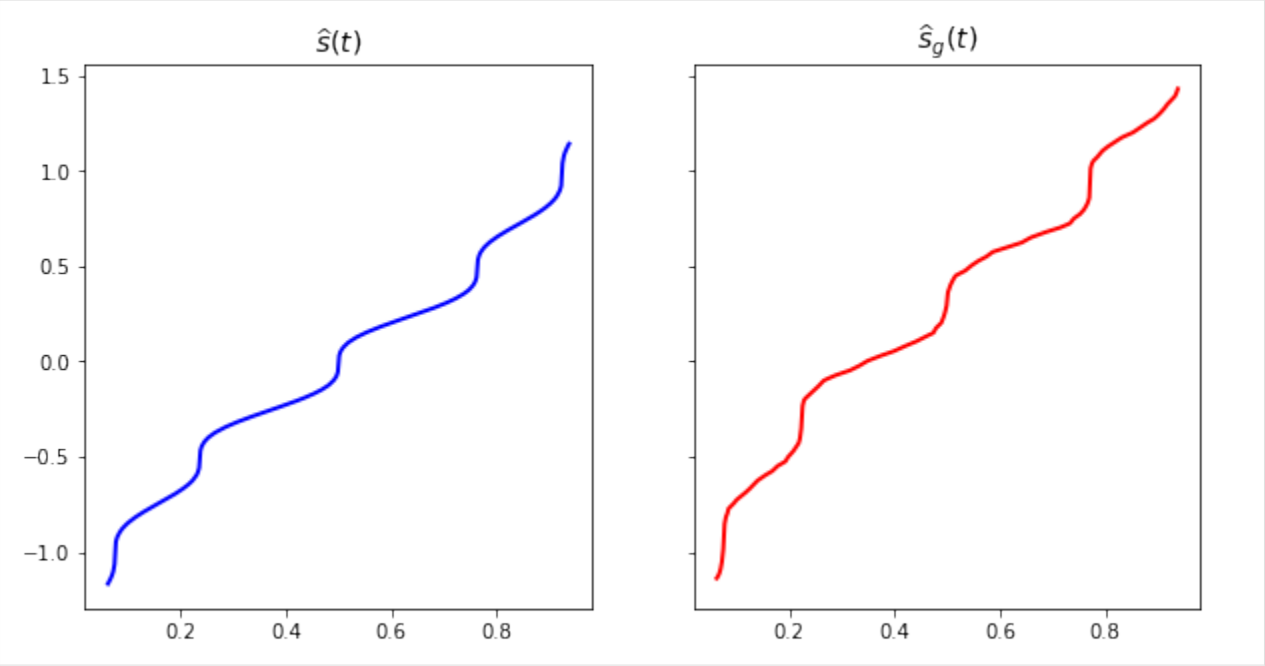
```
ax[1].plot(t, Stilde, 'k--',linewidth=2)
ax[1].set_title('$\widetilde{S}(t)$',fontsize=fontSize)

plt.show()

# Calculate the noise-corrected PDF
sg = Stilde
sg[1:] -= sg[:-1].copy()
sg += epsilon

# Plot R and Stilde
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(t, r, 'r-',linewidth=2)
ax[0].set_title('$r(t)$',fontsize=fontSize)

ax[1].plot(t, sg, 'k--',linewidth=2)
ax[1].set_title('$\widetilde{s}_g(t)$',fontsize=fontSize)

plt.show()
```





Calculate CDTs using PyTransKit package

[15]:
```
# Reference signal
t0 = np.linspace(0, 1, N)
z0= np.ones(t0.size)+epsilon
s0 = z0**2/np.linalg.norm(z0)**2

# Create a CDT object
cdt = CDT()

# Compute the forward transform
s_hat, s_hat_old, xtilde = cdt.forward(t0, s0, t, s, rm_edge=False)
sg_hat, sg_hat_old, xtilde = cdt.forward(t0, s0, t, sg, rm_edge=False)

# remove the edges
s_hat = s_hat[25:N-25]
sg_hat = sg_hat[25:N-25]
xtilde = xtilde[25:N-25]

# Plot s_hat and sg_hat
fontSize=14
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(10,5))
ax[0].plot(xtilde, s_hat, 'b-',linewidth=2)
ax[0].set_title('$\widehat{s}(t)$',fontsize=fontSize)

ax[1].plot(xtilde, sg_hat, 'r-',linewidth=2)
ax[1].set_title('$\widehat{s}_g(t)$',fontsize=fontSize)

plt.show()
```



Calculate time delay and linear dispersion estimates

[16]:
```
X = -1*np.ones([len(s_hat),2])
X[:,0] = np.transpose(sg_hat)
estimates = np.linalg.solve(np.matmul(np.transpose(X),X), np.matmul(np.transpose(X),np.
↪transpose(s_hat)))

print('\nTrue value of time delay: '+str(tau) + ' seconds')
print('Estimated value of time delay: '+str(estimates[1]) + ' seconds\n')

print('True value of linear dispersion: '+str(omega))
print('Estimated value of linear dispersion: '+str(estimates[0]) +'\n')
```

```
True value of time delay: 0.2575 seconds
Estimated value of time delay: 0.2694287224582905 seconds

True value of linear dispersion: 0.8
Estimated value of linear dispersion: 0.8408065181070804
```

[ ]:

CHAPTER

FOUR

RESOURCES

- External website http://imagedatascience.com/transport/

CHAPTER

FIVE

API AND MODULES

5.1 pytranskit.optrans.continuous package

5.1.1 base transform

class pytranskit.optrans.continuous.base.**BaseMapper2D**

Bases: *BaseTransform*

Base class for 2D optimal transport transform methods (e.g. CLOT, VOT2D).

> **Warning:** This class should **not** be used directly. Use derived classes instead.

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*array, shape (2, height, width)*) – Forward transport map.
- **sig1** (*array, shape (height, width)*) – Signal to transform.

Returns

sig0_recon – Reconstructed reference signal sig0.

Return type

array, shape (height, width)

apply_inverse_map(*transport_map*, *sig0*)

Appy inverse transport map.

Parameters

- **transport_map** (*array, shape (2, height, width)*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*array, shape (height, width)*) – Reference signal.

Returns

sig1_recon – Reconstructed signal sig1.

Return type

array, shape (height, width)

class pytranskit.optrans.continuous.base.**BaseTransform**

Bases: object

Base class for optimal transport transform methods.

> **Warning:** This class should **not** be used directly. Use derived classes instead.

apply_forward_map()

Placeholder for application of forward transport map. Subclasses should implement this method!

apply_inverse_map()

Placeholder for application of inverse transport map. Subclasses should implement this method!

forward()

Placeholder for forward transform. Subclasses should implement this method!

inverse()

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

pytranskit.optrans.continuous.base.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.continuous.base.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns
array_converted – The converted and validated array.

Return type
object

pytranskit.optrans.continuous.base.**griddata2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolate 2d scattered data

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate.
- **f** (*3d array, shape (2, height, width)*) – Coordinates of at which img is defined. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the interpolation. Must be in the range 0-2.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns
Image interpolated on to regular gird defined by: x = 0:(width-1), y = 0:(height-1).

Return type
out, 2d array, shape (height, width)

pytranskit.optrans.continuous.base.**interp2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolation for 2D gridded data.

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate. This function assumes a grid of sample points: x = 0:(width-1), y = 0:(height-1).
- **f** (*3d array, shape (2, height, width)*) – Coordinates of interpolated points. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the spline interpolation. Must be in the range 0-5.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns
Image interpolated at points defined by f.

Return type
out, 2d array, shape (height, width)

5.1.2 cdt

class pytranskit.optrans.continuous.cdt.**BaseTransform**

Bases: object

Base class for optimal transport transform methods.

> **Warning:** This class should **not** be used directly. Use derived classes instead.

apply_forward_map()

Placeholder for application of forward transport map. Subclasses should implement this method!

apply_inverse_map()

Placeholder for application of inverse transport map. Subclasses should implement this method!

forward()

Placeholder for forward transform. Subclasses should implement this method!

inverse()

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

class pytranskit.optrans.continuous.cdt.**CDT**

Bases: *BaseTransform*

Cumulative Distribution Transform.

displacements_

Displacements u.

Type
1d array

transport_map_

Transport map f.

Type
1d array

References

[The cumulative distribution transform and linear pattern classification] (https://arxiv.org/abs/1507.05936)

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig1** (*1d array*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
1d array

apply_inverse_map(*transport_map*, *sig0*, *x*)

Apply inverse transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*1d array*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
1d array

forward(*x0*, *sig0*, *x1*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **x0** (*1d array*) – Independent axis variable of reference signal (sig0).
- **sig0** (*1d array*) – Reference signal.

- **x1** (*1d array*) – Independent axis variable of the signal to transform (sig1).
- **sig1** (*1d array*) – Signal to transform.

Returns

- **sig1_cdt** (*1d array*) – CDT of input signal sig1 (new definition).
- **sig1_hat** (*1d array*) – old definition.
- **xilde** (*1d array*) – Independent axis variable in CDT space.

inverse(*transport_map*, *sig0*, *x1*)

Inverse transform.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to reconstruct.

Returns
sig1_recon – Reconstructed signal.

Return type
1d array

pytranskit.optrans.continuous.cdt.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.continuous.cdt.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns
array_converted – The converted and validated array.

Return type
object

pytranskit.optrans.continuous.cdt.**interp**(*x*, *xp*, *fp*, *left=None*, *right=None*, *period=None*)

One-dimensional linear interpolation for monotonically increasing sample points.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (*xp*, *fp*), evaluated at *x*.

Parameters

- **x** (*array_like*) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (*1-D sequence of floats*) – The x-coordinates of the data points, must be increasing if argument *period* is not specified. Otherwise, *xp* is internally sorted after normalizing the periodic boundaries with `xp = xp % period`.
- **fp** (*1-D sequence of float or complex*) – The y-coordinates of the data points, same length as *xp*.
- **left** (*optional float or complex corresponding to fp*) – Value to return for *x* < *xp[0]*, default is *fp[0]*.
- **right** (*optional float or complex corresponding to fp*) – Value to return for *x* > *xp[-1]*, default is *fp[-1]*.
- **period** (*None or float, optional*) – A period for the x-coordinates. This parameter allows the proper interpolation of angular x-coordinates. Parameters *left* and *right* are ignored if *period* is specified.

  New in version 1.10.0.

Returns

y – The interpolated values, same shape as *x*.

Return type

float or complex (corresponding to fp) or ndarray

Raises

ValueError – If *xp* and *fp* have different length If *xp* or *fp* are not 1-D sequences If *period* == *0*

See also:

`scipy.interpolate`

Warning: The x-coordinate sequence is expected to be increasing, but this is not explicitly enforced. However, if the sequence *xp* is non-increasing, interpolation results are meaningless.

Note that, since NaN is unsortable, *xp* also cannot contain NaNs.

A simple check for *xp* being strictly increasing is:

```
np.all(np.diff(xp) > 0)
```

Examples

```
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([3.  , 3.  , 2.5 , 0.56, 0.  ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(xvals, yinterp, '-x')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.show()
```

Interpolation with periodic x-coordinates:

```
>>> x = [-180, -170, -185, 185, -10, -5, 0, 365]
>>> xp = [190, -190, 350, -350]
>>> fp = [5, 10, 3, 4]
>>> np.interp(x, xp, fp, period=360)
array([7.5 , 5.  , 8.75, 6.25, 3.  , 3.25, 3.5 , 3.75])
```

Complex interpolation:

```
>>> x = [1.5, 4.0]
>>> xp = [2,3,5]
>>> fp = [1.0j, 0, 2+3j]
>>> np.interp(x, xp, fp)
array([0.+1.j , 1.+1.5j])
```

pytranskit.optrans.continuous.cdt.**signal_to_pdf**(*input*, *sigma=0.0*, *epsilon=1e-08*, *total=1.0*)

Get the (smoothed) probability density function of a signal.

Performs the following operations: 1. Smooth sigma with a Gaussian filter 2. Normalize signal such that it sums to 1 3. Add epsilon to ensure signal is strictly positive 4. Re-normalize signal such that it sums to total

Parameters

- **input** (*ndarray*) – Input array
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **epsilon** (*scalar*) – Offset to ensure that signal is strictly positive.
- **total** (*scalar*) – Value of the signal summation.

Returns

pdf – Returned array of same shape as input

Return type

ndarray

5.1.3 clot

class pytranskit.optrans.continuous.clot.**CDT**

Bases: *BaseTransform*

Cumulative Distribution Transform.

displacements_

Displacements u.

Type
1d array

transport_map_

Transport map f.

Type
1d array

References

[The cumulative distribution transform and linear pattern classification] (https://arxiv.org/abs/1507.05936)

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig1** (*1d array*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
1d array

apply_inverse_map(*transport_map*, *sig0*, *x*)

Apply inverse transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*1d array*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
1d array

forward(*x0*, *sig0*, *x1*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **x0** (*1d array*) – Independent axis variable of reference signal (sig0).
- **sig0** (*1d array*) – Reference signal.

- **x1** (*1d array*) – Independent axis variable of the signal to transform (sig1).
- **sig1** (*1d array*) – Signal to transform.

Returns

- **sig1_cdt** (*1d array*) – CDT of input signal sig1 (new definition).
- **sig1_hat** (*1d array*) – old definition.
- **xilde** (*1d array*) – Independent axis variable in CDT space.

inverse(*transport_map*, *sig0*, *x1*)

Inverse transform.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to reconstruct.

Returns
sig1_recon – Reconstructed signal.

Return type
1d array

class pytranskit.optrans.continuous.clot.**CLOT**(*lr=0.01*, *momentum=0.0*, *decay=0.0*, *max_iter=300*, *tol=0.001*, *verbose=0*)

Bases: object

Continuous Linear Optimal Transport Transform.

This uses Nesterov's accelerated gradient descent to remove the curl in the initial mapping.

Parameters

- **lr** (*float (default=0.01)*) – Learning rate.
- **momentum** (*float (default=0.)*) – Nesterov accelerated gradient descent momentum.
- **decay** (*float (default=0.)*) – Learning rate decay over each update.
- **max_iter** (*int (default=300)*) – Maximum number of iterations.
- **tol** (*float (default=0.001)*) – Stop iterating when change in cost function is below this threshold.
- **verbose** (*int (default=1)*) – Verbosity during optimization. 0=no output, 1=print cost, 2=print all metrics.

displacements_

Displacements u. First index denotes direction: displacements_[0] is y-displacements, and displacements_[1] is x-displacements.

Type
array, shape (2, height, width)

transport_map_

Transport map f. First index denotes direction: transport_map_[0] is y-map, and transport_map_[1] is x-map.

Type
array, shape (2, height, width)

displacements_initial_

Initial displacements computed using the method by Haker et al.

Type
array, shape (2, height, width)

transport_map_initial_

Initial transport map computed using the method by Haker et al.

Type
array, shape (2, height, width)

cost_

Value of cost function at each iteration.

Type
list of float

curl_

Curl at each iteration.

Type
list of float

References

[A continuous linear optimal transport approach for pattern analysis in image datasets] (https://www.sciencedirect.com/science/article/pii/S0031320315003507) [Optimal mass transport for registration and warping] (https://link.springer.com/article/10.1023/B:VISI.0000036836.66311.97)

apply_forward_map(*transport_map*, *sig1*)

Apply forward transport map.

Parameters

- **transport_map** (*array, shape (2, height, width)*) – Forward transport map.
- **sig1** (*array, shape (height, width)*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
array, shape (height, width)

apply_inverse_map(*transport_map*, *sig0*)

Apply inverse transport map.

Parameters

- **transport_map** (*array, shape (2, height, width)*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*array, shape (height, width)*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

forward(*sig0*, *sig1*)

Forward transform.

Parameters

- **sig0** (*array, shape (height, width)*) – Reference image.
- **sig1** (*array, shape (height, width)*) – Signal to transform.

Returns

lot – LOT transform of input image sig1. First index denotes direction: lot[0] is y-LOT, and lot[1] is x-LOT.

Return type

array, shape (2, height, width)

pytranskit.optrans.continuous.clot.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.continuous.clot.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns

array_converted – The converted and validated array.

Return type

object

pytranskit.optrans.continuous.clot.**dct**(*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

Return the Discrete Cosine Transform of arbitrary type sequence x.

Parameters

- **x** (*array_like*) – The input array.
- **type** (*{1, 2, 3, 4}, optional*) – Type of the DCT (see Notes). Default type is 2.
- **n** (*int, optional*) – Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the dct is computed; the default is over the last axis (i.e., `axis=-1`).
- **norm** (*{None, 'ortho'}, optional*) – Normalization mode (see Notes). Default is None.

- **overwrite_x** (*bool, optional*) – If True, the contents of x can be destroyed; the default is False.

Returns
y – The transformed input array.

Return type
ndarray of real

See also:

idct
Inverse DCT

Notes

For a single dimension array `x`, `dct(x, norm='ortho')` is equal to MATLAB `dct(x)`.

There are, theoretically, 8 types of the DCT, only the first 4 types are implemented in scipy. 'The' DCT generally refers to DCT type 2, and 'the' Inverse DCT generally refers to DCT type 3.

Type I

There are several definitions of the DCT-I; we use the following (for `norm=None`)

$$y_k = x_0 + (-1)^k x_{N-1} + 2\sum_{n=1}^{N-2} x_n \cos\left(\frac{\pi k n}{N-1}\right)$$

If `norm='ortho'`, `x[0]` and `x[N-1]` are multiplied by a scaling factor of $\sqrt{2}$, and `y[k]` is multiplied by a scaling factor `f`

$$f = \begin{cases} \frac{1}{2}\sqrt{\frac{1}{N-1}} & \text{if } k = 0 \text{ or } N-1, \\ \frac{1}{2}\sqrt{\frac{2}{N-1}} & \text{otherwise} \end{cases}$$

New in version 1.2.0: Orthonormalization in DCT-I.

Note: The DCT-I is only supported for input size > 1.

Type II

There are several definitions of the DCT-II; we use the following (for `norm=None`)

$$y_k = 2\sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi k(2n+1)}{2N}\right)$$

If `norm='ortho'`, `y[k]` is multiplied by a scaling factor `f`

$$f = \begin{cases} \sqrt{\frac{1}{4N}} & \text{if } k = 0, \\ \sqrt{\frac{1}{2N}} & \text{otherwise} \end{cases}$$

which makes the corresponding matrix of coefficients orthonormal (`O @ O.T = np.eye(N)`).

Type III

There are several definitions, we use the following (for `norm=None`)

$$y_k = x_0 + 2\sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi(2k+1)n}{2N}\right)$$

or, for `norm='ortho'`

$$y_k = \frac{x_0}{\sqrt{N}} + \sqrt{\frac{2}{N}}\sum_{n=1}^{N-1} x_n \cos\left(\frac{\pi(2k+1)n}{2N}\right)$$

The (unnormalized) DCT-III is the inverse of the (unnormalized) DCT-II, up to a factor *2N*. The orthonormalized DCT-III is exactly the inverse of the orthonormalized DCT-II.

Type IV

There are several definitions of the DCT-IV; we use the following (for `norm=None`)

$$y_k = 2\sum_{n=0}^{N-1} x_n \cos\left(\frac{\pi(2k+1)(2n+1)}{4N}\right)$$

If `norm='ortho'`, `y[k]` is multiplied by a scaling factor `f`

$$f = \frac{1}{\sqrt{2N}}$$

New in version 1.2.0: Support for DCT-IV.

References

Examples

The Type 1 DCT is equivalent to the FFT (though faster) for real, even-symmetrical inputs. The output is also real and even-symmetrical. Half of the FFT input is used to generate half of the FFT output:

```
>>> from scipy.fftpack import fft, dct
>>> fft(np.array([4., 3., 5., 10., 5., 3.])).real
array([ 30.,  -8.,   6.,  -2.,   6.,  -8.])
>>> dct(np.array([4., 3., 5., 10.]), 1)
array([ 30.,  -8.,   6.,  -2.])
```

pytranskit.optrans.continuous.clot.**griddata2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolate 2d scattered data

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate.
- **f** (*3d array, shape (2, height, width)*) – Coordinates of at which img is defined. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the interpolation. Must be in the range 0-2.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns

Image interpolated on to regular gird defined by: x = 0:(width-1), y = 0:(height-1).

Return type

out, 2d array, shape (height, width)

pytranskit.optrans.continuous.clot.**idct**(*x*, *type=2*, *n=None*, *axis=-1*, *norm=None*, *overwrite_x=False*)

Return the Inverse Discrete Cosine Transform of an arbitrary type sequence.

Parameters

- **x** (*array_like*) – The input array.
- **type** (*{1, 2, 3, 4}, optional*) – Type of the DCT (see Notes). Default type is 2.
- **n** (*int, optional*) – Length of the transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the idct is computed; the default is over the last axis (i.e., `axis=-1`).
- **norm** (*{None, 'ortho'}, optional*) – Normalization mode (see Notes). Default is None.
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed; the default is False.

Returns

idct – The transformed input array.

Return type

ndarray of real

See also:

dct

Forward DCT

Notes

For a single dimension array *x*, `idct(x, norm='ortho')` is equal to MATLAB `idct(x)`.

'The' IDCT is the IDCT of type 2, which is the same as DCT of type 3.

IDCT of type 1 is the DCT of type 1, IDCT of type 2 is the DCT of type 3, and IDCT of type 3 is the DCT of type 2. IDCT of type 4 is the DCT of type 4. For the definition of these types, see *dct*.

Examples

The Type 1 DCT is equivalent to the DFT for real, even-symmetrical inputs. The output is also real and even-symmetrical. Half of the IFFT input is used to generate half of the IFFT output:

```
>>> from scipy.fftpack import ifft, idct
>>> ifft(np.array([ 30.,  -8.,   6.,  -2.,   6.,  -8.])).real
array([  4.,   3.,   5.,  10.,   5.,   3.])
>>> idct(np.array([ 30.,  -8.,   6.,  -2.]), 1) / 6
array([  4.,   3.,   5.,  10.])
```

pytranskit.optrans.continuous.clot.**interp2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolation for 2D gridded data.

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate. This function assumes a grid of sample points: x = 0:(width-1), y = 0:(height-1).

- **f** (*3d array, shape (2, height, width)*) – Coordinates of interpolated points. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the spline interpolation. Must be in the range 0-5.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns
Image interpolated at points defined by f.

Return type
out, 2d array, shape (height, width)

pytranskit.optrans.continuous.clot.**signal_to_pdf**(*input*, *sigma=0.0*, *epsilon=1e-08*, *total=1.0*)

Get the (smoothed) probability density function of a signal.

Performs the following operations: 1. Smooth sigma with a Gaussian filter 2. Normalize signal such that it sums to 1 3. Add epsilon to ensure signal is strictly positive 4. Re-normalize signal such that it sums to total

Parameters

- **input** (*ndarray*) – Input array
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **epsilon** (*scalar*) – Offset to ensure that signal is strictly positive.
- **total** (*scalar*) – Value of the signal summation.

Returns
pdf – Returned array of same shape as input

Return type
ndarray

5.1.4 radoncdt

class pytranskit.optrans.continuous.radoncdt.**BaseTransform**

Bases: object

Base class for optimal transport transform methods.

> **Warning:** This class should **not** be used directly. Use derived classes instead.

apply_forward_map()

Placeholder for application of forward transport map. Subclasses should implement this method!

apply_inverse_map()

Placeholder for application of inverse transport map. Subclasses should implement this method!

forward()

Placeholder for forward transform. Subclasses should implement this method!

inverse()

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

class pytranskit.optrans.continuous.radoncdt.CDT

Bases: *BaseTransform*

Cumulative Distribution Transform.

displacements_

Displacements u.

Type
1d array

transport_map_

Transport map f.

Type
1d array

References

[The cumulative distribution transform and linear pattern classification] (https://arxiv.org/abs/1507.05936)

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig1** (*1d array*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
1d array

apply_inverse_map(*transport_map*, *sig0*, *x*)

Apply inverse transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*1d array*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
1d array

forward(*x0*, *sig0*, *x1*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **x0** (*1d array*) – Independent axis variable of reference signal (sig0).
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to transform (sig1).
- **sig1** (*1d array*) – Signal to transform.

Returns

- **sig1_cdt** (*1d array*) – CDT of input signal sig1 (new definition).
- **sig1_hat** (*1d array*) – old definition.
- **xilde** (*1d array*) – Independent axis variable in CDT space.

inverse(*transport_map*, *sig0*, *x1*)

Inverse transform.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to reconstruct.

Returns
sig1_recon – Reconstructed signal.

Return type
1d array

class pytranskit.optrans.continuous.radoncdt.**RadonCDT**(*theta=array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179])*)

Bases: *BaseTransform*

Radon Cumulative Distribution Transform.

Parameters
theta (*1d array (default=np.arange(180))*) – Radon transform projection angles.

displacements_

Displacements u.

Type
array, shape (t, len(theta))

transport_map_

Transport map f.

Type
array, shape (t, len(theta))

References

[The Radon cumulative distribution transform and its application to image classification] (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4871726/)

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*array, shape (t, len(theta))*) – Forward transport map.
- **sig1** (*2d array, shape (height, width)*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
array, shape (height, width)

apply_inverse_map(*transport_map*, *sig0*, *x_range*)

Appy inverse transport map.

Parameters

- **transport_map** (*2d array, shape (t, len(theta))*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*array, shape (height, width)*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

forward(*x0_range*, *sig0*, *x1_range*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **sig0** (*array, shape (height, width)*) – Reference image.
- **sig1** (*array, shape (height, width)*) – Signal to transform.

Returns
rcdt – Radon-CDT of input image sig1.

Return type
array, shape (t, len(theta))

inverse(*transport_map*, *sig0*, *x1_range*)

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

pytranskit.optrans.continuous.radoncdt.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.continuous.radoncdt.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns
array_converted – The converted and validated array.

Return type
object

pytranskit.optrans.continuous.radoncdt.**iradon**(*radon_image*, *theta=None*, *output_size=None*, *filter_name='ramp'*, *interpolation='linear'*, *circle=True*, *preserve_range=True*)

Inverse radon transform.

Reconstruct an image from the radon transform, using the filtered back projection algorithm.

Parameters

- **radon_image** (*array*) – Image containing radon transform (sinogram). Each column of the image corresponds to a projection along a different angle. The tomography rotation axis should lie at the pixel index `radon_image.shape[0] // 2` along the 0th dimension of `radon_image`.
- **theta** (*array_like, optional*) – Reconstruction angles (in degrees). Default: m angles evenly spaced between 0 and 180 (if the shape of *radon_image* is (N, M)).
- **output_size** (*int, optional*) – Number of rows and columns in the reconstruction.
- **filter_name** (*str, optional*) – Filter used in frequency domain filtering. Ramp filter used by default. Filters available: ramp, shepp-logan, cosine, hamming, hann. Assign None to use no filter.
- **interpolation** (*str, optional*) – Interpolation method used in reconstruction. Methods available: 'linear', 'nearest', and 'cubic' ('cubic' is slow).

- **circle** (*boolean, optional*) – Assume the reconstructed image is zero outside the inscribed circle. Also changes the default output_size to match the behaviour of `radon` called with `circle=True`.
- **preserve_range** (*bool, optional*) – Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of *img_as_float*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

Returns

- **reconstructed** (*ndarray*) – Reconstructed image. The rotation axis will be located in the pixel with indices `(reconstructed.shape[0] // 2, reconstructed.shape[1] // 2)`.
- .. *versionchanged:: 0.19* – In `iradon`, `filter` argument is deprecated in favor of `filter_name`.

References

Notes

It applies the Fourier slice theorem to reconstruct an image by multiplying the frequency domain of the filter with the FFT of the projection data. This algorithm is called filtered back projection.

pytranskit.optrans.continuous.radoncdt.**match_shape2d**(*a*, *b*)

Crop array B such that it matches the shape of A.

Parameters

- **a** (*2d array*) – Array of desired size.
- **b** (*2d array*) – Array to crop. Shape must be larger than (or equal to) the shape of array a.

Returns
b_crop – Cropped version of b, with the same shape as a.

Return type
2d array

pytranskit.optrans.continuous.radoncdt.**radon**(*image*, *theta=None*, *circle=True*, ***, *preserve_range=False*)

Calculates the radon transform of an image given specified projection angles.

Parameters

- **image** (*array_like*) – Input image. The rotation axis will be located in the pixel with indices `(image.shape[0] // 2, image.shape[1] // 2)`.
- **theta** (*array_like, optional*) – Projection angles (in degrees). If *None*, the value is set to np.arange(180).
- **circle** (*boolean, optional*) – Assume image is zero outside the inscribed circle, making the width of each projection (the first dimension of the sinogram) equal to `min(image.shape)`.
- **preserve_range** (*bool, optional*) – Whether to keep the original range of values. Otherwise, the input image is converted according to the conventions of *img_as_float*. Also see https://scikit-image.org/docs/dev/user_guide/data_types.html

Returns
radon_image – Radon transform (sinogram). The tomography rotation axis will lie at the pixel index `radon_image.shape[0] // 2` along the 0th dimension of `radon_image`.

Return type
ndarray

References

Notes

Based on code of Justin K. Romberg (https://www.clear.rice.edu/elec431/projects96/DSP/bpanalysis.html)

`pytranskit.optrans.continuous.radoncdt.`**signal_to_pdf**(*input*, *sigma=0.0*, *epsilon=1e-08*, *total=1.0*)

Get the (smoothed) probability density function of a signal.

Performs the following operations: 1. Smooth sigma with a Gaussian filter 2. Normalize signal such that it sums to 1 3. Add epsilon to ensure signal is strictly positive 4. Re-normalize signal such that it sums to total

Parameters

- **input** (*ndarray*) – Input array
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **epsilon** (*scalar*) – Offset to ensure that signal is strictly positive.
- **total** (*scalar*) – Value of the signal summation.

Returns
pdf – Returned array of same shape as input

Return type
ndarray

5.1.5 radoncdt3D

class `pytranskit.optrans.continuous.radoncdt3D.`**BaseTransform**

Bases: `object`

Base class for optimal transport transform methods.

> **Warning:** This class should **not** be used directly. Use derived classes instead.

apply_forward_map()

Placeholder for application of forward transport map. Subclasses should implement this method!

apply_inverse_map()

Placeholder for application of inverse transport map. Subclasses should implement this method!

forward()

Placeholder for forward transform. Subclasses should implement this method!

inverse()

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

class pytranskit.optrans.continuous.radoncdt3D.**CDT**

Bases: *BaseTransform*

Cumulative Distribution Transform.

displacements_

Displacements u.

Type
1d array

transport_map_

Transport map f.

Type
1d array

References

[The cumulative distribution transform and linear pattern classification] (https://arxiv.org/abs/1507.05936)

apply_forward_map(*transport_map*, *sig1*)

Appy forward transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig1** (*1d array*) – Signal to transform.

Returns
sig0_recon – Reconstructed reference signal sig0.

Return type
1d array

apply_inverse_map(*transport_map*, *sig0*, *x*)

Apply inverse transport map.

Parameters

- **transport_map** (*1d array*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*1d array*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
1d array

forward(*x0*, *sig0*, *x1*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **x0** (*1d array*) – Independent axis variable of reference signal (sig0).
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to transform (sig1).
- **sig1** (*1d array*) – Signal to transform.

Returns

- **sig1_cdt** (*1d array*) – CDT of input signal sig1 (new definition).
- **sig1_hat** (*1d array*) – old definition.
- **xilde** (*1d array*) – Independent axis variable in CDT space.

inverse(*transport_map*, *sig0*, *x1*)

Inverse transform.

Parameters

- **transport_map** (*1d array*) – Forward transport map.
- **sig0** (*1d array*) – Reference signal.
- **x1** (*1d array*) – Independent axis variable of the signal to reconstruct.

Returns
sig1_recon – Reconstructed signal.

Return type
1d array

class pytranskit.optrans.continuous.radoncdt3D.**RadonCDT3D**(*Npoints=1024*, *use_gpu=False*)

Bases: *BaseTransform*

3D Radon Cumulative Distribution Transform.

Parameters

- **Npoints** (*scaler, number of radon projections*) –
- **use_gpu** (*boolean, use GPU if True*) –

apply_inverse_map(*transport_map*, *sig0*, *x_range*)

Appy inverse transport map.

Parameters

- **transport_map** (*2d array, shape (t, N)*) – Forward transport map. Inverse is computed in this function.
- **sig0** (*array, shape (height, width, depth)*) – Reference signal.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

dRamp(*x*, *h*, *d=3*)

forward(*x0_range*, *sig0*, *x1_range*, *sig1*, *rm_edge=False*)

Forward transform.

Parameters

- **sig0** (*array, shape (height, width, depth)*) – Reference image.
- **sig1** (*array, shape (height, width, depth)*) – Signal to transform.

Returns
rcdt – 3D Radon-CDT of input image sig1.

Return type
array, shape (t, N)

get_rotation_matrix(*phi*, *theta*, *forward=True*)

Calculates the Rotation matrix for, Input:

phi: Rotation along the x-axis (1,0,0) in degrees theta: Rotation along the y-axis (0,1,0) in degrees

output:
M: The rotation matrix

inverse(*transport_map*, *sig0*, *x1*)

Inverse transform.

Returns
sig1_recon – Reconstructed signal sig1.

Return type
array, shape (height, width)

iradon_3D(*img3Dhat*, *phis*, *thetas*)

radon_3D(*img3D*, *N*)

rotate3D(*img3D*, *phi*, *theta*, *forward=True*)

Rotates an image around the x and y axis with, Inputs:

img3D: Input image numpy 3D phi: Angle rotation around x theta: Angle rotation around y

Returns
Rotated image (numpy 3D)

Return type
img3D_rot

sample_sphere(*num_pts*, *return_type='spherical'*)

This function "uniformly" samples a sphere on num_pts Inputs:

num_pts= number of points to sample return_type = return points in 'spherical' or 'cartesian' coordinates

pytranskit.optrans.continuous.radoncdt3D.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.continuous.radoncdt3D.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns

array_converted – The converted and validated array.

Return type

object

pytranskit.optrans.continuous.radoncdt3D.**fft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)`, where

`y(j) = (x * exp(-2*pi*sqrt(-1)*j*np.arange(n)/n)).sum()`.

Parameters

- **x** (*array_like*) – Array to Fourier transform.
- **n** (*int, optional*) – Length of the Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the fft's are computed; the default is over the last axis (i.e., `axis=-1`).
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed; the default is False.

Returns

z –

with the elements:

```
[y(0),y(1),..,y(n/2),y(1-n/2),...,y(-1)]        if n is even
[y(0),y(1),..,y((n-1)/2),y(-(n-1)/2),...,y(-1)]  if n is odd
```

where:

```
y(j) = sum[k=0..n-1] x[k] * exp(-sqrt(-1)*j*k* 2*pi/n), j = 0..n-1
```

Return type

complex ndarray

See also:

ifft
Inverse FFT

rfft
FFT of a real sequence

Notes

The packing of the result is "standard": If `A = fft(a, n)`, then `A[0]` contains the zero-frequency term, `A[1: n/2]` contains the positive-frequency terms, and `A[n/2:]` contains the negative-frequency terms, in order of decreasingly negative frequency. So ,for an 8-point transform, the frequencies of the result are [0, 1, 2, 3, -4, -3, -2, -1]. To rearrange the fft output so that the zero-frequency component is centered, like [-4, -3, -2, -1, 0, 1, 2, 3], use *fftshift*.

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non-floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

Note that if `x` is real-valued, then `A[j] == A[n-j].conjugate()`. If `x` is real-valued and `n` is even, then `A[n/2]` is real.

If the data type of *x* is real, a "real FFT" algorithm is automatically used, which roughly halves the computation time. To increase efficiency a little further, use *rfft*, which does the same calculation, but only outputs half of the symmetrical spectrum. If the data is both real and symmetrical, the *dct* can again double the efficiency by generating half of the spectrum from half of the signal.

Examples

```
>>> from scipy.fftpack import fft, ifft
>>> x = np.arange(5)
>>> np.allclose(fft(ifft(x)), x, atol=1e-15)  # within numerical accuracy.
True
```

pytranskit.optrans.continuous.radoncdt3D.**fftshift**(*x*, *axes=None*)

Shift the zero-frequency component to the center of the spectrum.

This function swaps half-spaces for all axes listed (defaults to all). Note that `y[0]` is the Nyquist component only if `len(x)` is even.

Parameters

- **x** (*array_like*) – Input array.
- **axes** (*int or shape tuple, optional*) – Axes over which to shift. Default is None, which shifts all axes.

Returns
y – The shifted array.

Return type
ndarray

See also:

ifftshift
The inverse of *fftshift*.

Examples

```
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2., ..., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

Shift the zero-frequency component only along the second axis:

```
>>> freqs = np.fft.fftfreq(9, d=1./9).reshape(3, 3)
>>> freqs
array([[ 0.,  1.,  2.],
       [ 3.,  4., -4.],
       [-3., -2., -1.]])
>>> np.fft.fftshift(freqs, axes=(1,))
array([[ 2.,  0.,  1.],
       [-4.,  3.,  4.],
       [-1., -3., -2.]])
```

pytranskit.optrans.continuous.radoncdt3D.**ifft**(*x*, *n=None*, *axis=-1*, *overwrite_x=False*)

Return discrete inverse Fourier transform of real or complex sequence.

The returned complex array contains `y(0), y(1),..., y(n-1)`, where

`y(j) = (x * exp(2*pi*sqrt(-1)*j*np.arange(n)/n)).mean()`.

Parameters

- **x** (*array_like*) – Transformed data to invert.
- **n** (*int, optional*) – Length of the inverse Fourier transform. If `n < x.shape[axis]`, *x* is truncated. If `n > x.shape[axis]`, *x* is zero-padded. The default results in `n = x.shape[axis]`.
- **axis** (*int, optional*) – Axis along which the ifft's are computed; the default is over the last axis (i.e., `axis=-1`).
- **overwrite_x** (*bool, optional*) – If True, the contents of *x* can be destroyed; the default is False.

Returns

ifft – The inverse discrete Fourier transform.

Return type

ndarray of floats

See also:

fft

Forward FFT

Notes

Both single and double precision routines are implemented. Half precision inputs will be converted to single precision. Non-floating-point inputs will be converted to double precision. Long-double precision inputs are not supported.

This function is most efficient when *n* is a power of two, and least efficient when *n* is prime.

If the data type of *x* is real, a "real IFFT" algorithm is automatically used, which roughly halves the computation time.

Examples

```
>>> from scipy.fftpack import fft, ifft
>>> import numpy as np
>>> x = np.arange(5)
>>> np.allclose(ifft(fft(x)), x, atol=1e-15)  # within numerical accuracy.
True
```

pytranskit.optrans.continuous.radoncdt3D.**signal_to_pdf**(*input*, *sigma=0.0*, *epsilon=1e-08*, *total=1.0*)

Get the (smoothed) probability density function of a signal.

Performs the following operations: 1. Smooth sigma with a Gaussian filter 2. Normalize signal such that it sums to 1 3. Add epsilon to ensure signal is strictly positive 4. Re-normalize signal such that it sums to total

Parameters

- **input** (*ndarray*) – Input array
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **epsilon** (*scalar*) – Offset to ensure that signal is strictly positive.
- **total** (*scalar*) – Value of the signal summation.

Returns
pdf – Returned array of same shape as input

Return type
ndarray

5.1.6 scdt

SCDT.py

Class for computing the Signed Cumulative Distribution Transform (SCDT).

Authors: Sumati Thareja D.M. Rocio Ivan Medri Akram Aldroubi Gustavo Rohde

Based on paper: ArXiv paper link

class pytranskit.optrans.continuous.scdt.**SCDT**(*reference*, *x0=None*)

Bases: object

Signed Cumulative Distribution Transform (SCDT)

Parameters

- **reference** (*1D array representing the reference density*) –
- **x0** (*domain of reference*) –
- **reference_CDF** (*reference CDF*) –
- **xtilde** (*Domain of the reference's inverse CDF*) –
- **reference_CDT_inverse** (*inverse CDF of reference*) –

calc_scdt(*sig1*, *t1*, *s0*, *t0*)

gen_inverse(*f*, *dom_f*, *dom_gf*)

gen_inverse calculates the generalized inverse of the function f input:

f: A one dimensional function represented by a vector dom_f: The domain of the function f dom_gf: The domain of the generalizd inverse of f

output:
The generalized inverse of f

istransform(*Iposhat*, *Ineghat*, *Masspos*, *Massneg*)

istrasform calculates the inverse of the stransform. input:

The 4 components of the transport transform for signed signals

output:
The original signal

itransform(*Ihat*)

itransform calculates the inverse of the CDT. It receives a transport displacement and the reference, and finds the one dimensional distribution I from it. input:

Transport displacement map The reference used for calculating the CDT

output:
I: The original distribution

stransform(*I*, *x=None*)

stransform calculates the transport transform (CDT) of a signal I for signals that may change sign input:

The original density I x -> domain of the density I

output:
The 4 components of the transform of signed signals: the CDT of the positive and the negative part of I, and the total masses of the positive and the negative part of I

transform(*I*)

transform calculates the transport map that morphs the one-dimensional distribution I into the reference. input:

I: A one dimensional distributions of size self.dim

output:
The CDT transformation of I

pytranskit.optrans.continuous.scdt.**interp**(*x*, *xp*, *fp*, *left=None*, *right=None*, *period=None*)

One-dimensional linear interpolation for monotonically increasing sample points.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (*xp*, *fp*), evaluated at *x*.

Parameters

- **x** (*array_like*) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (*1-D sequence of floats*) – The x-coordinates of the data points, must be increasing if argument *period* is not specified. Otherwise, *xp* is internally sorted after normalizing the periodic boundaries with `xp = xp % period`.
- **fp** (*1-D sequence of float or complex*) – The y-coordinates of the data points, same length as *xp*.
- **left** (*optional float or complex corresponding to fp*) – Value to return for *x* < *xp[0]*, default is *fp[0]*.
- **right** (*optional float or complex corresponding to fp*) – Value to return for *x* > *xp[-1]*, default is *fp[-1]*.
- **period** (*None or float, optional*) – A period for the x-coordinates. This parameter allows the proper interpolation of angular x-coordinates. Parameters *left* and *right* are ignored if *period* is specified.

  New in version 1.10.0.

Returns

y – The interpolated values, same shape as *x*.

Return type

float or complex (corresponding to fp) or ndarray

Raises

ValueError – If *xp* and *fp* have different length If *xp* or *fp* are not 1-D sequences If *period* == *0*

See also:

`scipy.interpolate`

Warning: The x-coordinate sequence is expected to be increasing, but this is not explicitly enforced. However, if the sequence *xp* is non-increasing, interpolation results are meaningless.

Note that, since NaN is unsortable, *xp* also cannot contain NaNs.

A simple check for *xp* being strictly increasing is:

```
np.all(np.diff(xp) > 0)
```

Examples

```
>>> xp = [1, 2, 3]
>>> fp = [3, 2, 0]
>>> np.interp(2.5, xp, fp)
1.0
>>> np.interp([0, 1, 1.5, 2.72, 3.14], xp, fp)
array([3.  , 3.  , 2.5 , 0.56, 0.  ])
>>> UNDEF = -99.0
>>> np.interp(3.14, xp, fp, right=UNDEF)
-99.0
```

Plot an interpolant to the sine function:

```
>>> x = np.linspace(0, 2*np.pi, 10)
>>> y = np.sin(x)
>>> xvals = np.linspace(0, 2*np.pi, 50)
>>> yinterp = np.interp(xvals, x, y)
>>> import matplotlib.pyplot as plt
>>> plt.plot(x, y, 'o')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.plot(xvals, yinterp, '-x')
[<matplotlib.lines.Line2D object at 0x...>]
>>> plt.show()
```

Interpolation with periodic x-coordinates:

```
>>> x = [-180, -170, -185, 185, -10, -5, 0, 365]
>>> xp = [190, -190, 350, -350]
>>> fp = [5, 10, 3, 4]
>>> np.interp(x, xp, fp, period=360)
array([7.5 , 5.  , 8.75, 6.25, 3.  , 3.25, 3.5 , 3.75])
```

Complex interpolation:

```
>>> x = [1.5, 4.0]
>>> xp = [2,3,5]
>>> fp = [1.0j, 0, 2+3j]
>>> np.interp(x, xp, fp)
array([0.+1.j , 1.+1.5j])
```

5.2 pytranskit.optrans.decomposition package

5.2.1 CCA

class pytranskit.optrans.decomposition.cca.**CCA**(*n_components=1*, *scale=True*, *max_iter=500*, *tol=1e-06*, *copy=True*)

Bases: `object`

Canonical Correlation Analysis.

This is a wrapper for scikit-learn's CCA class, which allows it to be used in a similar manner to PLDA and PCA.

Parameters

- **n_components** (*int (default=1)*) – Number of components to keep.
- **scale** (*bool (default=True)*) – Whether to scale the data?
- **max_iter** (*int (default=500)*) – The maximum number of iterations of the NIPALS inner loop.
- **tol** (*float (default=1e-6)*) – The tolerance used in the iterative algorithm.
- **copy** (*bool (default=True)*) – Whether the deflation be done on a copy. Let the default value to True unless you don't care about side effects.

components_

X block weights vectors.

Type
array, shape (n_components, n_features)

components_y_

Y block weights vectors.

Type
array, shape (n_components, n_targets)

explained_variance_

The amount of variance explained by each of the selected weights for the X data.

Type
array, shape (n_components,)

explained_variance_y_

The amount of variance explained by each of the selected weights for the Y data.

Type
array, shape (n_components,)

mean_

Per-feature empirical mean of X, estimated from the training set.

Type
array, shape (n_features,)

mean_y_

Per-feature empirical mean of Y, estimated from the training set.

Type
array, shape (n_targets,)

n_components_

The number of components.

Type
int

References

[scikit-learn's documentation on CCA] (http://scikit-learn.org/stable/modules/generated/sklearn.cross_decomposition.CCA.html) Jacob A. Wegelin. A survey of Partial Least Squares (PLS) methods, with emphasis on the two-block case. Technical Report 371, Department of Statistics, University of Washington, Seattle, 2000.

`fit`(*X*, *Y*)

Fit model to data.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Training vectors, where n_samples is the number of samples and n_features is the number of predictors.
- **Y** (*array, shape (n_samples, n_targets)*) – Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

`fit_transform`(*X*, *Y*)

Learn and apply the dimension reduction on the train data.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Training vectors, where n_samples is the number of samples and n_features is the number of predictors.
- **Y** (*array, shape (n_samples, n_targets)*) – Target vectors, where n_samples is the number of samples and n_targets is the number of response variables.

Returns

- **X_new** (*array, shape (n_samples, n_components)*) – Transformed X data.
- **Y_new** (*array, shape (n_samples, n_components)*) – Transformed Y data.

`inverse_transform`(*X*, *Y=None*)

Transform data back to its original space.

Note: This is not exact!

Parameters

- **X** (*array, shape (n_samples, n_components)*) – Transformed X data.
- **Y** (*array, shape (n_samples, n_components) or None (default=None)*) – Transformed Y data. If Y=None, only the X data are transformed back to the original space.

Returns

- **X_original** (*array, shape (n_samples, n_features)*) – X data transformed back into original space.
- **Y_original** (*array, shape (n_samples, n_targets)*) – Y data transformed back into original space. If Y=None, only X_original is returned.

`score`(*X*, *Y*)

Return Pearson product-moment correlation coefficients for each component.

The values of R are between -1 and 1, inclusive.

Note: This is different from sklearn.cross_decomposition.CCA.score(), which returns the coefficient of determination of the prediction.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Input X data.
- **Y** (*array, shape (n_samples, n_targets) or None (default=None)*) – Input Y data.

Returns

score – Pearson product-moment correlation coefficients. If n_components=1, a single value is returned, else an array of correlation coefficients is returned.

Return type

float or array, shape (n_components,)

transform(*X*, *Y=None*)

Apply the dimension reduction learned on the train data.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Input X data.
- **Y** (*array, shape (n_samples, n_targets) or None (default=None)*) – Input Y data. If Y=None, then only the transformed X data are returned.

Returns

- **X_new** (*array, shape (n_samples, n_components)*) – Transformed X data.
- **Y_new** (*array, shape (n_samples, n_components)*) – Transformed Y data. If Y=None, only X_new is returned.

pytranskit.optrans.decomposition.cca.**CanonCorr**

alias of CCA

pytranskit.optrans.decomposition.cca.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns

array_converted – The converted and validated array.

Return type

object

5.2.2 PLDA

class pytranskit.optrans.decomposition.plda.**BaseEstimator**

Bases: object

Base class for all estimators in scikit-learn.

Notes

All estimators should specify all the parameters that can be set at the class level in their __init__ as explicit keyword arguments (no *args or **kwargs).

get_params(*deep=True*)

Get parameters for this estimator.

Parameters
deep (*bool, default=True*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns
params – Parameter names mapped to their values.

Return type
dict

set_params(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as Pipeline). The latter have parameters of the form <component>__<parameter> so that it's possible to update each component of a nested object.

Parameters
****params** (*dict*) – Estimator parameters.

Returns
self – Estimator instance.

Return type
estimator instance

class pytranskit.optrans.decomposition.plda.**PLDA**(*alpha=1.0*, *n_components=None*)

Bases: *BaseEstimator*

Penalized Linear Discriminant Analysis.

This is both a dimensionality reduction method and a linear classifier.

Parameters

- **alpha** (*scalar (default=1.)*) – Parameter that controls the proportion of LDA vs PCA. If alpha=0, PLDA functions like LDA. If alpha is large, PLDA functions more like PCA.
- **n_components** (*int or None (default=None)*) – Number of components to keep. If n_components is not set, all components are kept: n_components == min(n_samples, n_features).

components_

Axes in the feature space. The components are sorted by the explained variance.

Type
array, shape (n_components, n_features)

`explained_variance_`

The amount of variance explained by each of the selected components.

Type
array, shape (n_components,)

`explained_variance_ratio_`

Proportion of variance explained by each of the selected components. If n_components is not set then all components are stored and the sum of explained variance ratios is equal to 1.0.

Type
array, shape(n_components,)

`mean_`

Per-feature empirical mean, estimated from the training set.

Type
array, shape (n_features,)

`n_components_`

The number of components.

Type
int

`coef_`

Weight vector(s).

Type
array, shape (n_features,) or (n_classes, n_features)

`intercept_`

Intercept term.

Type
array, shape (n_features,)

`class_means_`

Class means, estimated from the training set.

Type
array, shape (n_classes, n_features)

`classes_`

Unique class labels.

Type
array, shape (n_classes,)

References

W. Wang et al. Penalized Fisher Discriminant Analysis and its Application to Image-Based Morphometry. Pattern Recognit. Lett., 32(15):2128-35, 2011

decision_function(*X*)

Predict confidence scores for samples.

The confidence score for a sample is the signed distance of that sample to the hyperplane.

Parameters
X (*array, shape (n_samples, n_features)*) – Input data.

Returns
scores – else (n_samples, n_classes) Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where >0 means this class would be predicted.

Return type
array, shape=(n_samples,) if n_classes == 2

fit(*X*, *y*)

Fit PLDA model according to the given training data and parameters.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Training data.
- **y** (*array, shape (n_samples,)*) – Target values.

fit_transform(*X*, *y*)

Fit the model with X and transform X.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Training data.
- **y** (*array, shape (n_samples,)*) – Target values.

Returns
X_new – Transformed data.

Return type
array, shape (n_samples, n_components)

inverse_transform(*X*)

Transform data back to its original space.

Note: If n_components is less than the maximum, information will be lost, so reconstructed data will not exactly match the original data.

Parameters
X (*array shape (n_samples, n_components)*) – New data.

Returns
X_original – Data transformed back into original space.

Return type
array, shape (n_samples, n_features)

predict(*X*)

Predict class labels for samples in X.

Parameters
X (*array, shape (n_samples, n_features)*) – Input data.

Returns
C – Predicted class label per sample.

Return type
array, shape (n_samples,)

`predict_log_proba`(*X*)

Estimate log probability.

Parameters
X (*array, shape (n_samples, n_features)*) – Input data.

Returns
C – Estimated log probabilities.

Return type
array, shape (n_samples, n_classes)

`predict_proba`(*X*)

Estimate probability.

Parameters
X (*array, shape (n_samples, n_features)*) – Input data.

Returns
C – Estimated probabilities.

Return type
array, shape (n_samples, n_classes)

`predict_transformed`(*X_trans*)

Predict class labels for data that have already been transformed by self.transform(X).

This is useful for plotting classification boundaries. Note: Due to arithemtic discrepancies, this may return slightly different class labels to self.predict(X).

Parameters
X_trans (*array, shape (n_samples, n_components)*) – Test samples that have already been transformed into PLDA space.

Returns
y – Predicted class labels for X_trans.

Return type
array, shape (n_samples,)

`score`(*X*, *y*, *sample_weight=None*)

Returns the mean accuracy on the given test data and labels.

Parameters

- **X** (*array, shape (n_samples, n_features)*) – Test samples.
- **y** (*array, shape (n_samples,)*) – True labels for X.
- **sample_weight** (*array, shape (n_samples,), optional*) – Sample weights.

Returns
score – Mean accuracy of self.predict(X) w.r.t. y.

Return type
float

`transform`(*X*)

Transform data.

Parameters
X (*array, shape (n_samples, n_features)*) – Input data.

Returns
X_new – Transformed data.

Return type
array, shape (n_samples, n_components)

`pytranskit.optrans.decomposition.plda.accuracy_score`(*y_true*, *y_pred*, ***, *normalize=True*, *sample_weight=None*)

Accuracy classification score.

In multilabel classification, this function computes subset accuracy: the set of labels predicted for a sample must *exactly* match the corresponding set of labels in y_true.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) labels.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted labels, as returned by a classifier.
- **normalize** (*bool, default=True*) – If `False`, return the number of correctly classified samples. Otherwise, return the fraction of correctly classified samples.
- **sample_weight** (*array-like of shape (n_samples,), default=None*) – Sample weights.

Returns

score – If `normalize == True`, return the fraction of correctly classified samples (float), else returns the number of correctly classified samples (int).

The best performance is 1 with `normalize == True` and the number of samples with `normalize == False`.

Return type
float

See also:

`balanced_accuracy_score`
Compute the balanced accuracy to deal with imbalanced datasets.

`jaccard_score`
Compute the Jaccard similarity coefficient score.

`hamming_loss`
Compute the average Hamming loss or Hamming distance between two sets of samples.

`zero_one_loss`
Compute the Zero-one classification loss. By default, the function will return the percentage of imperfectly predicted subsets.

Notes

In binary classification, this function is equal to the *jaccard_score* function.

Examples

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = [0, 2, 1, 3]
>>> y_true = [0, 1, 2, 3]
>>> accuracy_score(y_true, y_pred)
0.5
>>> accuracy_score(y_true, y_pred, normalize=False)
2
```

In the multilabel case with binary label indicators:

```
>>> import numpy as np
>>> accuracy_score(np.array([[0, 1], [1, 1]]), np.ones((2, 2)))
0.5
```

pytranskit.optrans.decomposition.plda.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns

array_converted – The converted and validated array.

Return type

object

pytranskit.optrans.decomposition.plda.**eigh**(*a*, *b=None*, *lower=True*, *eigvals_only=False*, *overwrite_a=False*, *overwrite_b=False*, *turbo=True*, *eigvals=None*, *type=1*, *check_finite=True*, *subset_by_index=None*, *subset_by_value=None*, *driver=None*)

Solve a standard or generalized eigenvalue problem for a complex Hermitian or real symmetric matrix.

Find eigenvalues array `w` and optionally eigenvectors array `v` of array `a`, where `b` is positive definite such that for every eigenvalue (i-th entry of w) and its eigenvector `vi` (i-th column of `v`) satisfies:

```
              a @ vi =  * b @ vi
vi.conj().T @ a @ vi =
vi.conj().T @ b @ vi = 1
```

In the standard problem, b is assumed to be the identity matrix.

Parameters

- **a** (*(M, M) array_like*) – A complex Hermitian or real symmetric matrix whose eigenvalues and eigenvectors will be computed.
- **b** (*(M, M) array_like, optional*) – A complex Hermitian or real symmetric definite positive matrix in. If omitted, identity matrix is assumed.
- **lower** (*bool, optional*) – Whether the pertinent array data is taken from the lower or upper triangle of a and, if applicable, b. (Default: lower)
- **eigvals_only** (*bool, optional*) – Whether to calculate only eigenvalues and no eigenvectors. (Default: both are calculated)
- **subset_by_index** (*iterable, optional*) – If provided, this two-element iterable defines the start and the end indices of the desired eigenvalues (ascending order and 0-indexed). To return only the second smallest to fifth smallest eigenvalues, `[1, 4]` is used. `[n-3, n-1]` returns the largest three. Only available with "evr", "evx", and "gvx" drivers. The entries are directly converted to integers via `int()`.
- **subset_by_value** (*iterable, optional*) – If provided, this two-element iterable defines the half-open interval `(a, b]` that, if any, only the eigenvalues between these values are returned. Only available with "evr", "evx", and "gvx" drivers. Use `np.inf` for the unconstrained ends.
- **driver** (*str, optional*) – Defines which LAPACK driver should be used. Valid options are "ev", "evd", "evr", "evx" for standard problems and "gv", "gvd", "gvx" for generalized (where b is not None) problems. See the Notes section.
- **type** (*int, optional*) – For the generalized problems, this keyword specifies the problem type to be solved for w and v (only takes 1, 2, 3 as possible inputs):

  ```
  1 =>     a @ v = w @ b @ v
  2 => a @ b @ v = w @ v
  3 => b @ a @ v = w @ v
  ```

  This keyword is ignored for standard problems.
- **overwrite_a** (*bool, optional*) – Whether to overwrite data in a (may improve performance). Default is False.
- **overwrite_b** (*bool, optional*) – Whether to overwrite data in b (may improve performance). Default is False.
- **check_finite** (*bool, optional*) – Whether to check that the input matrices contain only finite numbers. Disabling may give a performance gain, but may result in problems (crashes, non-termination) if the inputs do contain infinities or NaNs.
- **turbo** (*bool, optional*) – *Deprecated since v1.5.0, use ``driver=gvd`` keyword instead.* Use divide and conquer algorithm (faster but expensive in memory, only for generalized eigenvalue problem and if full set of eigenvalues are requested.). Has no significant effect if eigenvectors are not requested.
- **eigvals** (*tuple (lo, hi), optional*) – *Deprecated since v1.5.0, use ``subset_by_index`` keyword instead.* Indexes of the smallest and largest (in ascending

order) eigenvalues and corresponding eigenvectors to be returned: 0 <= lo <= hi <= M-1. If omitted, all eigenvalues and eigenvectors are returned.

Returns

- **w** (*(N,) ndarray*) – The N (1<=N<=M) selected eigenvalues, in ascending order, each repeated according to its multiplicity.
- **v** (*(M, N) ndarray*) – (if `eigvals_only == False`)

Raises

LinAlgError – If eigenvalue computation does not converge, an error occurred, or b matrix is not definite positive. Note that if input matrices are not symmetric or Hermitian, no error will be reported but results will be wrong.

See also:

eigvalsh
eigenvalues of symmetric or Hermitian arrays

eig
eigenvalues and right eigenvectors for non-symmetric arrays

eigh_tridiagonal
eigenvalues and right eiegenvectors for symmetric/Hermitian tridiagonal matrices

Notes

This function does not check the input array for being Hermitian/symmetric in order to allow for representing arrays with only their upper/lower triangular parts. Also, note that even though not taken into account, finiteness check applies to the whole array and unaffected by "lower" keyword.

This function uses LAPACK drivers for computations in all possible keyword combinations, prefixed with `sy` if arrays are real and `he` if complex, e.g., a float array with "evr" driver is solved via "syevr", complex arrays with "gvx" driver problem is solved via "hegvx" etc.

As a brief summary, the slowest and the most robust driver is the classical `<sy/he>ev` which uses symmetric QR. `<sy/he>evr` is seen as the optimal choice for the most general cases. However, there are certain occasions that `<sy/he>evd` computes faster at the expense of more memory usage. `<sy/he>evx`, while still being faster than `<sy/he>ev`, often performs worse than the rest except when very few eigenvalues are requested for large arrays though there is still no performance guarantee.

For the generalized problem, normalization with respect to the given type argument:

```
type 1 and 3 :      v.conj().T @ a @ v = w
type 2       : inv(v).conj().T @ a @ inv(v) = w

type 1 or 2  :      v.conj().T @ b @ v  = I
type 3       : v.conj().T @ inv(b) @ v  = I
```

Examples

```
>>> from scipy.linalg import eigh
>>> A = np.array([[6, 3, 1, 5], [3, 0, 5, 1], [1, 5, 6, 2], [5, 1, 2, 2]])
>>> w, v = eigh(A)
>>> np.allclose(A @ v - v @ np.diag(w), np.zeros((4, 4)))
True
```

Request only the eigenvalues

```
>>> w = eigh(A, eigvals_only=True)
```

Request eigenvalues that are less than 10.

```
>>> A = np.array([[34, -4, -10, -7, 2],
...               [-4, 7, 2, 12, 0],
...               [-10, 2, 44, 2, -19],
...               [-7, 12, 2, 79, -34],
...               [2, 0, -19, -34, 29]])
>>> eigh(A, eigvals_only=True, subset_by_value=[-np.inf, 10])
array([6.69199443e-07, 9.11938152e+00])
```

Request the largest second eigenvalue and its eigenvector

```
>>> w, v = eigh(A, subset_by_index=[1, 1])
>>> w
array([9.11938152])
>>> v.shape  # only a single column is returned
(5, 1)
```

5.3 pytranskit.optrans.utils package

5.3.1 data_utils

pytranskit.optrans.utils.data_utils.**griddata2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolate 2d scattered data

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate.
- **f** (*3d array, shape (2, height, width)*) – Coordinates of at which img is defined. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the interpolation. Must be in the range 0-2.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns

Image interpolated on to regular gird defined by: x = 0:(width-1), y = 0:(height-1).

Return type

out, 2d array, shape (height, width)

pytranskit.optrans.utils.data_utils.**interp2d**(*img*, *f*, *order=1*, *fill_value=0.0*)

Interpolation for 2D gridded data.

Parameters

- **img** (*2d array, shape (height, width)*) – Image to interpolate. This function assumes a grid of sample points: x = 0:(width-1), y = 0:(height-1).
- **f** (*3d array, shape (2, height, width)*) – Coordinates of interpolated points. First dimension f[0] corresponds to y-coordinates, second dimension f[1] is x-coordinates.
- **order** (*int (default=1)*) – Order of the spline interpolation. Must be in the range 0-5.
- **fill_value** (*float (default=0.)*) – Value used for points outside the boundaries.

Returns
Image interpolated at points defined by f.

Return type
out, 2d array, shape (height, width)

pytranskit.optrans.utils.data_utils.**match_shape2d**(*a*, *b*)

Crop array B such that it matches the shape of A.

Parameters

- **a** (*2d array*) – Array of desired size.
- **b** (*2d array*) – Array to crop. Shape must be larger than (or equal to) the shape of array a.

Returns
b_crop – Cropped version of b, with the same shape as a.

Return type
2d array

pytranskit.optrans.utils.data_utils.**signal_to_pdf**(*input*, *sigma=0.0*, *epsilon=1e-08*, *total=1.0*)

Get the (smoothed) probability density function of a signal.

Performs the following operations: 1. Smooth sigma with a Gaussian filter 2. Normalize signal such that it sums to 1 3. Add epsilon to ensure signal is strictly positive 4. Re-normalize signal such that it sums to total

Parameters

- **input** (*ndarray*) – Input array
- **sigma** (*scalar*) – Standard deviation for Gaussian kernel. The standard deviations of the Gaussian filter are given for each axis as a sequence, or as a single number, in which case it is equal for all axes.
- **epsilon** (*scalar*) – Offset to ensure that signal is strictly positive.
- **total** (*scalar*) – Value of the signal summation.

Returns
pdf – Returned array of same shape as input

Return type
ndarray

5.3.2 validation

pytranskit.optrans.utils.validation.**assert_all_finite**(*X*)

Throw a ValueError if X contains NaN or infinity.

pytranskit.optrans.utils.validation.**assert_equal_shape**(*a*, *b*, *names=None*)

Throw a ValueError if a and b are not the same shape.

pytranskit.optrans.utils.validation.**check_array**(*array*, *ndim=None*, *dtype='numeric'*, *force_all_finite=True*, *force_strictly_positive=False*)

Input validation on an array, list, or similar.

Parameters

- **array** (*object*) – Input object to check/convert
- **ndim** (*int or None (default=None)*) – Number of dimensions that array should have. If None, the dimensions are not checked
- **dtype** (*string, type, list of types or None (default='numeric')*) – Data type of result. If None, the dtype of the input is preserved. If 'numeric', dtype is preserved unless array.dtype is object. If dtype is a list of types, conversion on the first type is only performed if the dtype of the input is not in the list.
- **force_all_finite** (*boolean (default=True)*) – Whether to raise an error on np.inf and np.nan in array
- **force_strictly_positive** (*boolean (default=False)*) – Whether to raise an error if any array elements are <= 0

Returns

array_converted – The converted and validated array.

Return type

object

pytranskit.optrans.utils.validation.**check_decomposition**(*obj*)

Check that an object is a PCA or PLDA (i.e. decomposition) object.

Parameters

obj (*object*) – Object to check

Returns

- **mean** (*array, shape (n_features,)*) – Mean of the data in the decomposition object
- **components** (*array, shape (n_components, n_features)*) – Components learned by the decomposition object
- **std** (*array, shape (n_components,)*) – Standard deviation of the training data projected on to each component

5.3.3 visualize

pytranskit.optrans.utils.visualize.**plot_displacements2d**(*disp*, *ax=None*, *scale=1.0*, *count=50*, *lw=1*, *c='k'*)

Plot 2D pixel displacements as a wireframe grid.

Parameters

- **disp** (*array, shape (2, height, width)*) – Pixel displacements. First index denotes direction: disp[0] is y-displacements, and disp[1] is x-displacements.
- **ax** (*matplotlib.axes.Axes object or None (default=None)*) – Axes in which to plot the wireframe. If None, a new figure is created.
- **scale** (*float (default=1.)*) – Exaggeration scale applied to the displacements before visualization.
- **count** (*int (default=50)*) – Use at most this many rows and columns in the wireframe.

Returns
ax – Axes object.

Return type
matplotlib.axes.Axes object

5.4 pytranskit.classification package

5.4.1 cdt_ns

class pytranskit.classification.cdt_ns.**CDT_NS**(*num_classes*, *rm_edge=False*)

Bases: object

add_trans_samples(*cdt_features*)

cdt_parallel(*X*)

fit(*Xtrain*, *Ytrain*, *no_deform_model=False*)

Fit linear model. :param Xtrain: 1D data for training. :type Xtrain: array-like, shape (n_samples, n_columns) :param Ytrain: Labels of the training samples. :type Ytrain: ndarray of shape (n_samples,) :param no_deform_model: default = False. :type no_deform_model: boolean flag; IF TRUE, no deformation model will be added

fun_cdt_batch(*data*)

fun_cdt_single(*sig1*)

predict(*Xtest*, *use_gpu=False*)

Predict using the linear model :param Xtest: 1D data for testing. :type Xtest: array-like, shape (n_samples, n_columns) :param use_gpu: default = False. :type use_gpu: boolean flag; IF TRUE, use gpu for calculations

Returns
Predicted target values per sample in Xtest.

Return type
ndarray of shape (n_samples,)

5.4.2 rcdt_ns

class pytranskit.classification.rcdt_ns.**RCDT_NS**(*num_classes*, *thetas=array([0.0, 4.0, 8.0, 12.0, 16.0, 20.0, 24.0, 28.0, 32.0, 36.0, 40.0, 44.0, 48.0, 52.0, 56.0, 60.0, 64.0, 68.0, 72.0, 76.0, 80.0, 84.0, 88.0, 92.0, 96.0, 100.0, 104.0, 108.0, 112.0, 116.0, 120.0, 124.0, 128.0, 132.0, 136.0, 140.0, 144.0, 148.0, 152.0, 156.0, 160.0, 164.0, 168.0, 172.0, 176.0])*, *rm_edge=False*)

Bases: object

add_trans_samples(*rcdt_features*)

fit(*Xtrain*, *Ytrain*, *no_deform_model=False*)

Fit linear model.

Parameters

- **Xtrain** (*array-like, shape (n_samples, n_rows, n_columns)*) – Image data for training.
- **Ytrain** (*ndarray of shape (n_samples,)*) – Labels of the training images.
- **no_deform_model** (*boolean flag; IF TRUE, no deformation model will be added*) – default = False.

fun_rcdt_batch(*data*)

fun_rcdt_single(*I*)

predict(*Xtest*, *use_gpu=False*)

Predict using the linear model

Let B^k be the basis vectors of class k, and x be the RCDT sapce feature vector of an input, the NS method performs classification by

$$arg\min_k \|B^k(B^k)^T x - x\|^2$$

Parameters

- **Xtest** (*array-like, shape (n_samples, n_rows, n_columns)*) – Image data for testing.
- **use_gpu** (*boolean flag; IF TRUE, use gpu for calculations*) – default = False.

Returns
Predicted target values per element in Xtest.

Return type
ndarray of shape (n_samples,)

rcdt_parallel(*X*)

5.4.3 rcdt_ns_3d

class pytranskit.classification.rcdt_ns_3d.**RCDT_NS_3D**(*num_classes*, *Npoints=500*, *use_gpu=False*, *rm_edge=False*)

Bases: object

add_trans_samples(*rcdt_features*)

fit(*Xtrain*, *Ytrain*, *no_deform_model=False*)

Fit linear model. :param Xtrain: Image data for training. :type Xtrain: array-like, shape (n_samples, n_rows, n_columns, n_width) :param Ytrain: Labels of the training images. :type Ytrain: ndarray of shape (n_samples,) :param no_deform_model: default = False. :type no_deform_model: boolean flag; IF TRUE, no deformation model will be added

fun_rcdt_batch(*data*)

fun_rcdt_single(*I*)

predict(*Xtest*, *use_gpu=False*)

Predict using the linear model :param Xtest: Image data for testing. :type Xtest: array-like, shape (n_samples, n_rows, n_columns, n_width) :param use_gpu: default = False. :type use_gpu: boolean flag; IF TRUE, use gpu for calculations

Returns
Predicted target values per element in Xtest.

Return type
ndarray of shape (n_samples,)

rcdt_parallel(*X*)

5.4.4 scdt_nls

class pytranskit.classification.scdt_nls.**SCDT_NLS**(*num_classes*, *rm_edge=False*)

Bases: object

calc_scdt(*sig1*, *t1*, *s0*, *t0*)

enrichment(*scdt_features*, *k*)

find_kN(*X*, *y*, *k_range*, *n_range*)

fit(*X*, *Y*, *Ttrain=None*, *no_local_enrichment=True*)

Fit SCDT-NLS. :param X: 1D data for training. :type X: array-like, shape (n_samples, n_columns) :param Y: Labels of the training samples. :type Y: ndarray of shape (n_samples,) :param Ttrain: domain for corresponding training signals. :type Ttrain: [optional] array-like, shape (n_samples, n_columns) :param no_local_enrichment: IF FALSE, apply deformation while searching k samples :type no_local_enrichment: [optional] boolean, default TRUE

predict(*Xtest*, *Ttest=None*, *k=None*, *N=None*)

Predict using SCDT-NLS :param Xtest: 1D data for testing. :type Xtest: array-like, shape (n_samples, n_columns) :param Ttest: domain for corresponding test signals. :type Ttest: [optional] array-like, shape (n_samples, n_columns) :param k: :type k: [pre-tuned parameter] number of closest points to test sample :param N: :type N: [pre-tuned parameter] number of sinusoidal bases used for subspace enrrichment

Returns
Predicted target values per sample in Xtest.

Return type
ndarray of shape (n_samples,)

score(*X*, *y*)

5.4.5 scdt_ns

class pytranskit.classification.scdt_ns.**SCDT_NS**(*num_classes*, *rm_edge=False*)

Bases: object

add_trans_samples(*scdt_features*)

calc_scdt(*sig1*, *t1*, *s0*, *t0*)

fit(*Xtrain*, *Ytrain*, *Ttrain=None*, *no_deform_model=True*)

Fit SCDT-NS. :param Xtrain: 1D data for training. :type Xtrain: array-like, shape (n_samples, n_columns) :param Ytrain: Labels of the training samples. :type Ytrain: ndarray of shape (n_samples,) :param Ttrain: domain for corresponding training signals. :type Ttrain: [optional] array-like, shape (n_samples, n_columns) :param no_deform_model: default = True. :type no_deform_model: [optional] boolean flag; IF TRUE, no deformation model will be added

predict(*Xtest*, *Ttest=None*, *use_gpu=False*)

Predict using SCDT-NS :param Xtest: 1D data for testing. :type Xtest: array-like, shape (n_samples, n_columns) :param Ttest: domain for corresponding test signals. :type Ttest: [optional] array-like, shape (n_samples, n_columns) :param use_gpu: default = False. :type use_gpu: [optional] boolean flag; IF TRUE, use gpu for calculations

Returns
Predicted target values per sample in Xtest.

Return type
ndarray of shape (n_samples,)

5.4.6 utils

pytranskit.classification.utils.**dataset_config**(*dataset*)

pytranskit.classification.utils.**load_data**(*dataset*, *num_classes*, *datadir='data'*)

pytranskit.classification.utils.**load_data_1D**(*dataset*, *num_classes*, *datadir='data'*)

pytranskit.classification.utils.**load_data_3D**(*dataset*, *num_classes*, *datadir='data'*)

pytranskit.classification.utils.**new_index_matrix**(*max_index*, *n_samples_perclass*, *num_classes*, *repeat*, *y_train*)

pytranskit.classification.utils.**take_samples**(*data*, *labels*, *index*, *num_classes*)

pytranskit.classification.utils.**take_train_samples**(*x_train*, *y_train*, *n_samples_perclass*, *num_classes*, *repeat*)

pytranskit.classification.utils.**take_train_val_samples**(*x_train*, *y_train*, *n_samples_perclass*, *num_classes*, *repeat*)

5.5 pytranskit.TBM package

5.5.1 TBM_CLOT

5.5.2 TBM_PLOT

Created on Mon Aug 10 11:14:01 2020

@author: Imaging and Data Science Lab

pytranskit.TBM.TBM_PLOT.**L2_distance**(*a*, *b*)

class pytranskit.TBM.TBM_PLOT.**PLOT_CCA**(*n_components=2*)

Bases: object

plot_cca(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*mean_x_train_hat*, *Intensity*, *directions=5*, *points=5*, *SD_spread=1*)

class pytranskit.TBM.TBM_PLOT.**PLOT_NS_Classifier**(*train_sample=None*, *use_gpu=False*)

Bases: object

classify_PLOT_NS(*x_train*, *y_train*, *x_test*, *y_test*)

fit(*X*, *y*)

Fit linear model. :param X: Training data. :type X: array-like, shape (n_samples, n_proj, n_angles)) :param y: Target values. :type y: ndarray of shape (n_samples,)

Returns
Returns an instance of self.

Return type
self

predict(*X*)

Predict using the linear model :param X: :type X: array-like, sparse matrix, shape (n_samples, n_proj, n_angles))

Returns
Predicted target values per element in X.

Return type
ndarray of shape (n_samples,)

score(*y_test*)

class pytranskit.TBM.TBM_PLOT.**PLOT_PCA**(*n_components=2*)

Bases: object

plot_pca(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*mean_x_train_hat*, *Intensity*, *directions=5*, *points=5*, *SD_spread=2*)

class pytranskit.TBM.TBM_PLOT.**PLOT_PLDA**(*n_components=2*)

Bases: object

plot_plda(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*mean_x_train_hat*, *Intensity*, *directions=5*, *points=5*, *SD_spread=2*)

pytranskit.TBM.TBM_PLOT.**Visualize_LOT**(*Data*, *Intensity*, *Nx*, *Ny*, *scale*)

class pytranskit.TBM.TBM_PLOT.**batch_PLOT**(*Nmasses=50*)

Bases: object

forward_seq(*x_train*, *x_test*, *x_template*)

class pytranskit.TBM.TBM_PLOT.**batch_PLOT_v0**(*Nmasses=50*)

Bases: object

forward_seq(*x_train*, *x_test*)

pytranskit.TBM.TBM_PLOT.**fromInd2Coord**(*ind*, *Ny*)

pytranskit.TBM.TBM_PLOT.**gaussian2D**(*x*, *mean*, *sigma*)

pytranskit.TBM.TBM_PLOT.**get_particles**(*img*, *N*)

pytranskit.TBM.TBM_PLOT.**img2pts_Lloyd**(*img*, *Nmasses*)

pytranskit.TBM.TBM_PLOT.**pLOT_single**(*x_temp*, *x_targ*, *a_temp*, *a_targ*)

pytranskit.TBM.TBM_PLOT.**particle2image**(*x*, *a*, *sigma*, *imgshape*)

This function gets a set of coordinates, x, their amplitude, a, and generates a PDF image using a gaussian kernel

pytranskit.TBM.TBM_PLOT.**particleApproximation**(*imgs*, *Nmasses*)

pytranskit.TBM.TBM_PLOT.**particleApproximation_v0**(*imgs*, *Nmasses*)

pytranskit.TBM.TBM_PLOT.**sub2ind**(*array_shape*, *rows*, *cols*)

5.5.3 TBM_RCDT

Created on Tue Aug 4 22:47:30 2020

@author: Imaging and Data Science Lab

class pytranskit.TBM.TBM_RCDT.**RCDT_CCA**(*n_components=2*)

Bases: object

rcdt_cca(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*directions=5*, *points=5*, *thetas=array([0.0, 1.00558659, 2.01117318, 3.01675978, 4.02234637, 5.02793296, 6.03351955, 7.03910615, 8.04469274, 9.05027933, 10.05586592, 11.06145251, 12.06703911, 13.0726257, 14.07821229, 15.08379888, 16.08938547, 17.09497207, 18.10055866, 19.10614525, 20.11173184, 21.11731844, 22.12290503, 23.12849162, 24.13407821, 25.1396648, 26.1452514, 27.15083799, 28.15642458, 29.16201117, 30.16759777, 31.17318436, 32.17877095, 33.18435754, 34.18994413, 35.19553073, 36.20111732, 37.20670391, 38.2122905, 39.21787709, 40.22346369, 41.22905028, 42.23463687, 43.24022346, 44.24581006, 45.25139665, 46.25698324, 47.26256983, 48.26815642, 49.27374302, 50.27932961, 51.2849162, 52.29050279, 53.29608939, 54.30167598, 55.30726257, 56.31284916, 57.31843575, 58.32402235, 59.32960894, 60.33519553, 61.34078212, 62.34636872, 63.35195531, 64.3575419, 65.36312849, 66.36871508, 67.37430168, 68.37988827, 69.38547486, 70.39106145, 71.39664804, 72.40223464, 73.40782123, 74.41340782, 75.41899441, 76.42458101, 77.4301676, 78.43575419, 79.44134078, 80.44692737, 81.45251397, 82.45810056, 83.46368715, 84.46927374, 85.47486034, 86.48044693, 87.48603352, 88.49162011, 89.4972067, 90.5027933, 91.50837989, 92.51396648, 93.51955307, 94.52513966, 95.53072626, 96.53631285, 97.54189944, 98.54748603, 99.55307263, 100.55865922, 101.56424581, 102.5698324, 103.57541899, 104.58100559, 105.58659218, 106.59217877, 107.59776536, 108.60335196, 109.60893855, 110.61452514, 111.62011173, 112.62569832, 113.63128492, 114.63687151, 115.6424581, 116.64804469, 117.65363128, 118.65921788, 119.66480447, 120.67039106, 121.67597765, 122.68156425, 123.68715084, 124.69273743, 125.69832402, 126.70391061, 127.70949721, 128.7150838, 129.72067039, 130.72625698, 131.73184358, 132.73743017, 133.74301676, 134.74860335, 135.75418994, 136.75977654, 137.76536313, 138.77094972, 139.77653631, 140.78212291, 141.7877095, 142.79329609, 143.79888268, 144.80446927, 145.81005587, 146.81564246, 147.82122905, 148.82681564, 149.83240223, 150.83798883, 151.84357542, 152.84916201, 153.8547486, 154.8603352, 155.86592179, 156.87150838, 157.87709497, 158.88268156, 159.88826816, 160.89385475, 161.89944134, 162.90502793, 163.91061453, 164.91620112, 165.92178771, 166.9273743, 167.93296089, 168.93854749, 169.94413408, 170.94972067, 171.95530726, 172.96089385, 173.96648045, 174.97206704, 175.97765363, 176.98324022, 177.98882682, 178.99441341, 180.0])*, *SD_spread=1*)

class pytranskit.TBM.TBM_RCDT.**RCDT_NS_Classifier**(*train_sample=None*, *use_gpu=False*)

Bases: object

classify_RCDT_NS(*x_train*, *y_train*, *x_test*, *y_test*)

fit(*X*, *y*)

Fit linear model. :param X: Training data. :type X: array-like, shape (n_samples, n_proj, n_angles)) :param y: Target values. :type y: ndarray of shape (n_samples,)

Returns
Returns an instance of self.

Return type
self

predict(*X*)

Predict using the linear model :param X: :type X: array-like, sparse matrix, shape (n_samples, n_proj, n_angles))

Returns
Predicted target values per element in X.

Return type
ndarray of shape (n_samples,)

score(*y_test*)

class pytranskit.TBM.TBM_RCDT.**RCDT_PCA**(*n_components=2*)

Bases: object

rcdt_pca(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*directions=5*, *points=5*, *thetas=array([0.0, 1.00558659, 2.01117318, 3.01675978, 4.02234637, 5.02793296, 6.03351955, 7.03910615, 8.04469274, 9.05027933, 10.05586592, 11.06145251, 12.06703911, 13.0726257, 14.07821229, 15.08379888, 16.08938547, 17.09497207, 18.10055866, 19.10614525, 20.11173184, 21.11731844, 22.12290503, 23.12849162, 24.13407821, 25.1396648, 26.1452514, 27.15083799, 28.15642458, 29.16201117, 30.16759777, 31.17318436, 32.17877095, 33.18435754, 34.18994413, 35.19553073, 36.20111732, 37.20670391, 38.2122905, 39.21787709, 40.22346369, 41.22905028, 42.23463687, 43.24022346, 44.24581006, 45.25139665, 46.25698324, 47.26256983, 48.26815642, 49.27374302, 50.27932961, 51.2849162, 52.29050279, 53.29608939, 54.30167598, 55.30726257, 56.31284916, 57.31843575, 58.32402235, 59.32960894, 60.33519553, 61.34078212, 62.34636872, 63.35195531, 64.3575419, 65.36312849, 66.36871508, 67.37430168, 68.37988827, 69.38547486, 70.39106145, 71.39664804, 72.40223464, 73.40782123, 74.41340782, 75.41899441, 76.42458101, 77.4301676, 78.43575419, 79.44134078, 80.44692737, 81.45251397, 82.45810056, 83.46368715, 84.46927374, 85.47486034, 86.48044693, 87.48603352, 88.49162011, 89.4972067, 90.5027933, 91.50837989, 92.51396648, 93.51955307, 94.52513966, 95.53072626, 96.53631285, 97.54189944, 98.54748603, 99.55307263, 100.55865922, 101.56424581, 102.5698324, 103.57541899, 104.58100559, 105.58659218, 106.59217877, 107.59776536, 108.60335196, 109.60893855, 110.61452514, 111.62011173, 112.62569832, 113.63128492, 114.63687151, 115.6424581, 116.64804469, 117.65363128, 118.65921788, 119.66480447, 120.67039106, 121.67597765, 122.68156425, 123.68715084, 124.69273743, 125.69832402, 126.70391061, 127.70949721, 128.7150838, 129.72067039, 130.72625698, 131.73184358, 132.73743017, 133.74301676, 134.74860335, 135.75418994, 136.75977654, 137.76536313, 138.77094972, 139.77653631, 140.78212291, 141.7877095, 142.79329609, 143.79888268, 144.80446927, 145.81005587, 146.81564246, 147.82122905, 148.82681564, 149.83240223, 150.83798883, 151.84357542, 152.84916201, 153.8547486, 154.8603352, 155.86592179, 156.87150838, 157.87709497, 158.88268156, 159.88826816, 160.89385475, 161.89944134, 162.90502793, 163.91061453, 164.91620112, 165.92178771, 166.9273743, 167.93296089, 168.93854749, 169.94413408, 170.94972067, 171.95530726, 172.96089385, 173.96648045, 174.97206704, 175.97765363, 176.98324022, 177.98882682, 178.99441341, 180.0])*, *SD_spread=1*)

class pytranskit.TBM.TBM_RCDT.**RCDT_PLDA**(*n_components=2*)

Bases: object

rcdt_plda(*x_train_hat*, *y_train*, *x_test_hat*, *y_test*, *template*)

visualize(*directions=5*, *points=5*, *thetas=array([0.0, 1.00558659, 2.01117318, 3.01675978, 4.02234637, 5.02793296, 6.03351955, 7.03910615, 8.04469274, 9.05027933, 10.05586592, 11.06145251, 12.06703911, 13.0726257, 14.07821229, 15.08379888, 16.08938547, 17.09497207, 18.10055866, 19.10614525, 20.11173184, 21.11731844, 22.12290503, 23.12849162, 24.13407821, 25.1396648, 26.1452514, 27.15083799, 28.15642458, 29.16201117, 30.16759777, 31.17318436, 32.17877095, 33.18435754, 34.18994413, 35.19553073, 36.20111732, 37.20670391, 38.2122905, 39.21787709, 40.22346369, 41.22905028, 42.23463687, 43.24022346, 44.24581006, 45.25139665, 46.25698324, 47.26256983, 48.26815642, 49.27374302, 50.27932961, 51.2849162, 52.29050279, 53.29608939, 54.30167598, 55.30726257, 56.31284916, 57.31843575, 58.32402235, 59.32960894, 60.33519553, 61.34078212, 62.34636872, 63.35195531, 64.3575419, 65.36312849, 66.36871508, 67.37430168, 68.37988827, 69.38547486, 70.39106145, 71.39664804, 72.40223464, 73.40782123, 74.41340782, 75.41899441, 76.42458101, 77.4301676, 78.43575419, 79.44134078, 80.44692737, 81.45251397, 82.45810056, 83.46368715, 84.46927374, 85.47486034, 86.48044693, 87.48603352, 88.49162011, 89.4972067, 90.5027933, 91.50837989, 92.51396648, 93.51955307, 94.52513966, 95.53072626, 96.53631285, 97.54189944, 98.54748603, 99.55307263, 100.55865922, 101.56424581, 102.5698324, 103.57541899, 104.58100559, 105.58659218, 106.59217877, 107.59776536, 108.60335196, 109.60893855, 110.61452514, 111.62011173, 112.62569832, 113.63128492, 114.63687151, 115.6424581, 116.64804469, 117.65363128, 118.65921788, 119.66480447, 120.67039106, 121.67597765, 122.68156425, 123.68715084, 124.69273743, 125.69832402, 126.70391061, 127.70949721, 128.7150838, 129.72067039, 130.72625698, 131.73184358, 132.73743017, 133.74301676, 134.74860335, 135.75418994, 136.75977654, 137.76536313, 138.77094972, 139.77653631, 140.78212291, 141.7877095, 142.79329609, 143.79888268, 144.80446927, 145.81005587, 146.81564246, 147.82122905, 148.82681564, 149.83240223, 150.83798883, 151.84357542, 152.84916201, 153.8547486, 154.8603352, 155.86592179, 156.87150838, 157.87709497, 158.88268156, 159.88826816, 160.89385475, 161.89944134, 162.90502793, 163.91061453, 164.91620112, 165.92178771, 166.9273743, 167.93296089, 168.93854749, 169.94413408, 170.94972067, 171.95530726, 172.96089385, 173.96648045, 174.97206704, 175.97765363, 176.98324022, 177.98882682, 178.99441341, 180.0])*, *SD_spread=1*)

class pytranskit.TBM.TBM_RCDT.**batch_RCDT**(*thetas=array([0.0, 1.00558659, 2.01117318, 3.01675978, 4.02234637, 5.02793296, 6.03351955, 7.03910615, 8.04469274, 9.05027933, 10.05586592, 11.06145251, 12.06703911, 13.0726257, 14.07821229, 15.08379888, 16.08938547, 17.09497207, 18.10055866, 19.10614525, 20.11173184, 21.11731844, 22.12290503, 23.12849162, 24.13407821, 25.1396648, 26.1452514, 27.15083799, 28.15642458, 29.16201117, 30.16759777, 31.17318436, 32.17877095, 33.18435754, 34.18994413, 35.19553073, 36.20111732, 37.20670391, 38.2122905, 39.21787709, 40.22346369, 41.22905028, 42.23463687, 43.24022346, 44.24581006, 45.25139665, 46.25698324, 47.26256983, 48.26815642, 49.27374302, 50.27932961, 51.2849162, 52.29050279, 53.29608939, 54.30167598, 55.30726257, 56.31284916, 57.31843575, 58.32402235, 59.32960894, 60.33519553, 61.34078212, 62.34636872, 63.35195531, 64.3575419, 65.36312849, 66.36871508, 67.37430168, 68.37988827, 69.38547486, 70.39106145, 71.39664804, 72.40223464, 73.40782123, 74.41340782, 75.41899441, 76.42458101, 77.4301676, 78.43575419, 79.44134078, 80.44692737, 81.45251397, 82.45810056, 83.46368715, 84.46927374, 85.47486034, 86.48044693, 87.48603352, 88.49162011, 89.4972067, 90.5027933, 91.50837989, 92.51396648, 93.51955307, 94.52513966, 95.53072626, 96.53631285, 97.54189944, 98.54748603, 99.55307263, 100.55865922, 101.56424581, 102.5698324, 103.57541899, 104.58100559, 105.58659218, 106.59217877, 107.59776536, 108.60335196, 109.60893855, 110.61452514, 111.62011173, 112.62569832, 113.63128492, 114.63687151, 115.6424581, 116.64804469, 117.65363128, 118.65921788, 119.66480447, 120.67039106, 121.67597765, 122.68156425, 123.68715084, 124.69273743, 125.69832402, 126.70391061, 127.70949721, 128.7150838, 129.72067039, 130.72625698, 131.73184358, 132.73743017, 133.74301676, 134.74860335, 135.75418994, 136.75977654, 137.76536313, 138.77094972, 139.77653631, 140.78212291, 141.7877095, 142.79329609, 143.79888268, 144.80446927, 145.81005587, 146.81564246, 147.82122905, 148.82681564, 149.83240223, 150.83798883, 151.84357542, 152.84916201, 153.8547486, 154.8603352, 155.86592179, 156.87150838, 157.87709497, 158.88268156, 159.88826816, 160.89385475, 161.89944134, 162.90502793, 163.91061453, 164.91620112, 165.92178771, 166.9273743, 167.93296089, 168.93854749, 169.94413408, 170.94972067, 171.95530726, 172.96089385, 173.96648045, 174.97206704, 175.97765363, 176.98324022, 177.98882682, 178.99441341, 180.0])*, *rm_edge=False*)

Bases: object

forward(*X*, *template*)

forward_seq(*X*, *template*)

fun_ircdt_batch(*data*)

fun_ircdt_single(*Ihat*)

fun_rcdt_batch(*data*)

fun_rcdt_single(*I*)

inverse(*Xhat*, *template*)

ircdt_parallel(*Xhat*)

rcdt_parallel(*X*)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

A

B

C

D

E

F

G

I

R

S

T

V