mvlearn Documentation

Release alpha

Richard Guo, Ronan Perry, Gavin Mischler, Theo Lee, Alexander C

Aug 11, 2020

Contents

mvlearn is a Python module for multiview learning.

CHAPTER 1

Motivation

In many data sets, there are multiple measurement modalities of the same subject, i.e. multiple X matrices (views) for the same class label vector y. For example, a set of diseased and healthy patients in a neuroimaging study may undergo both CT and MRI scans. Traditional methods for inference and analysis are often poorly suited to account for multiple views of the same subject as they cannot account for complementing views that hold different statistical properties. While single-view methods are consolidated in well-documented packages such as scikit-learn, there is no equivalent for multiview methods. In this package, we provide a well-documented and tested collection of utilities and algorithms designed for the processing and analysis of multiview data sets.

CHAPTER 2

Python

Python is a powerful programming language that allows concise expressions of network algorithms. Python has a vibrant and growing ecosystem of packages that mvlearn uses to provide more features such as numerical linear algebra. In order to make the most out of mvlearn you will want to know how to write basic programs in Python. Among the many guides to Python, we recommend the Python documentation.

Currently, mvlearn is supported for Python 3.6, 3.7, and 3.8.

CHAPTER 3

Free software

mvlearn is free software; you can redistribute it and/or modify it under the terms of the *Apache-2.0*. We welcome contributions. Join us on GitHub.

CHAPTER 4

History

mvlearn was developed during the end of 2019 by Richard Guo, Ronan Perry, Gavin Mischler, Theo Lee, Alexander Chang, Arman Koul, and Cameron Franz, a team out of the Johns Hopkins University NeuroData group.

CHAPTER 5

Documentation

mvlearn is a Python package of multiview learning tools.

5.1 Install

`mvlearn` can be installed by using `pip`, GitHub, or through the conda-forge channel into an existing `conda` environment.

IMPORTANT NOTE: `mvlearn` has an optional dependency to `torch` and `tqdm`, so special instructions must be followed to include these optional dependencies in the installation (if you do not have those packages already) in order to access all the features within `mvlearn`. More details can be found in *Including optional torch dependencies for full functionality*.

5.1.1 Installing the released version with pip

Below we assume you have the default Python3 environment already configured on your computer and you intend to install `mvlearn` inside of it. If you want to create and work with Python virtual environments, please follow instructions on venv and virtual environments.

First, make sure you have the latest version of `pip3` (the Python3 package manager) installed. If you do not, refer to the Pip documentation and install `pip3` first.

Install the current release of `mvlearn` with `pip3`:

```
$ pip3 install mvlearn
```

To upgrade to a newer release use the `--upgrade` flag:

```
$ pip3 install --upgrade mvlearn
```

If you do not have permission to install software systemwide, you can install into your user directory using the `--user` flag:

```
$ pip3 install --user mvlearn
```

Alternatively, you can manually download `mvlearn` from GitHub or PyPI. To install one of these versions, unpack it and run the following from the top-level source directory using the Terminal:

```
$ pip3 install -e .
```

This will install `mvlearn` and the required dependencies (see below).

Including optional torch dependencies for full functionality

Due to the size of the `torch` dependency, it is an optional installation. Because it, and `tqdm`, are only used by Deep CCA and SplitAE, they are not included in the basic `mvlearn` download. If you wish to use functionality associated with these dependencies (Deep CCA and SplitAE), you must install additional dependencies. You can install them independently, or to install everything from PyPI, simply call:

```
$ pip3 install mvlearn[torch]
```

To upgrade the package and torch requirements:

```
$ pip3 install --upgrade mvlearn[torch]
```

If you have the package locally, from the top level folder call:

```
$ pip3 install -e .[torch]
```

5.1.2 Installing the released version with conda-forge

Here, we assume you have created a conda environment with one of the accepted python versions, and you intend to install the full `mvlearn` release into it (with torch dependencies included). For more information about using conda-forge feedstocks, see the about page, or the mvlearn feedstock.

To install `mvlearn` with conda, run:

```
$ conda install -c conda-forge mvlearn
```

To list all versions of `mvlearn` available on your platform, use:

```
$ conda search mvlearn --channel conda-forge
```

5.1.3 Python package dependencies

`mvlearn` requires the following packages:

- graspy >=0.1.1
- matplotlib >=3.0.0
- numpy >=1.17.0
- pandas >=0.25.0
- scikit-learn >=0.19.1
- scipy >=1.1.0

- seaborn >=0.9.0
- joblib >=0.11
- python-picard >= 0.4

with optional dependencies

- torch >=1.1.0
- tqdm

Currently, `mvlearn` is supported for Python 3.6, 3.7, and 3.8.

5.1.4 Hardware requirements

The `mvlearn` package requires only a standard computer with enough RAM to support the in-memory operations and free memory to install required packages.

5.1.5 OS Requirements

This package is supported for *Linux* and *macOS* and can also be run on Windows machines.

5.1.6 Testing

`mvlearn` uses the Python `pytest` testing package. If you don't already have that package installed, follow the directions on the pytest homepage.

5.2 Tutorials

5.2.1 Clustering

The following tutorials demonstrate the effectiveness of clustering algorithms designed specifically for multiview datasets.

Multi-view KMeans

[15]:
```
from mvlearn.datasets import load_UCImultifeature
from mvlearn.cluster import MultiviewKMeans
from sklearn.cluster import KMeans
import numpy as np
from sklearn.manifold import TSNE
from sklearn.metrics import normalized_mutual_info_score as nmi_score
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
RANDOM_SEED=5
```

Load in UCI digits multiple feature data set as an example

[16]:
```
# Load dataset along with labels for digits 0 through 4
n_class = 5
data, labels = load_UCImultifeature(select_labeled = list(range(n_class)))

# Just get the first two views of data
m_data = data[:2]
```

[17]:
```
# Helper function to display data and the results of clustering
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(data[0][:, 0], data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(data[1][:, 0], data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Single-view and multi-view clustering of the data with 2 views

Here we will compare the performance of the Multi-view and Single-view versions of kmeans clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

As we can see, Multi-view clustering produces clusters with higher purity compared to those produced by clustering on just a single view or by clustering the two views concatenated together.

[18]:
```
##################Single-view kmeans clustering#####################
# Cluster each view separately
s_kmeans = KMeans(n_clusters=n_class, random_state=RANDOM_SEED)
s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

# Concatenate the multiple views into a single view
s_data = np.hstack(m_data)
s_clusters = s_kmeans.fit_predict(s_data)

# Compute nmi between true class labels and single-view cluster labels
s_nmi_v1 = nmi_score(labels, s_clusters_v1)
s_nmi_v2 = nmi_score(labels, s_clusters_v2)
s_nmi = nmi_score(labels, s_clusters)
print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))
```

(continues on next page)

(continued from previous page)

```
#################Multi-view kmeans clustering#####################

# Use the MultiviewKMeans instance to cluster the data
m_kmeans = MultiviewKMeans(n_clusters=n_class, random_state=RANDOM_SEED)
m_clusters = m_kmeans.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi = nmi_score(labels, m_clusters)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))
```

```
Single-view View 1 NMI Score: 0.635

Single-view View 2 NMI Score: 0.746

Single-view Concatenated NMI Score: 0.746

Multi-view NMI Score: 0.770
```

Plot clusters produced by multi-view spectral clustering and the true clusters

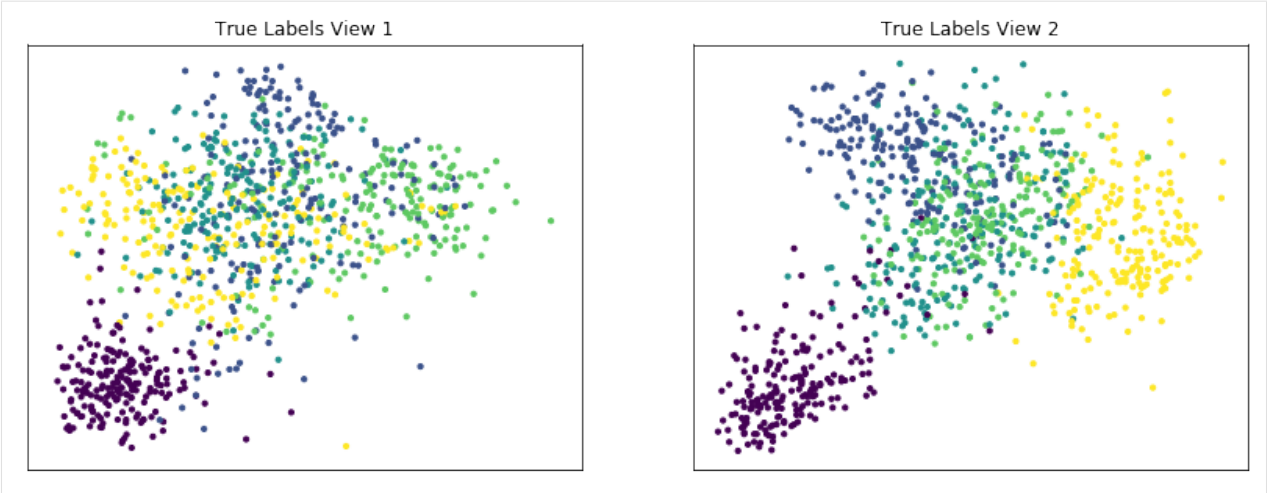
We will display the clustering results of the Multi-view kmeans clustering algorithm below, along with the true class labels.

[19]:
```
# Running TSNE to display clustering results via low dimensional embedding
tsne = TSNE()
new_data_1 = tsne.fit_transform(m_data[0])
new_data_2 = tsne.fit_transform(m_data[1])
```

[20]:
```
display_plots('Multi-view KMeans Clusters', m_data, m_clusters)
display_plots('True Labels', m_data, labels)
```

```
<Figure size 432x288 with 0 Axes>
```


```
<Figure size 432x288 with 0 Axes>
```



Spectral clustering with different parameters

Here we will again compare the performance of the Multi-view and Single-view versions of kmeans clusteringon data with 2 views. We will follow a similar procedure as before, but we will be using a different configuration of parameters for Multi-view Spectral Clustering.

Again, we can see that Multi-view clustering produces clusters with higher purity compared to those produced by clustering on just a single view or by clustering the two views concatenated together.

[21]:
```
#################Single-view kmeans clustering#####################
# Cluster each view separately
s_kmeans = KMeans(n_clusters=n_class, random_state=RANDOM_SEED)
s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

# Concatenate the multiple views into a single view
s_data = np.hstack(m_data)
s_clusters = s_kmeans.fit_predict(s_data)

# Compute nmi between true class labels and single-view cluster labels
s_nmi_v1 = nmi_score(labels, s_clusters_v1)
s_nmi_v2 = nmi_score(labels, s_clusters_v2)
s_nmi = nmi_score(labels, s_clusters)
print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

#################Multi-view kmeans clustering#####################

# Use the MultiviewKMeans instance to cluster the data
m_kmeans = MultiviewKMeans(n_clusters=n_class,
        n_init=10, max_iter=6, patience=2, random_state=RANDOM_SEED)
m_clusters = m_kmeans.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi = nmi_score(labels, m_clusters)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))
```

```
Single-view View 1 NMI Score: 0.635

Single-view View 2 NMI Score: 0.746

Single-view Concatenated NMI Score: 0.746

Multi-view NMI Score: 0.747
```

Assessing the Conditional Independence Views Requirement of Multi-view KMeans

In the following experiments, we will perform single-view kmeans clustering on the two views separately and on them concatenated together. We also perform multi-view clustering using the multi-view algorithm. We will also compare the performance of multi-view and single-view versions of kmeans clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

[8]:
```
import numpy as np
from numpy.random import multivariate_normal
import scipy as scp
from mvlearn.cluster.mv_k_means import MultiviewKMeans
from sklearn.metrics import normalized_mutual_info_score as nmi_score
from sklearn.cluster import KMeans
from sklearn.datasets import fetch_covtype
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.manifold import TSNE
import warnings
warnings.filterwarnings("ignore")
RANDOM_SEED=10
```

Artificial dataset with conditionally independent views

Here, we create an artificial dataset where the conditional independence assumption between views, given the true labels, is enforced. Our artificial dataset is derived from the forest covertypes dataset from the scikit-learn package. This dataset is comprised of 7 different classes, with with 54 different numerical features per sample. To create our artificial data, we will select 500 samples from each of the first 6 classes in the dataset, and from these, construct 3 artificial classes with 2 views each.

[2]:
```
def get_ci_data(num_samples=500):

    #Load in the vectorized news group data from scikit-learn package
    cov = fetch_covtype()
    all_data = np.array(cov.data)
    all_targets = np.array(cov.target)

    #Set class pairings as described in the multiview clustering paper
    view1_classes = [1, 2, 3]
    view2_classes = [4, 5, 6]

    #Create lists to hold data and labels for each of the classes across 2 different␣
↪views
    labels =  [num for num in range(len(view1_classes)) for _ in range(num_samples)]
    labels = np.array(labels)
```

(continues on next page)

(continued from previous page)

```
    view1_data = list()
    view2_data = list()

    #Randomly sample items from each of the selected classes in view1
    for class_num in view1_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view1_data.append(class_data[indices])
    view1_data = np.concatenate(view1_data)


    #Randomly sample items from each of the selected classes in view2
    for class_num in view2_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view2_data.append(class_data[indices])
    view2_data = np.concatenate(view2_data)

    #Shuffle and normalize vectors
    shuffled_inds = np.random.permutation(num_samples * len(view1_classes))
    view1_data = np.vstack(view1_data)
    view2_data = np.vstack(view2_data)
    view1_data = view1_data[shuffled_inds]
    view2_data = view2_data[shuffled_inds]
    magnitudes1 = np.linalg.norm(view1_data, axis=0)
    magnitudes2 = np.linalg.norm(view2_data, axis=0)
    magnitudes1[magnitudes1 == 0] = 1
    magnitudes2[magnitudes2 == 0] = 1
    magnitudes1 = magnitudes1.reshape((1, -1))
    magnitudes2 = magnitudes2.reshape((1, -1))
    view1_data /= magnitudes1
    view2_data /= magnitudes2
    labels = labels[shuffled_inds]
    return [view1_data, view2_data], labels
```

[3]:
```
def perform_clustering(seed, m_data, labels, n_clusters):
    ##################Single-view kmeans clustering#####################
    # Cluster each view separately
    s_kmeans = KMeans(n_clusters=n_clusters, random_state=seed, n_init=100)
    s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
    s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

    # Concatenate the multiple views into a single view
    s_data = np.hstack(m_data)
    s_clusters = s_kmeans.fit_predict(s_data)

    # Compute nmi between true class labels and single-view cluster labels
    s_nmi_v1 = nmi_score(labels, s_clusters_v1)
    s_nmi_v2 = nmi_score(labels, s_clusters_v2)
    s_nmi = nmi_score(labels, s_clusters)
    print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
    print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
    print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

    ##################Multi-view kmeans clustering#####################
```

(continues on next page)

(continued from previous page)

```
    # Use the MultiviewKMeans instance to cluster the data
    m_kmeans = MultiviewKMeans(n_clusters=n_clusters, n_init=100, random_state=seed)
    m_clusters = m_kmeans.fit_predict(m_data)

    # Compute nmi between true class labels and multi-view cluster labels
    m_nmi = nmi_score(labels, m_clusters)
    print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))

    return m_clusters
```

[4]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(new_data[0][:, 0], new_data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(new_data[1][:, 0], new_data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Comparing the performance of multi-view and single-view KMeans on our dataset with conditionally independent views

The co-Expectation Maximization framework (and co-training), relies on the fundamental assumption that data views are conditionally independent. If all views are informative and conditionally independent, then Multi-view KMeans is expected to produce higher quality clusters than Single-view KMeans, for either view or for both views concatenated together. Here, we will evaluate the quality of clusters by using the normalized mutual information metric, which is essentially a measure of the purity of clusters with respect to the true underlying class labels.

As we see below, Multi-view KMeans produces clusters with higher purity than Single-view KMeans across a range of values for the n_clusters parameter for data with complex and informative views, which is consistent with some of the results from the original Multi-view clustering paper.

[9]:
```
data, labels = get_ci_data()
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 3)

# Running TSNE to display clustering results via low dimensional embedding
tsne = TSNE()
new_data = list()
new_data.append(tsne.fit_transform(data[0]))
new_data.append(tsne.fit_transform(data[1]))
display_plots('True Labels', new_data, labels)
display_plots('Multi-view Clustering Results', new_data, m_clusters)
```

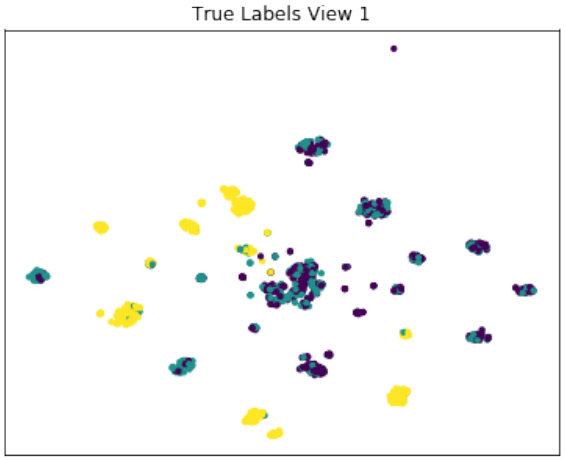
```
Single-view View 1 NMI Score: 0.342
```

(continues on next page)

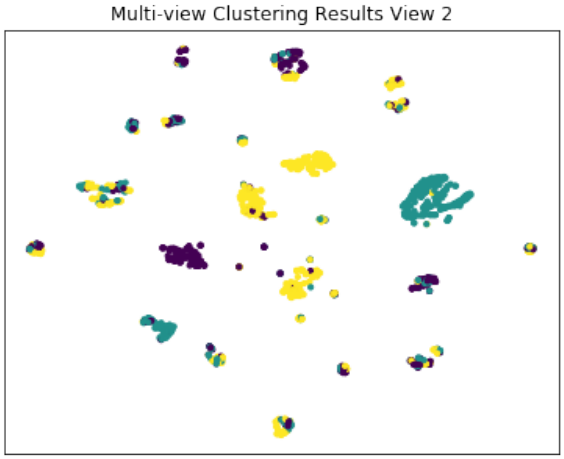
(continued from previous page)

```
Single-view View 2 NMI Score: 0.503

Single-view Concatenated NMI Score: 0.422

Multi-view NMI Score: 0.530
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Artificial dataset with conditionally dependent views

Here, we create an artificial dataset where the conditional independence assumption between views, given the true labels, is violated. We again derive our dataset from the forest covertypes dataset from sklearn. However, this time, we use only the first 3 classes of the dataset, which will correspond to the 3 clusters for view 1. To produce view 2, we will apply a simple nonlinear transformation to view 1 using the logistic function, and we will apply a negligible amount of noise to the second view to avoid convergence issues. This will result in a dataset where the correspondance between views is very high.

[6]:
```
def get_cd_data(num_samples=500):
```

(continues on next page)

(continued from previous page)

```
    #Load in the vectorized news group data from scikit-learn package
    cov = fetch_covtype()
    all_data = np.array(cov.data)
    all_targets = np.array(cov.target)

    #Set class pairings as described in the multiview clustering paper
    view1_classes = [1, 2, 3]
    view2_classes = [4, 5, 6]

    #Create lists to hold data and labels for each of the classes across 2 different␣
↪views
    labels =  [num for num in range(len(view1_classes)) for _ in range(num_samples)]
    labels = np.array(labels)
    view1_data = list()
    view2_data = list()

    #Randomly sample 500 items from each of the selected classes in view1
    for class_num in view1_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view1_data.append(class_data[indices])
    view1_data = np.concatenate(view1_data)


    #Construct view 2 by applying a nonlinear transformation
    #to data from view 1 comprised of a linear transformation
    #and a logistic nonlinearity
    t_mat =  np.random.random((view1_data.shape[1], 50))
    noise = 0.005 - 0.01*np.random.random((view1_data.shape[1], 50))
    t_mat *= noise
    transformed = view1_data @ t_mat
    view2_data = scp.special.expit(transformed)

    #Shuffle and normalize vectors
    shuffled_inds = np.random.permutation(num_samples * len(view1_classes))
    view1_data = np.vstack(view1_data)
    view2_data = np.vstack(view2_data)
    view1_data = view1_data[shuffled_inds]
    view2_data = view2_data[shuffled_inds]
    magnitudes1 = np.linalg.norm(view1_data, axis=0)
    magnitudes2 = np.linalg.norm(view2_data, axis=0)
    magnitudes1[magnitudes1 == 0] = 1
    magnitudes2[magnitudes2 == 0] = 1
    magnitudes1 = magnitudes1.reshape((1, -1))
    magnitudes2 = magnitudes2.reshape((1, -1))
    view1_data /= magnitudes1
    view2_data /= magnitudes2
    labels = labels[shuffled_inds]
    return [view1_data, view2_data], labels
```

Comparing the performance of multi-view and single-view KMeans on our dataset with conditionally dependent views

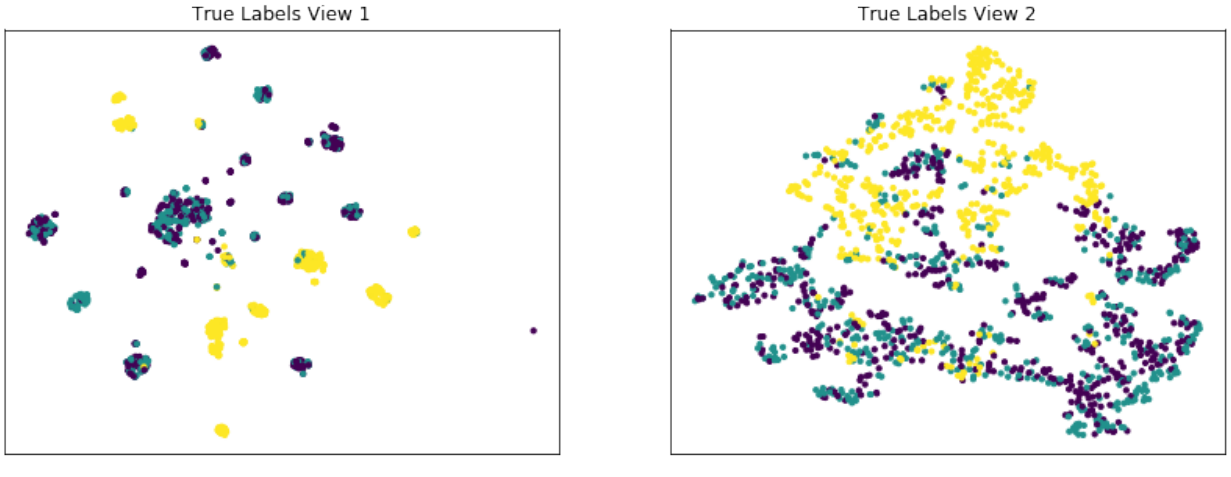
As mentioned before co-Expectation Maximization framework (and co-training), relies on the fundamental assumption that data views are conditionally independent. Here, we will again compare the performance of single-view and multi-view kmeans clustering using the same methods as before, but on our conditionally dependent dataset.

As we see below, Multi-view KMeans does not beat the best Single-view clustering performance with respect to purity, since that the views are conditionally dependent.

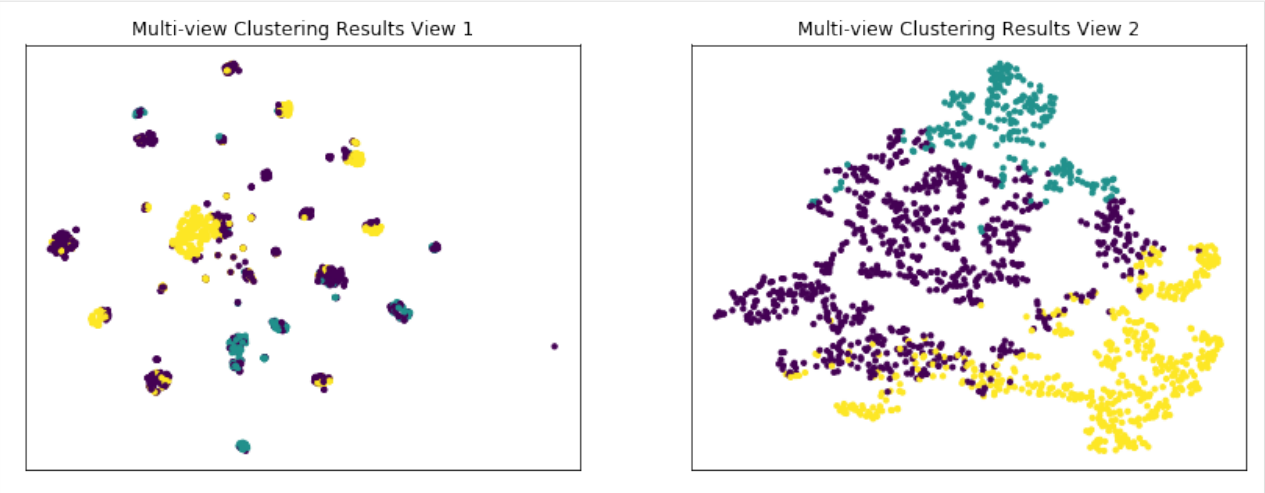
[10]:
```
data, labels = get_cd_data()
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 3)

# Running TSNE to display clustering results via low dimensional embedding
tsne = TSNE()
new_data = list()
new_data.append(tsne.fit_transform(data[0]))
new_data.append(tsne.fit_transform(data[1]))
display_plots('True Labels', new_data, labels)
display_plots('Multi-view Clustering Results', new_data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.342

Single-view View 2 NMI Score: 0.184

Single-view Concatenated NMI Score: 0.222

Multi-view NMI Score: 0.236
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Evaluating the performance of Multi-view and Single-view KMeans clustering on other complex data

To see the relative performance of single-view and multi-view clustering on complex, real world data, please refer to the MultiviewKMeans_Tutorial notebook, which illustrates the application of both of these clustering methods to the UCI Digits Multiple Features Dataset. In this notebook, we can see that multi-view kmeans clustering produces clusters with higher purity than the single-view analogs when given informative views of data, even if conditional independence is not strictly enforced.

Multi-view vs. Single-view KMeans

[1]:
```
import numpy as np
from numpy.random import multivariate_normal
from mvlearn.cluster.mv_k_means import MultiviewKMeans
from sklearn.cluster import KMeans
from sklearn.metrics import normalized_mutual_info_score as nmi_score
import matplotlib.pyplot as plt
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
RANDOM_SEED=10
```

A function to generate 2 views of data for 2 classes

This function takes parameters for means, variances, and number of samples for class and generates data based on those parameters. The underlying probability distribution of the data is a multivariate gaussian distribution.

[2]:
```
def create_data(seed, vmeans, vvars, num_per_class=500):

    np.random.seed(seed)
    data = [[],[]]

    for view in range(2):
        for comp in range(len(vmeans[0])):
            cov = np.eye(2) * vvars[view][comp]
```

(continues on next page)

(continued from previous page)

```
            comp_samples = np.random.multivariate_normal(vmeans[view][comp], cov,␣
↪size=num_per_class)
            data[view].append(comp_samples)
    for view in range(2):
        data[view] = np.vstack(data[view])

    labels = list()
    for ind in range(len(vmeans[0])):
        labels.append(ind * np.ones(num_per_class,))

    labels = np.concatenate(labels)

    return data, labels
```

Creating a function to display data and the results of clustering

The following function plots both views of data given a dataset and corresponding labels.

[3]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(data[0][:, 0], data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(data[1][:, 0], data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Creating a function to perform both single-view and multi-view kmeans clustering

In the following function, we will perform single-view kmeans clustering on the two views separately and on them concatenated together. We also perform multi-view clustering using the multi-view algorithm. We will also compare the performance of multi-view and single-view versions of kmeans clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

[4]:
```
def perform_clustering(seed, m_data, labels, n_clusters):
    ##################Single-view kmeans clustering####################
    # Cluster each view separately
    s_kmeans = KMeans(n_clusters=n_clusters, random_state=seed, n_init=100)
    s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
    s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

    # Concatenate the multiple views into a single view
    s_data = np.hstack(m_data)
    s_clusters = s_kmeans.fit_predict(s_data)
```

(continues on next page)

(continued from previous page)

```
    # Compute nmi between true class labels and single-view cluster labels
    s_nmi_v1 = nmi_score(labels, s_clusters_v1)
    s_nmi_v2 = nmi_score(labels, s_clusters_v2)
    s_nmi = nmi_score(labels, s_clusters)
    print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
    print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
    print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

    ################Multi-view kmeans clustering#####################

    # Use the MultiviewKMeans instance to cluster the data
    m_kmeans = MultiviewKMeans(n_clusters=n_clusters, n_init=100, random_state=seed)
    m_clusters = m_kmeans.fit_predict(m_data)

    # Compute nmi between true class labels and multi-view cluster labels
    m_nmi = nmi_score(labels, m_clusters)
    print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))

    return m_clusters
```

General experimentation procedures

For each of the experiments below, we run both single-view kmeans clustering and multi-view kmeans clustering. For evaluating single-view performance, we run the algorithm on each view separately as well as all views concatenated together. We evalaute performance using normalized mutual information, which is a measure of cluster purity with respect to the true labels. For both algorithms, we use an n_init value of 100, which means that we run each algorithm across 100 random cluster initializations and select the best clustering results with respect to cluster inertia (within cluster sum-of-squared distances).

Performance when cluster components in both views are well separated

Cluster components 1: * Mean: [3, 3] (both views) * Covariance = I (both views)

Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

As we can see, multi-view kmeans clustering performs about as well as single-view kmeans clustering for the concatenated views, and both of these perform better than on single-view clustering for just one view.

[5]:
```
v1_means = [[3, 3], [0, 0]]
v2_means = [[3, 3], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

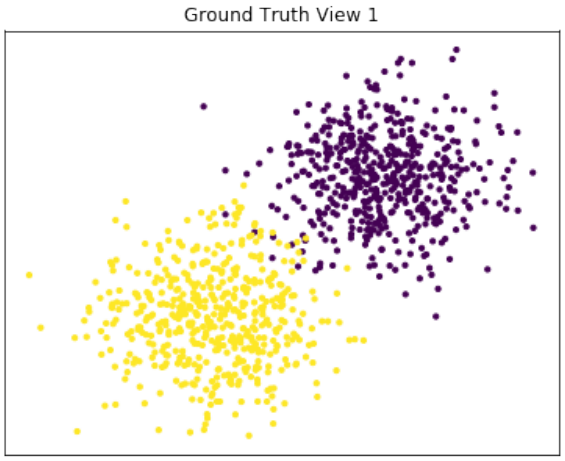
```
Single-view View 1 NMI Score: 0.901

Single-view View 2 NMI Score: 0.888
```

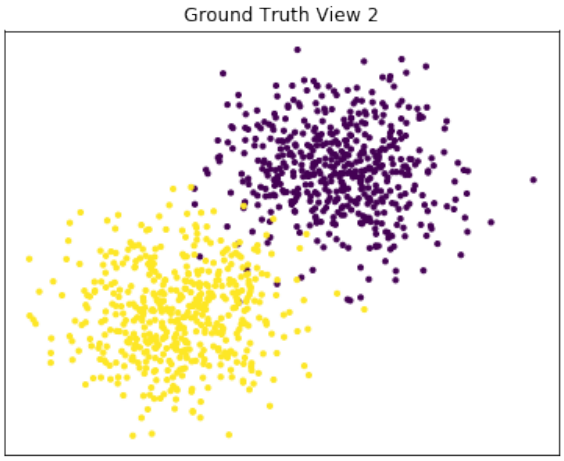
(continues on next page)

(continued from previous page)

```
Single-view Concatenated NMI Score: 0.990

Multi-view NMI Score: 0.990
```

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are relatively inseparable (highly overlapping) in both views

Cluster components 1: * Mean: [0.5, 0.5] (both views) * Covariance = I (both views)

Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

As we can see, multi-view kmeans clustering performs about as poorly as single-view kmeans clustering across both individual views and concatenated views as inputs.

[6]:
```
v1_means = [[0.5, 0.5], [0, 0]]
v2_means = [[0.5, 0.5], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
```

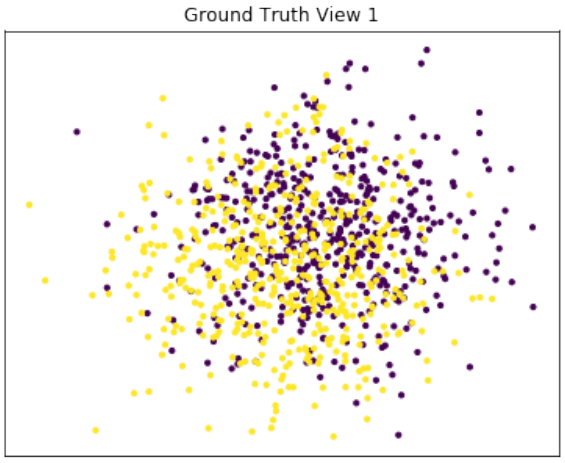
(continues on next page)

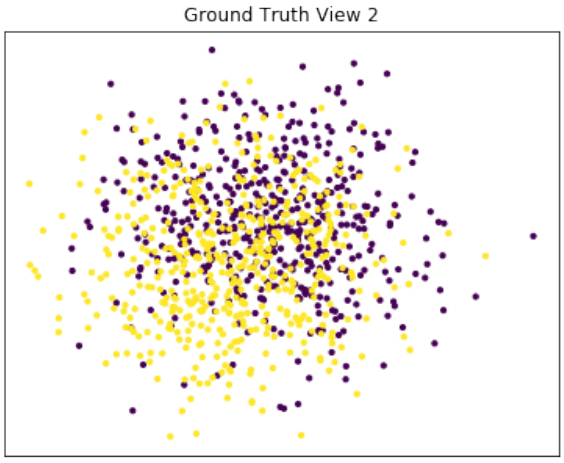
(continued from previous page)

```
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

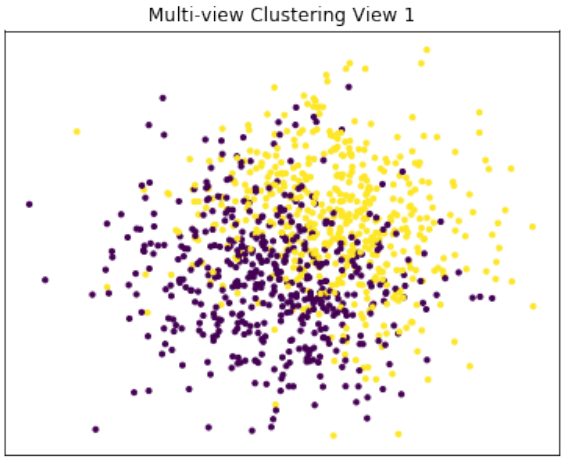
```
Single-view View 1 NMI Score: 0.062

Single-view View 2 NMI Score: 0.044

Single-view Concatenated NMI Score: 0.098

Multi-view NMI Score: 0.110
```

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```



Performance when cluster components are somewhat separable (somewhat overlapping) in both views

Cluster components 1: * Mean: [1.5, 1.5] (both views) * Covariance = I (both views)

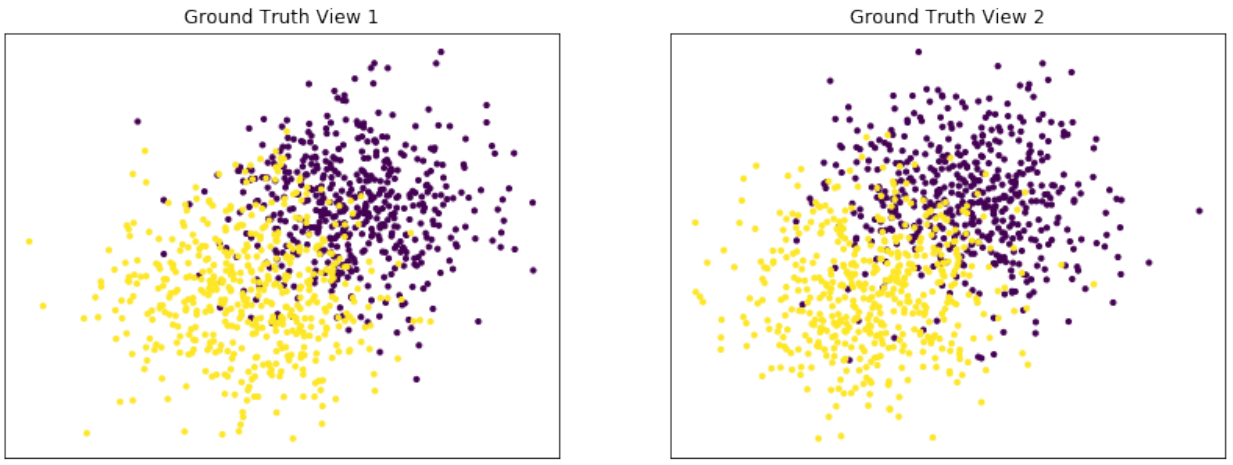
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

Again we can see that multi-view kmeans clustering performs about as well as single-view kmeans clustering for the concatenated views, and both of these perform better than on single-view clustering for just one view.

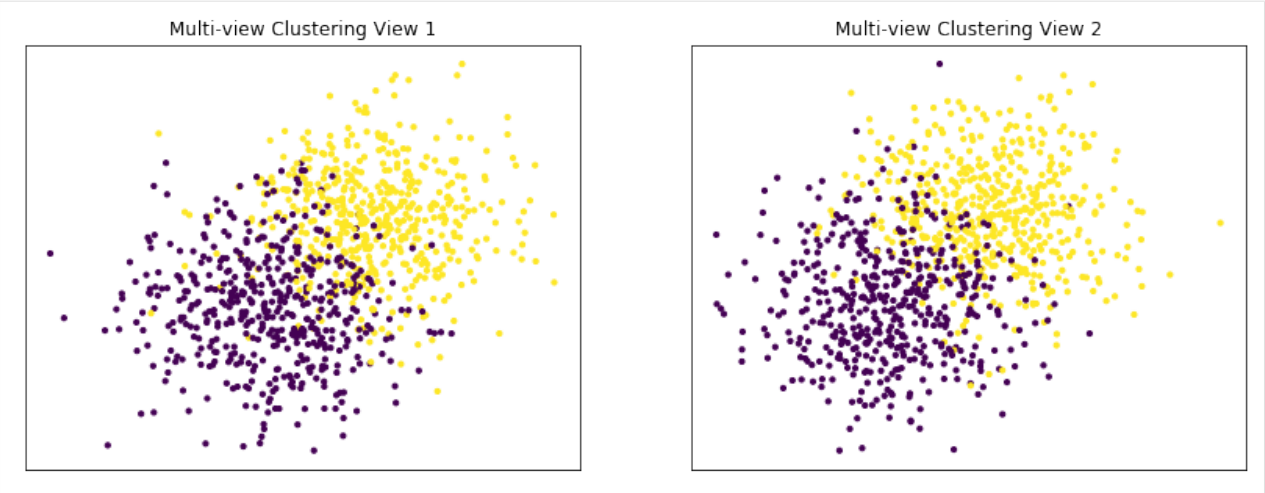
[7]:
```
v1_means = [[1.5, 1.5], [0, 0]]
v2_means = [[1.5, 1.5], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.425

Single-view View 2 NMI Score: 0.410

Single-view Concatenated NMI Score: 0.657

Multi-view NMI Score: 0.632
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Performance when cluster components are highly overlapping in one view

Cluster components 1: * Mean: View 1 = [0.5, 0.5], View 2 = [2, 2] * Covariance = I (both views)

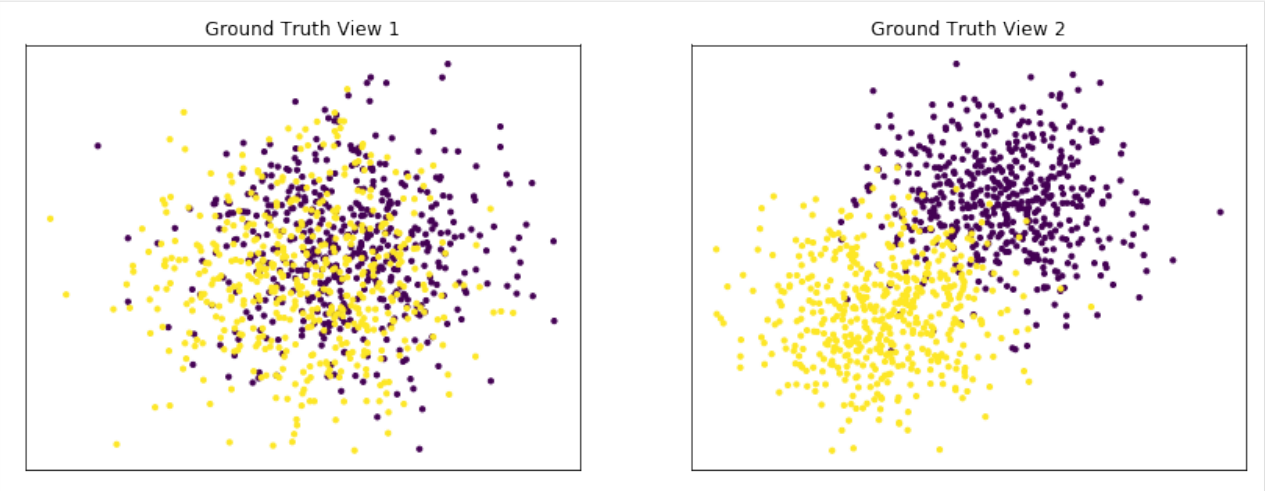
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

As we can see, multi-view kmeans clustering performs worse than single-view kmeans clustering with concatenated views as inputs and with the best view as the input.

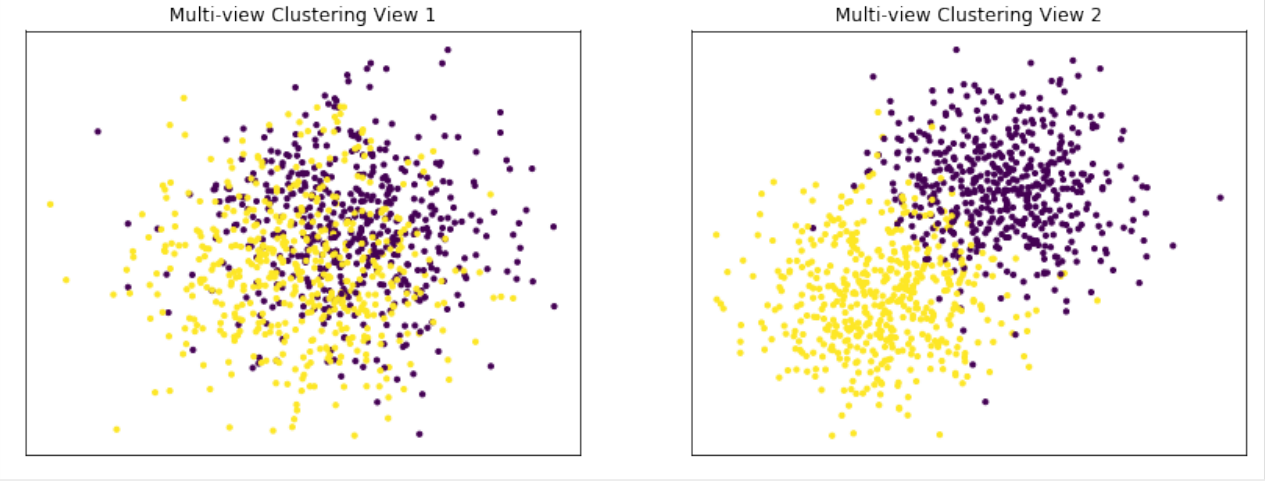
[8]:
```
v1_means = [[0.5, 0.5], [0, 0]]
v2_means = [[2, 2], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.062

Single-view View 2 NMI Score: 0.608

Single-view Concatenated NMI Score: 0.616

Multi-view NMI Score: 0.591
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Conclusions

Here, we have seen some of the limitations of multi-view kmeans clustering. From the experiments above, it is apparent that multi-view kmeans clustering performs equally as well or worse than single-view kmeans clustering on concatenated data when views are informative but the data is fairly simple (i.e. only has 2 features per view). However, it is clear that the multi-view kmeans algorithm does perform better on well separated cluster components than it does on highly overlapping cluster components, which does validate it's basic functionality as a clustering algorithm.

Multi-view Spectral Clustering

[1]:
```
from mvlearn.datasets import load_UCImultifeature
from mvlearn.cluster import MultiviewSpectralClustering
from mvlearn.plotting import quick_visualize
import numpy as np
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score as nmi_score
from sklearn.datasets import make_moons
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import scipy
import warnings

warnings.simplefilter('ignore') # Ignore warnings
%matplotlib inline
RANDOM_SEED=10
```

Creating a function to display data and the results of clustering

The following function plots both views of data given a dataset and corresponding labels.

[2]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(data[0][:, 0], data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(data[1][:, 0], data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Performance on moons dataset

For this example, we use the sklearn make_moons function to make two interleaving half circles in two views. We then use spectral clustering to separate the two views. As we can see below, multi-view spectral clustering is capable of effectively clustering non-convex cluster shapes, similarly to its single-view analog.

[3]:
```
# A function to generate the moons data
def create_moons(seed, num_per_class=500):

    np.random.seed(seed)
    data = []
    labels = []

    for view in range(2):
        v_dat, v_labs = make_moons(num_per_class*2,
                random_state=seed + view, noise=0.05, shuffle=False)
        if view == 1:
            v_dat = v_dat[:, ::-1]

        data.append(v_dat)
    for ind in range(len(data)):
        labels.append(ind * np.ones(num_per_class,))
    labels = np.concatenate(labels)
```

(continues on next page)

(continued from previous page)

```
    return data, labels
```

[4]:
```
# Generating the data
m_data, labels = create_moons(RANDOM_SEED)
n_class = 2

#################Single-view spectral clustering####################
# Cluster each view separately
s_spectral = SpectralClustering(n_clusters=n_class,
            affinity='nearest_neighbors', random_state=RANDOM_SEED, n_init=100)
s_clusters_v1 = s_spectral.fit_predict(m_data[0])
s_clusters_v2 = s_spectral.fit_predict(m_data[1])

# Concatenate the multiple views into a single view
s_data = np.hstack(m_data)
s_clusters = s_spectral.fit_predict(s_data)

# Compute nmi between true class labels and single-view cluster labels
s_nmi_v1 = nmi_score(labels, s_clusters_v1)
s_nmi_v2 = nmi_score(labels, s_clusters_v2)
s_nmi = nmi_score(labels, s_clusters)
print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

################Multi-view spectral clustering####################

# Use the MultiviewSpectralClustering instance to cluster the data
m_spectral = MultiviewSpectralClustering(n_clusters=n_class,
                affinity='nearest_neighbors', max_iter=12, random_state=RANDOM_SEED,␣
↪n_init=100)
m_clusters = m_spectral.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi = nmi_score(labels, m_clusters)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))
```

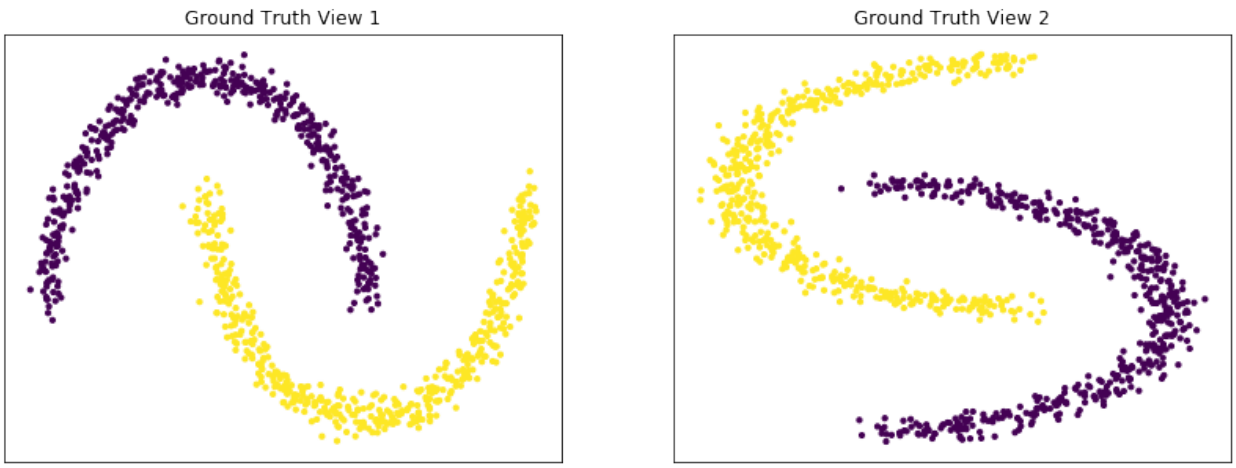
```
Single-view View 1 NMI Score: 1.000

Single-view View 2 NMI Score: 1.000

Single-view Concatenated NMI Score: 1.000

Multi-view NMI Score: 1.000
```

Plots of clusters produced by multi-view spectral clustering and the true clusters

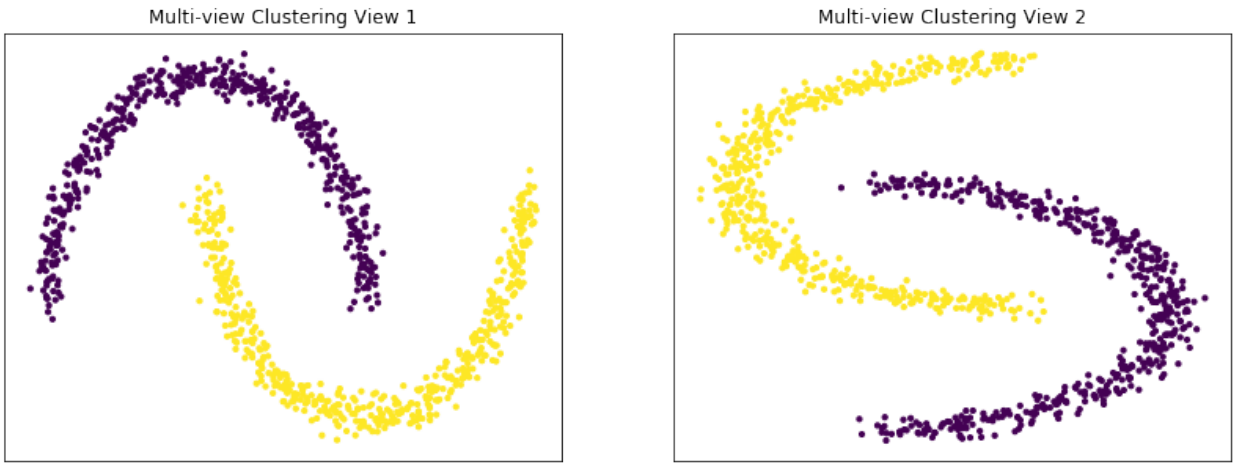
We will display the clustering results of the Multi-view spectral clustering algorithm below, along with the true class labels.

[5]:
```
display_plots('Ground Truth' , m_data, labels)
display_plots('Multi-view Clustering' , m_data, m_clusters)
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Performance on the UCI Digits Multiple Features data set with 2 views

Here we will compare the performance of the Multi-view and Single-view versions of spectral clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

As we can see, Multi-view clustering produces clusters with higher purity compared to those produced by Single-view clustering for all 3 input types.

[6]:
```
# Load dataset along with labels for digits 0 through 4
n_class = 5
m_data, labels = load_UCImultifeature(select_labeled = list(range(n_class)))
```

[7]:
```
##################Single-view spectral clustering#####################
# Cluster each view separately
s_spectral = SpectralClustering(n_clusters=n_class, random_state=RANDOM_SEED, n_
↪init=100)
```

(continues on next page)

(continued from previous page)

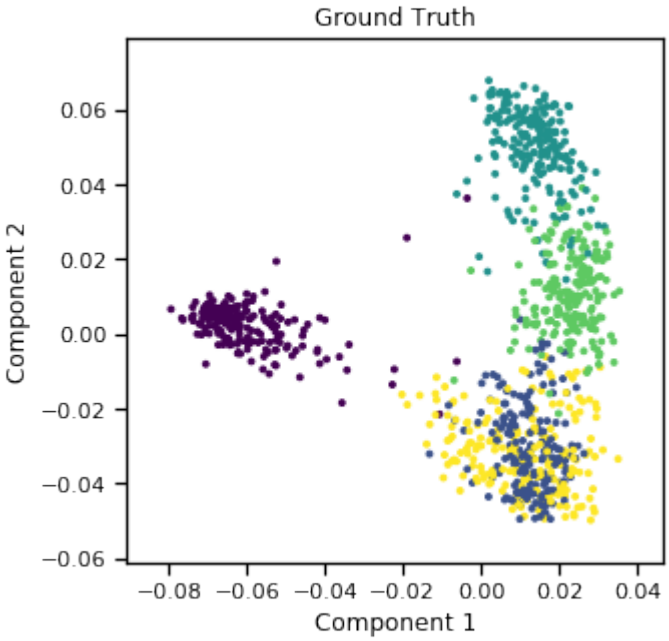
```
for i in range(len(m_data)):
    s_clusters = s_spectral.fit_predict(m_data[i])
    s_nmi = nmi_score(labels, s_clusters, average_method='arithmetic')
    print('Single-view View {0:d} NMI Score: {1:.3f}\n'.format(i + 1, s_nmi))

# Concatenate the multiple views into a single view and produce clusters
s_data = np.hstack(m_data)
s_clusters = s_spectral.fit_predict(s_data)

s_nmi = nmi_score(labels, s_clusters)
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

#################Multi-view spectral clustering####################

# Use the MultiviewSpectralClustering instance to cluster the data
m_spectral1 = MultiviewSpectralClustering(n_clusters=n_class,
            random_state=RANDOM_SEED, n_init=100)
m_clusters1 = m_spectral1.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi1 = nmi_score(labels, m_clusters1)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi1))
```

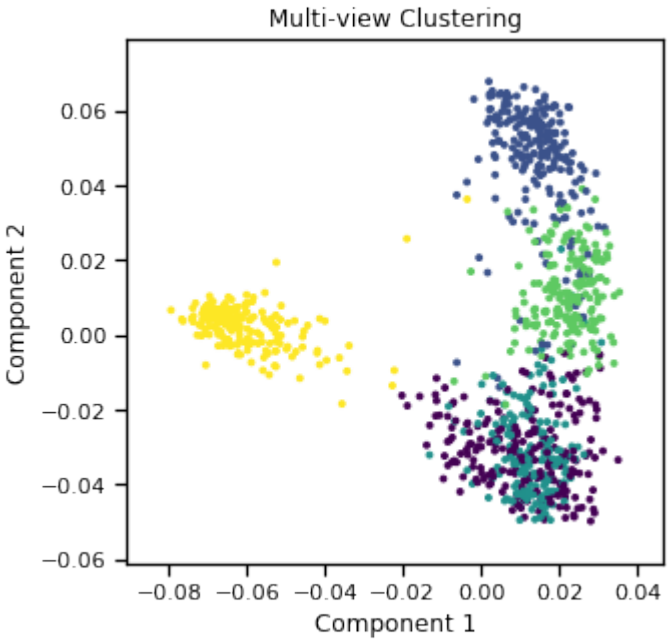
```
Single-view View 1 NMI Score: 0.620

Single-view View 2 NMI Score: 0.007

Single-view View 3 NMI Score: 0.004

Single-view View 4 NMI Score: -0.000

Single-view View 5 NMI Score: 0.007

Single-view View 6 NMI Score: 0.010

Single-view Concatenated NMI Score: 0.008

Multi-view NMI Score: 0.881
```

Plots of clusters produced by multi-view spectral clustering and the true clusters

We will display the clustering results of the Multi-view spectral clustering algorithm below, along with the true class labels.

[8]:
```
quick_visualize(m_data, labels=labels, title='Ground Truth', scatter_kwargs={'s':8})
quick_visualize(m_data, labels=m_clusters1, title='Multi-view Clustering', scatter_
↪kwargs={'s':8})
```





Assessing the Conditional Independence Views Requirement of Multi-view Spectral Clustering

[2]:
```
import numpy as np
from numpy.random import multivariate_normal
import scipy as scp
from mvlearn.cluster.mv_spectral import MultiviewSpectralClustering
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score as nmi_score
from sklearn.datasets import fetch_covtype
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.manifold import TSNE
import warnings
warnings.filterwarnings("ignore")
RANDOM_SEED=10
```

Creating an artificial dataset where the conditional independence assumption between views holds

Here, we create an artificial dataset where the conditional independence assumption between views, given the true labels, is enforced. Our artificial dataset is derived from the forest covertypes dataset from the scikit-learn package. This dataset is comprised of 7 different classes, with with 54 different numerical features per sample. To create our artificial data, we will select 500 samples from each of the first 6 classes in the dataset, and from these, construct 3 artificial classes with 2 views each.

[3]:
```
def get_ci_data(num_samples=500):

    #Load in the vectorized news group data from scikit-learn package
    cov = fetch_covtype()
    all_data = np.array(cov.data)
    all_targets = np.array(cov.target)

    #Set class pairings as described in the multiview clustering paper
    view1_classes = [1, 2, 3]
    view2_classes = [4, 5, 6]

    #Create lists to hold data and labels for each of the classes across 2 different␣
↪views
    labels =  [num for num in range(len(view1_classes)) for _ in range(num_samples)]
    labels = np.array(labels)
    view1_data = list()
    view2_data = list()

    #Randomly sample items from each of the selected classes in view1
    for class_num in view1_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view1_data.append(class_data[indices])
    view1_data = np.concatenate(view1_data)


    #Randomly sample items from each of the selected classes in view2
    for class_num in view2_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view2_data.append(class_data[indices])
    view2_data = np.concatenate(view2_data)

    #Shuffle and normalize vectors
    shuffled_inds = np.random.permutation(num_samples * len(view1_classes))
    view1_data = np.vstack(view1_data)
    view2_data = np.vstack(view2_data)
    view1_data = view1_data[shuffled_inds]
    view2_data = view2_data[shuffled_inds]
```

(continues on next page)

(continued from previous page)

```
    magnitudes1 = np.linalg.norm(view1_data, axis=0)
    magnitudes2 = np.linalg.norm(view2_data, axis=0)
    magnitudes1[magnitudes1 == 0] = 1
    magnitudes2[magnitudes2 == 0] = 1
    magnitudes1 = magnitudes1.reshape((1, -1))
    magnitudes2 = magnitudes2.reshape((1, -1))
    view1_data /= magnitudes1
    view2_data /= magnitudes2
    labels = labels[shuffled_inds]
    return [view1_data, view2_data], labels
```

Creating a function to perform both single-view and multi-view spectral clustering

In the following function, we will perform single-view spectral clustering on the two views separately and on them concatenated together. We also perform multi-view clustering using the multi-view algorithm. We will also compare the performance of multi-view and single-view versions of spectral clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

[4]:
```
def perform_clustering(seed, m_data, labels, n_clusters):

    ##################Single-view spectral clustering#####################
    # Cluster each view separately
    s_spectral = SpectralClustering(n_clusters=n_clusters, random_state=RANDOM_SEED,␣
↪n_init=100)
    s_clusters_v1 = s_spectral.fit_predict(m_data[0])
    s_clusters_v2 = s_spectral.fit_predict(m_data[1])

    # Concatenate the multiple views into a single view
    s_data = np.hstack(m_data)
    s_clusters = s_spectral.fit_predict(s_data)

    # Compute nmi between true class labels and single-view cluster labels
    s_nmi_v1 = nmi_score(labels, s_clusters_v1)
    s_nmi_v2 = nmi_score(labels, s_clusters_v2)
    s_nmi = nmi_score(labels, s_clusters)
    print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
    print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
    print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

    ##################Multi-view spectral clustering#####################

    # Use the MultiviewSpectralClustering instance to cluster the data
    m_spectral = MultiviewSpectralClustering(n_clusters=n_clusters, random_
↪state=RANDOM_SEED, n_init=100)
    m_clusters = m_spectral.fit_predict(m_data)

    # Compute nmi between true class labels and multi-view cluster labels
    m_nmi = nmi_score(labels, m_clusters)
    print('Multi-view Concatenated NMI Score: {0:.3f}\n'.format(m_nmi))

    return m_clusters
```

Creating a function to display data and the results of clustering

The following function plots both views of data given a dataset and corresponding labels.

[5]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(new_data[0][:, 0], new_data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(new_data[1][:, 0], new_data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Comparing multi-view and single-view spectral clustering on our data set with conditionally independent views

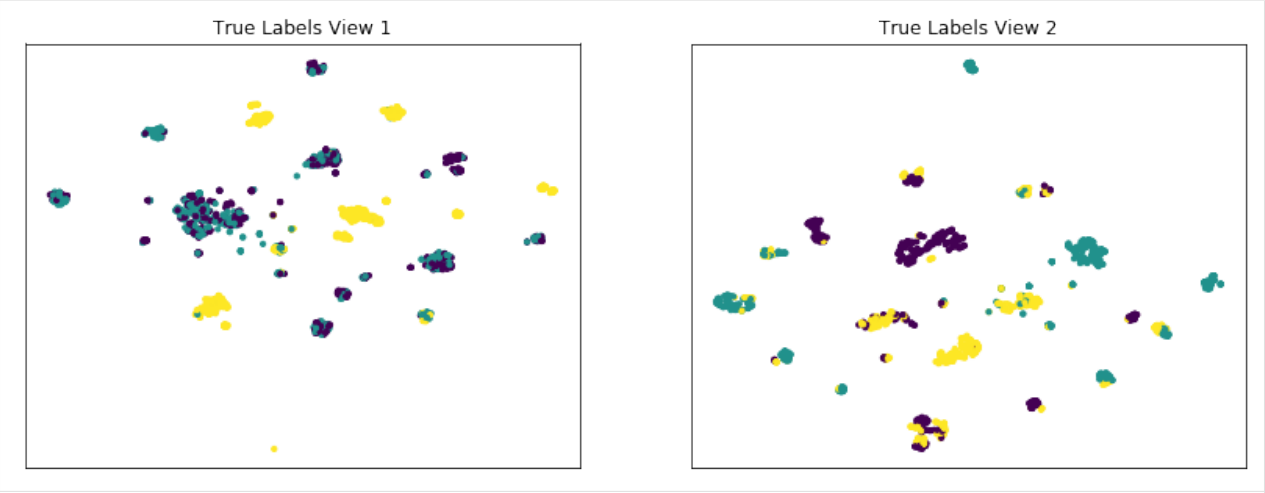
The co-training framework relies on the fundamental assumption that data views are conditionally independent. If all views are informative and conditionally independent, then Multi-view Spectral Clustering is expected to produce higher quality clusters than Single-view Spectral Clustering, for either view or for both views concatenated together. Here, we will evaluate the quality of clusters by using the normalized mutual information metric, which is essentially a measure of the purity of clusters with respect to the true underlying class labels.

As we see below, Multi-view Spectral Clustering produces clusters with lower purity than those produced by Single-view Spectral clustering on the concatenated views, which is surprising.

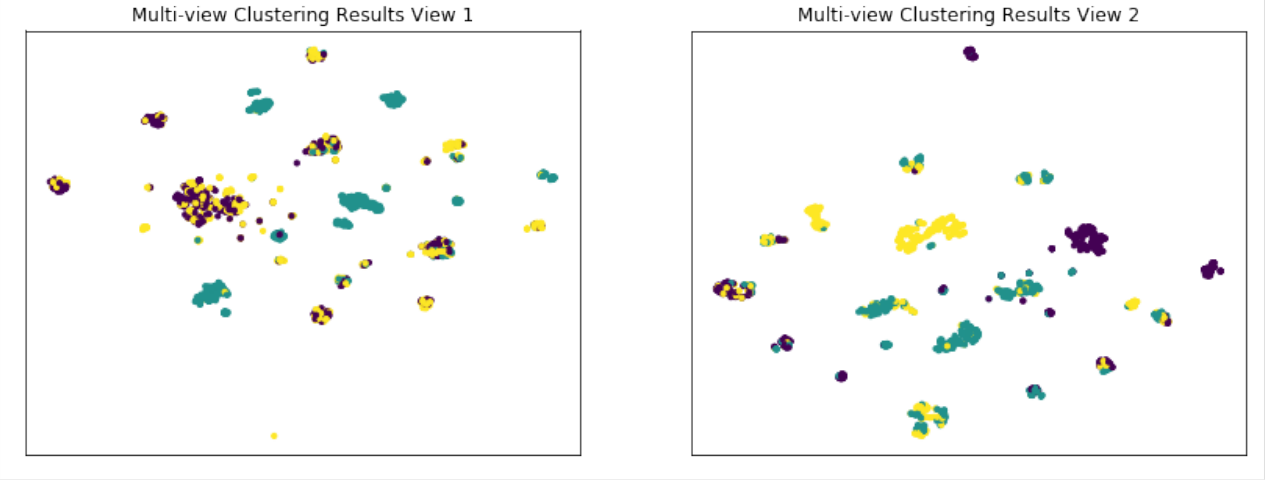
[6]:
```
data, labels = get_ci_data()
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 3)

# Running TSNE to display clustering results via low dimensional embedding
tsne = TSNE()
new_data = list()
new_data.append(tsne.fit_transform(data[0]))
new_data.append(tsne.fit_transform(data[1]))
display_plots('True Labels', new_data, labels)
display_plots('Multi-view Clustering Results', new_data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.316

Single-view View 2 NMI Score: 0.500

Single-view Concatenated NMI Score: 0.758

Multi-view Concatenated NMI Score: 0.552
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Creating an artificial dataset where the conditional independence assumption between views does not hold

Here, we create an artificial dataset where the conditional independence assumption between views, given the true labels, is violated. We again derive our dataset from the forest covertypes dataset from sklearn. However, this time, we use only the first 3 classes of the dataset, which will correspond to the 3 clusters for view 1. To produce view 2, we will apply a simple nonlinear transformation to view 1 using the logistic function, and we will apply a negligible amount of noise to the second view to avoid convergence issues. This will result in a dataset where the correspondance between views is very high.

[7]:
```
def get_cd_data(num_samples=500):

    #Load in the vectorized news group data from scikit-learn package
    cov = fetch_covtype()
    all_data = np.array(cov.data)
    all_targets = np.array(cov.target)

    #Set class pairings as described in the multiview clustering paper
    view1_classes = [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
    view2_classes = [4, 5, 6]

    #Create lists to hold data and labels for each of the classes across 2 different␣
↪views
    labels =  [num for num in range(len(view1_classes)) for _ in range(num_samples)]
    labels = np.array(labels)
    view1_data = list()
    view2_data = list()

    #Randomly sample 500 items from each of the selected classes in view1
    for class_num in view1_classes:
        class_data = all_data[(all_targets == class_num)]
        indices = np.random.choice(class_data.shape[0], num_samples)
        view1_data.append(class_data[indices])
    view1_data = np.concatenate(view1_data)


    #Construct view 2 by applying a nonlinear transformation
    #to data from view 1 comprised of a linear transformation
    #and a logistic nonlinearity
    t_mat =  np.random.random((view1_data.shape[1], 50))
    noise = 0.005 - 0.01*np.random.random((view1_data.shape[1], 50))
    t_mat *= noise
    transformed = view1_data @ t_mat
    view2_data = scp.special.expit(transformed)

    #Shuffle and normalize vectors
    shuffled_inds = np.random.permutation(num_samples * len(view1_classes))
    view1_data = np.vstack(view1_data)
    view2_data = np.vstack(view2_data)
    view1_data = view1_data[shuffled_inds]
    view2_data = view2_data[shuffled_inds]
    magnitudes1 = np.linalg.norm(view1_data, axis=0)
    magnitudes2 = np.linalg.norm(view2_data, axis=0)
    magnitudes1[magnitudes1 == 0] = 1
    magnitudes2[magnitudes2 == 0] = 1
    magnitudes1 = magnitudes1.reshape((1, -1))
    magnitudes2 = magnitudes2.reshape((1, -1))
    view1_data /= magnitudes1
    view2_data /= magnitudes2
    labels = labels[shuffled_inds]
    return [view1_data, view2_data], labels
```

Comparing multi-view and single-view spectral clustering on our data set with conditionally dependent views

As mentioned before, the co-training framework relies on the fundamental assumption that data views are conditionally independent. Here, we will again compare the performance of single-view and multi-view spectral clustering using the same methods as before, but on our conditionally dependent dataset.

As we see below, Multi-view Spectral Clustering does not beat the best Single-view spectral clustering performance with respect to purity, since that the views are conditionally dependent.

[8]:
```
data, labels = get_cd_data()
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 3)
```

```
Single-view View 1 NMI Score: 0.327

Single-view View 2 NMI Score: 0.160

Single-view Concatenated NMI Score: 0.239

Multi-view Concatenated NMI Score: 0.308
```

Multi-view vs Single-view Spectral Clustering

[1]:
```
import numpy as np
from numpy.random import multivariate_normal
from mvlearn.cluster.mv_spectral import MultiviewSpectralClustering
from sklearn.cluster import SpectralClustering
from sklearn.datasets import make_moons
from sklearn.metrics import normalized_mutual_info_score as nmi_score
import matplotlib
import matplotlib.pyplot as plt
import warnings

warnings.simplefilter('ignore') # Ignore warnings
%matplotlib inline
RANDOM_SEED=10
```

A function to generate 2 views of data for 2 classes

This function takes parameters for means, variances, and number of samples for class and generates data based on those parameters. The underlying probability distribution of the data is a multivariate gaussian distribution.

[2]:
```
def create_data(seed, vmeans, vvars, num_per_class=500):

    np.random.seed(seed)
    data = [[],[]]

    for view in range(2):
        for comp in range(len(vmeans[0])):
            cov = np.eye(2) * vvars[view][comp]
            comp_samples = np.random.multivariate_normal(vmeans[view][comp], cov,␣
↪size=num_per_class)
            data[view].append(comp_samples)
    for view in range(2):
        data[view] = np.vstack(data[view])

    labels = list()
    for ind in range(len(vmeans[0])):
        labels.append(ind * np.ones(num_per_class,))

    labels = np.concatenate(labels)

    return data, labels
```

Creating a function to display data and the results of clustering

The following function plots both views of data given a dataset and corresponding labels.

[3]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(data[0][:, 0], data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(data[1][:, 0], data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Creating a function to perform both single-view and multi-view spectral clustering

In the following function, we will perform single-view spectral clustering on the two views separately and on them concatenated together. We also perform multi-view clustering using the multi-view algorithm. We will also compare the performance of multi-view and single-view versions of spectral clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

[4]:
```
def perform_clustering(seed, m_data, labels, n_clusters, kernel='rbf'):

    ##################Single-view spectral clustering####################
    # Cluster each view separately
    s_spectral = SpectralClustering(n_clusters=n_clusters, random_state=RANDOM_SEED,
                                    affinity=kernel, n_init=100)
    s_clusters_v1 = s_spectral.fit_predict(m_data[0])
    s_clusters_v2 = s_spectral.fit_predict(m_data[1])

    # Concatenate the multiple views into a single view
    s_data = np.hstack(m_data)
    s_clusters = s_spectral.fit_predict(s_data)

    # Compute nmi between true class labels and single-view cluster labels
    s_nmi_v1 = nmi_score(labels, s_clusters_v1)
    s_nmi_v2 = nmi_score(labels, s_clusters_v2)
    s_nmi = nmi_score(labels, s_clusters)
    print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
    print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
    print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

    ##################Multi-view spectral clustering####################

    # Use the MultiviewSpectralClustering instance to cluster the data
    m_spectral = MultiviewSpectralClustering(n_clusters=n_clusters, random_
↪state=RANDOM_SEED,
```

(continues on next page)

(continued from previous page)

```
                        affinity=kernel, n_init=100)
    m_clusters = m_spectral.fit_predict(m_data)

    # Compute nmi between true class labels and multi-view cluster labels
    m_nmi = nmi_score(labels, m_clusters)
    print('Multi-view Concatenated NMI Score: {0:.3f}\n'.format(m_nmi))

    return m_clusters
```

General experimentation procedures

For each of the experiments below, we run both single-view spectral clustering and multi-view spectral clustering. For evaluating single-view performance, we run the algorithm on each view separately as well as all views concatenated together. We evalaute performance using normalized mutual information, which is a measure of cluster purity with respect to the true labels. For both algorithms, we use an n_init value of 100, which means that we run each algorithm across 100 random cluster initializations and select the best clustering results with respect to cluster inertia (within cluster sum-of-squared distances).

Performance when cluster components in both views are well separated

Cluster components 1: * Mean: [3, 3] (both views) * Covariance = I (both views)

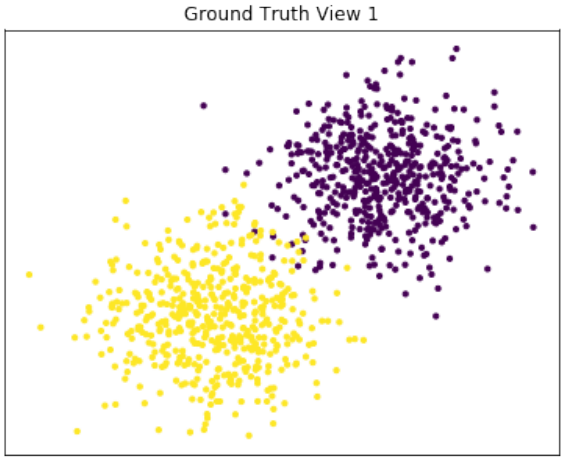
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

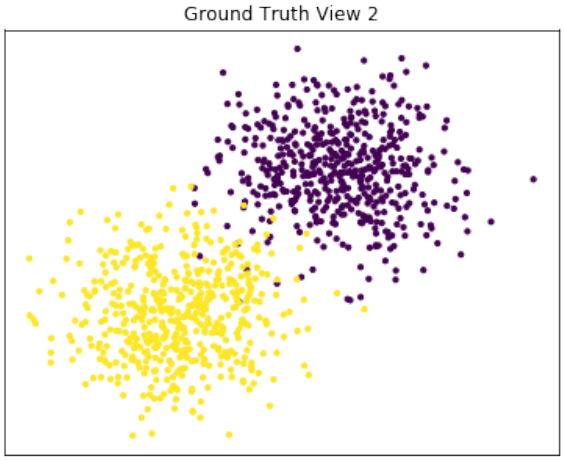
As we can see, multi-view spectral clustering performs better than single-view spectral clustering for all 3 inputs.

[5]:
```
v1_means = [[3, 3], [0, 0]]
v2_means = [[3, 3], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.896

Single-view View 2 NMI Score: 0.870

Single-view Concatenated NMI Score: 0.981

Multi-view Concatenated NMI Score: 0.990
```

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are relatively inseparable (highly overlapping) in both views

Cluster components 1: * Mean: [0.5, 0.5] (both views) * Covariance = I (both views)

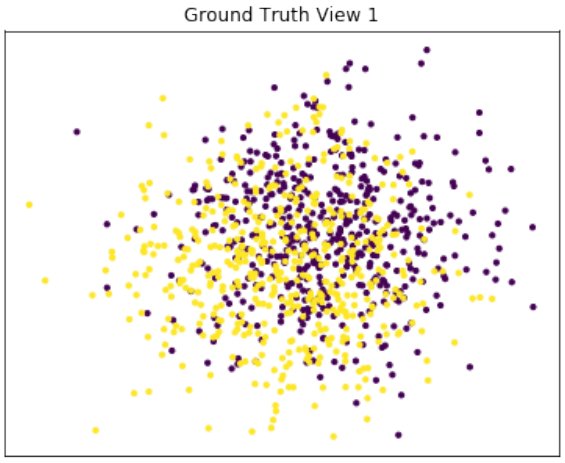
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

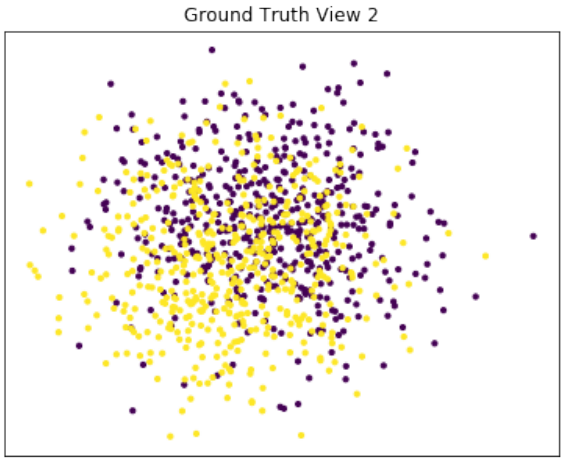
As we can see, multi-view spectral clustering performs about as poorly as single-view spectral clustering on all 3 input types.

[6]:
```
v1_means = [[0.5, 0.5], [0, 0]]
v2_means = [[0.5, 0.5], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

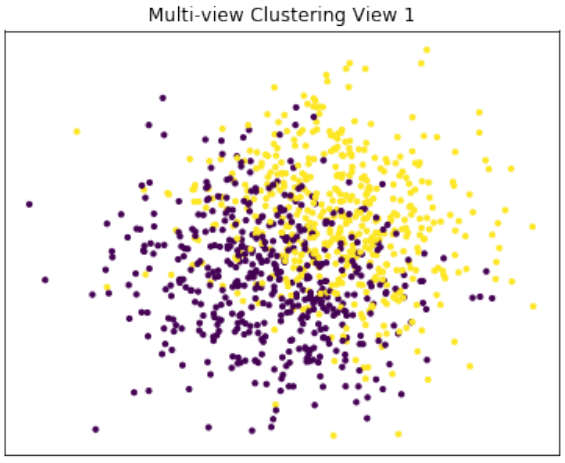
```
Single-view View 1 NMI Score: 0.064

Single-view View 2 NMI Score: 0.049

Single-view Concatenated NMI Score: 0.105

Multi-view Concatenated NMI Score: 0.110
```

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```



Performance when cluster components are somewhat separable (somewhat overlapping) in both views

Cluster components 1: * Mean: [1.5, 1.5] (both views) * Covariance = I (both views)

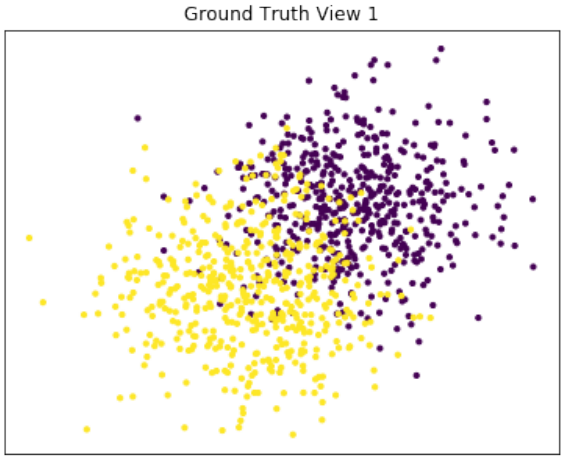
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

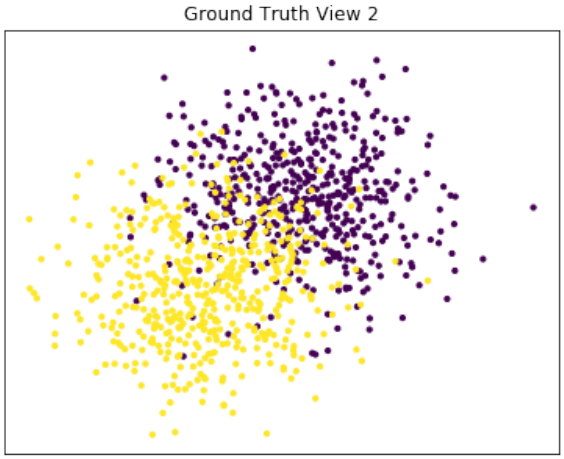
As we can see, multi-view spectral clustering performs better than single-view spectral clustering for all 3 inputs.

[7]:
```
v1_means = [[1.5, 1.5], [0, 0]]
v2_means = [[1.5, 1.5], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.410

Single-view View 2 NMI Score: 0.413

Single-view Concatenated NMI Score: 0.661

Multi-view Concatenated NMI Score: 0.649
```

```
<Figure size 432x288 with 0 Axes>
```






```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are highly overlapping in one view

Cluster components 1: * Mean: View 1 = [0.5, 0.5], View 2 = [2, 2] * Covariance = I (both views)

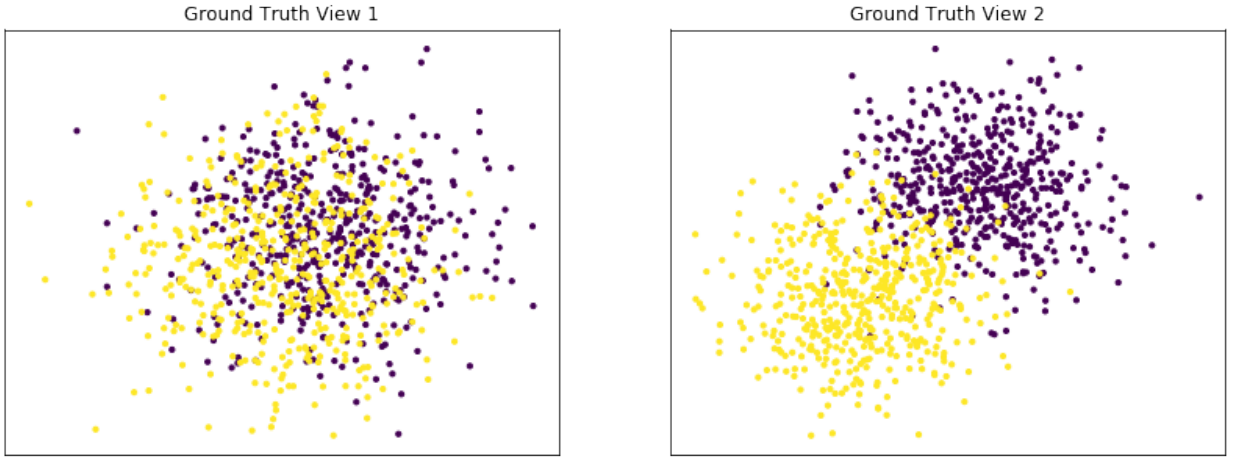
Cluster components 2: * Mean = [0, 0] (both views) * Covariance = I (both views)

As we can see, multi-view spectral clustering performs worse than single-view spectral clustering on the concatenated data and with the best view as input.

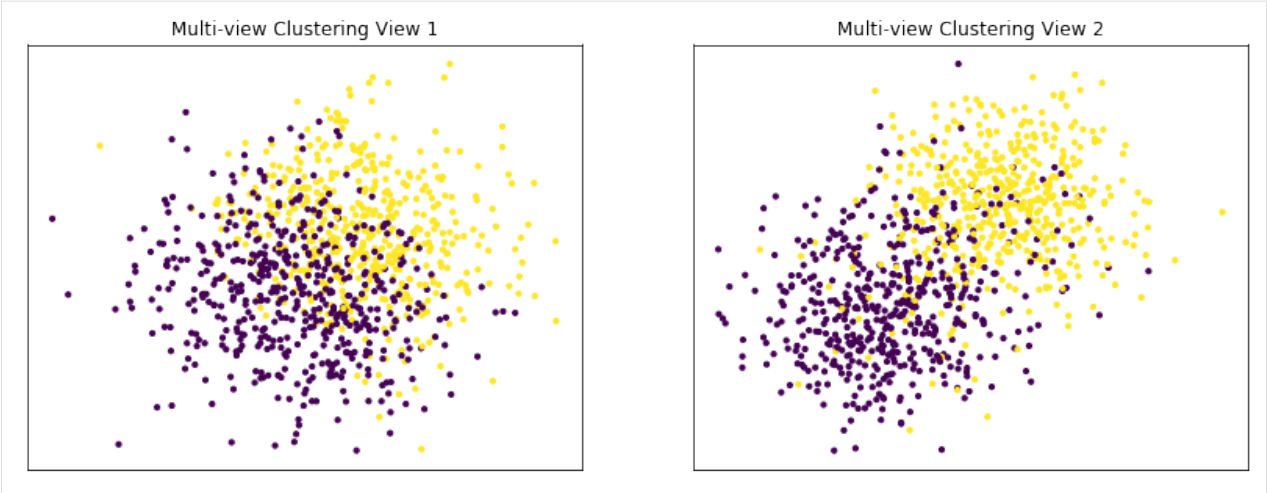
[8]:
```
v1_means = [[0.5, 0.5], [0, 0]]
v2_means = [[2, 2], [0, 0]]
v1_vars = [1, 1]
v2_vars = [1, 1]
vmeans = [v1_means, v2_means]
vvars = [v1_vars, v2_vars]

data, labels = create_data(RANDOM_SEED, vmeans, vvars)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.064

Single-view View 2 NMI Score: 0.588

Single-view Concatenated NMI Score: 0.610

Multi-view Concatenated NMI Score: 0.393
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Performance on moons data

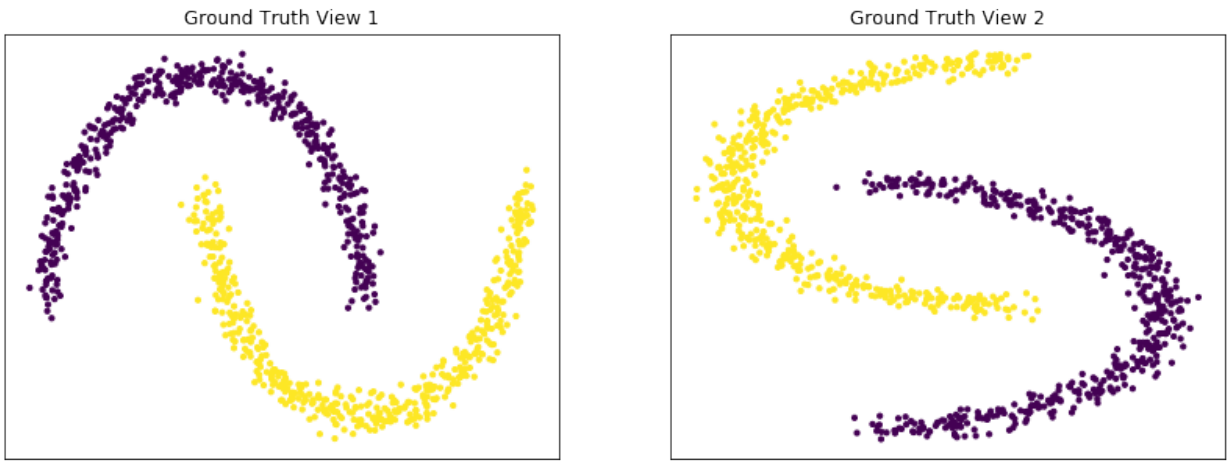
For this experiment, we use the sklearn make_moons function to make two interleaving half circles. We then use spectral clustering to separate the two views. In this experiment, the two views are identical. This experiment demonstrates the efficacy of using multi-view spectral clustering for non-convex clusters.

[9]:
```
def create_moons(seed, num_per_class=500):

    np.random.seed(seed)
    data = []
    labels = []

    for view in range(2):
        v_dat, v_labs = make_moons(num_per_class*2,
                random_state=seed + view, noise=0.05, shuffle=False)
        if view == 1:
            v_dat = v_dat[:, ::-1]

        data.append(v_dat)
    for ind in range(len(data)):
        labels.append(ind * np.ones(num_per_class,))
    labels = np.concatenate(labels)

    return data, labels
```

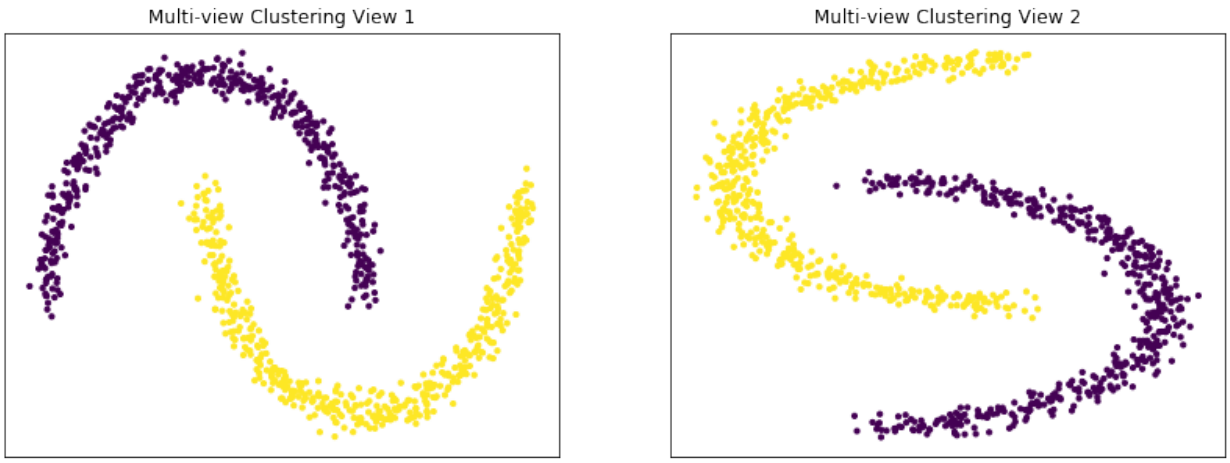
[10]:
```
data, labels = create_moons(RANDOM_SEED)
m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2, kernel='nearest_
↪neighbors')
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 1.000

Single-view View 2 NMI Score: 1.000

Single-view Concatenated NMI Score: 1.000

Multi-view Concatenated NMI Score: 1.000
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```



Conclusions

From the above experiments, we can see some of the advantages and limitations of multi-view spectral clustering. We can see that it outperforms single-view spectral clustering when data views are both informative and relatively separable. However, when one view is particularly inseparable, it can perform worse than its single-view analog. Additionally, we can see that the clustering algorithm is capable of clustering nonconvex-shaped clusters. These results were obtained using simple, simulated data, so results may vary on more complex data from the real world.

Multi-view Spherical KMeans

Note, this tutorial compares performance against the SphericalKMeans function from the spherecluster package which is not a installed dependency of mvlearn.

[1]:
```
!pip3 install spherecluster==0.1.7

from mvlearn.datasets import load_UCImultifeature
from mvlearn.cluster import MultiviewSphericalKMeans
```

(continues on next page)

(continued from previous page)

```
from sphere cluster import SphericalKMeans
import numpy as np
from sklearn.manifold import TSNE
from sklearn.metrics import normalized_mutual_info_score as nmi_score
import matplotlib.pyplot as plt
import warnings
warnings.simplefilter('ignore') # Ignore warnings
%matplotlib inline
```

```
Requirement already satisfied: sphere cluster==0.1.7 in /home/alex/MLenv/lib/python3.6/
↪site-packages (0.1.7)
Requirement already satisfied: scipy in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from sphere cluster==0.1.7) (1.3.1)
Requirement already satisfied: pytest in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from sphere cluster==0.1.7) (5.2.1)
Requirement already satisfied: numpy in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from sphere cluster==0.1.7) (1.18.1)
Requirement already satisfied: scikit-learn>=0.20 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from sphere cluster==0.1.7) (0.21.3)
Requirement already satisfied: nose in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from sphere cluster==0.1.7) (1.3.7)
Requirement already satisfied: importlib-metadata>=0.12; python_version < "3.8" in /
↪home/alex/MLenv/lib/python3.6/site-packages (from pytest->sphere cluster==0.1.7) (0.
↪23)
Requirement already satisfied: atomicwrites>=1.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (1.3.0)
Requirement already satisfied: py>=1.5.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (1.8.0)
Requirement already satisfied: packaging in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (19.2)
Requirement already satisfied: more-itertools>=4.0.0 in /home/alex/MLenv/lib/python3.
↪6/site-packages (from pytest->sphere cluster==0.1.7) (7.2.0)
Requirement already satisfied: attrs>=17.4.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (19.2.0)
Requirement already satisfied: wcwidth in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (0.1.7)
Requirement already satisfied: pluggy<1.0,>=0.12 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->sphere cluster==0.1.7) (0.13.0)
Requirement already satisfied: joblib>=0.11 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from scikit-learn>=0.20->sphere cluster==0.1.7) (0.14.1)
Requirement already satisfied: zipp>=0.5 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from importlib-metadata>=0.12; python_version < "3.
↪8"->pytest->sphere cluster==0.1.7) (0.6.0)
Requirement already satisfied: pyparsing>=2.0.2 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from packaging->pytest->sphere cluster==0.1.7) (2.3.0)
Requirement already satisfied: six in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from packaging->pytest->sphere cluster==0.1.7) (1.11.0)
WARNING: You are using pip version 19.3.1; however, version 20.1 is available.
You should consider upgrading via the 'pip install -upgrade pip' command.
```

```
/home/alex/MLenv/lib/python3.6/site-packages/sklearn/externals/joblib/__init__.py:15:␣
↪DeprecationWarning: sklearn.externals.joblib is deprecated in 0.21 and will be␣
↪removed in 0.23. Please import this functionality directly from joblib, which can␣
↪be installed with: pip install joblib. If this warning is raised when loading␣
↪pickled models, you may need to re-serialize those models with scikit-learn 0.21+.
  warnings.warn(msg, category=DeprecationWarning)
```

Load in UCI digits multiple feature dataset as an example

[2]:
```
RANDOM_SEED=5

# Load dataset along with labels for digits 0 through 4
n_class = 5
data, labels = load_UCImultifeature(select_labeled = list(range(n_class)))

# Just get the first two views of data
m_data = data[:2]
```

Creating a function to display data and the results of clustering

[3]:
```
def display_plots(pre_title, data, labels):

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))
    dot_size=10
    ax[0].scatter(data[0][:, 0], data[0][:, 1],c=labels,s=dot_size)
    ax[0].set_title(pre_title + ' View 1')
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(data[1][:, 0], data[1][:, 1],c=labels,s=dot_size)
    ax[1].set_title(pre_title + ' View 2')
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Multi-view spherical KMeans clustering on 2 views

Here we will compare the performance of the Multi-view and Single-view versions of spherical kmeans clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

As we can see, Multi-view clustering produces clusters with slightly higher purity compared to those produced by clustering on just a single view or by clustering the two views concatenated together.

[4]:
```
#################Single-view spherical kmeans clustering####################
# Cluster each view separately
s_kmeans = SphericalKMeans(n_clusters=n_class, random_state=RANDOM_SEED)
s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

# Concatenate the multiple views into a single view
s_data = np.hstack(m_data)
s_clusters = s_kmeans.fit_predict(s_data)

# Compute nmi between true class labels and single-view cluster labels
s_nmi_v1 = nmi_score(labels, s_clusters_v1)
s_nmi_v2 = nmi_score(labels, s_clusters_v2)
```

(continues on next page)

(continued from previous page)

```
s_nmi = nmi_score(labels, s_clusters)
print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

################Multi-view spherical kmeans clustering####################

# Use the MultiviewSphericalKMeans instance to cluster the data
m_kmeans = MultiviewSphericalKMeans(n_clusters=n_class, random_state=RANDOM_SEED)
m_clusters = m_kmeans.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi = nmi_score(labels, m_clusters)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))
```

```
Single-view View 1 NMI Score: 0.631

Single-view View 2 NMI Score: 0.730

Single-view Concatenated NMI Score: 0.730

Multi-view NMI Score: 0.823
```

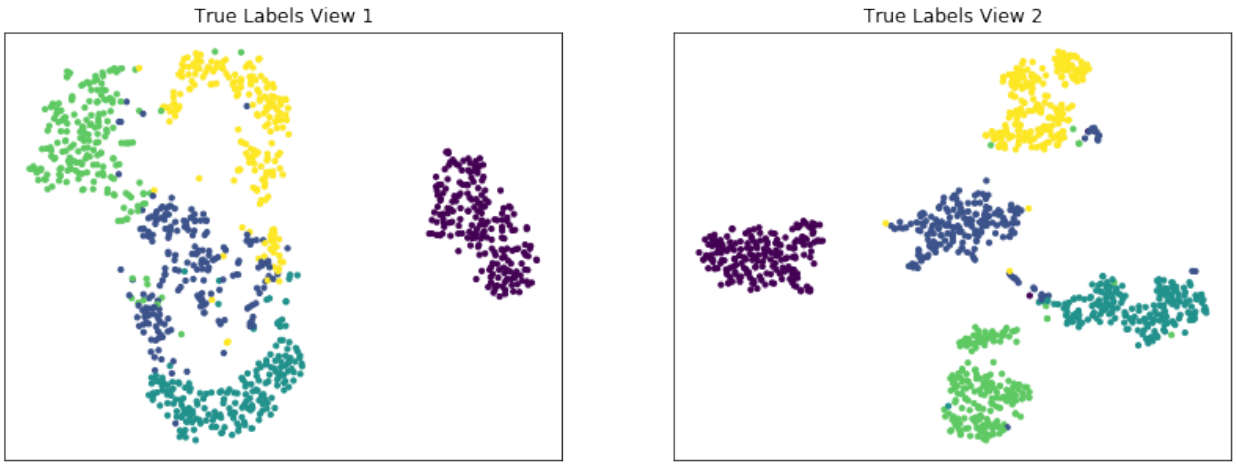
Plots of clusters produced by multi-view spectral clustering and the true clusters

We will display the clustering results of the Multi-view kmeans clustering algorithm below, along with the true class labels.

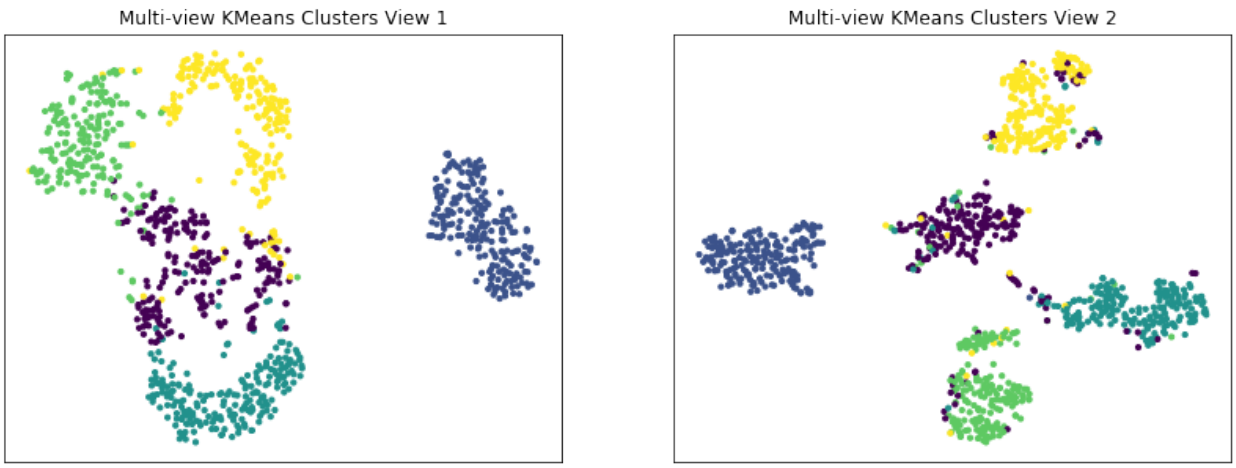
[5]:
```
# Running TSNE to display clustering results via low dimensional embedding
tsne = TSNE()
new_data_1 = tsne.fit_transform(m_data[0])
new_data_2 = tsne.fit_transform(m_data[1])
new_data = [new_data_1, new_data_2]
```

[6]:
```
display_plots('True Labels', new_data, labels)
display_plots('Multi-view KMeans Clusters', new_data, m_clusters)
```

```
<Figure size 432x288 with 0 Axes>
```



```
<Figure size 432x288 with 0 Axes>
```



Multi-view spherical KMeans clustering different parameters

Here we will again compare the performance of the Multi-view and Single-view versions of spherical kmeans clustering on data with 2 views. We will follow a similar procedure as before, but we will be using a different configuration of parameters for Multi-view Spherical KMeans Clustering.

Again, we can see that Multi-view clustering produces clusters with slightly higher purity compared to those produced by clustering on just a single view or by clustering the two views concatenated together.

[7]:
```
##################Single-view spherical kmeans clustering#####################
# Cluster each view separately
s_kmeans = SphericalKMeans(n_clusters=n_class, random_state=RANDOM_SEED)
s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

# Concatenate the multiple views into a single view
s_data = np.hstack(m_data)
s_clusters = s_kmeans.fit_predict(s_data)

# Compute nmi between true class labels and single-view cluster labels
s_nmi_v1 = nmi_score(labels, s_clusters_v1)
s_nmi_v2 = nmi_score(labels, s_clusters_v2)
s_nmi = nmi_score(labels, s_clusters)
print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

#################Multi-view spherical kmeans clustering#####################

# Use the MultiviewSphericalKMeans instance to cluster the data
m_kmeans = MultiviewSphericalKMeans(n_clusters=n_class,
        n_init=10, max_iter=6, patience=2, random_state=RANDOM_SEED)
m_clusters = m_kmeans.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi = nmi_score(labels, m_clusters)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))
```

```
Single-view View 1 NMI Score: 0.631

Single-view View 2 NMI Score: 0.730

Single-view Concatenated NMI Score: 0.730

Multi-view NMI Score: 0.684
```

Multi-view vs Single-view Spherical KMeans

Note, this tutorial compares performance against the SphericalKMeans function from the spherecluster package which is not a installed dependency of mvlearn.

[1]:
```
!pip3 install spherecluster==0.1.7

import numpy as np
from numpy.random import multivariate_normal
from mvlearn.cluster.mv_spherical_kmeans import MultiviewSphericalKMeans
from spherecluster import SphericalKMeans, sample_vMF
from sklearn.metrics import normalized_mutual_info_score as nmi_score
from sklearn.preprocessing import normalize
import matplotlib.pyplot as plt
%matplotlib inline
from mpl_toolkits.mplot3d import axes3d, Axes3D
import warnings
warnings.filterwarnings('ignore')
```

```
Requirement already satisfied: spherecluster==0.1.7 in /home/alex/MLenv/lib/python3.6/
↪site-packages (0.1.7)
Requirement already satisfied: scipy in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from spherecluster==0.1.7) (1.3.1)
Requirement already satisfied: pytest in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from spherecluster==0.1.7) (5.2.1)
Requirement already satisfied: numpy in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from spherecluster==0.1.7) (1.18.1)
Requirement already satisfied: nose in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from spherecluster==0.1.7) (1.3.7)
Requirement already satisfied: scikit-learn>=0.20 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from spherecluster==0.1.7) (0.21.3)
Requirement already satisfied: py>=1.5.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (1.8.0)
Requirement already satisfied: pluggy<1.0,>=0.12 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (0.13.0)
Requirement already satisfied: atomicwrites>=1.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (1.3.0)
Requirement already satisfied: wcwidth in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (0.1.7)
Requirement already satisfied: importlib-metadata>=0.12; python_version < "3.8" in /
↪home/alex/MLenv/lib/python3.6/site-packages (from pytest->spherecluster==0.1.7) (0.
↪23)
Requirement already satisfied: attrs>=17.4.0 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (19.2.0)
Requirement already satisfied: packaging in /home/alex/MLenv/lib/python3.6/
↪site-packages (from pytest->spherecluster==0.1.7) (19.2)
Requirement already satisfied: more-itertools>=4.0.0 in /home/alex/MLenv/lib/python3.
↪6/site-packages (from pytest->spherecluster==0.1.7) (7.2.0)
```

(continues on next page)

(continued from previous page)

```
Requirement already satisfied: joblib>=0.11 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from scikit-learn>=0.20->sphere cluster==0.1.7) (0.14.1)
Requirement already satisfied: zipp>=0.5 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from importlib-metadata>=0.12; python_version < "3.
↪8"->pytest->sphere cluster==0.1.7) (0.6.0)
Requirement already satisfied: pyparsing>=2.0.2 in /home/alex/MLenv/lib/python3.6/
↪site-packages (from packaging->pytest->sphere cluster==0.1.7) (2.3.0)
Requirement already satisfied: six in /home/alex/MLenv/lib/python3.6/site-packages␣
↪(from packaging->pytest->sphere cluster==0.1.7) (1.11.0)
WARNING: You are using pip version 19.3.1; however, version 20.1 is available.
You should consider upgrading via the 'pip install -upgrade pip' command.
```

```
/home/alex/MLenv/lib/python3.6/site-packages/sklearn/externals/joblib/__init__.py:15:␣
↪DeprecationWarning: sklearn.externals.joblib is deprecated in 0.21 and will be␣
↪removed in 0.23. Please import this functionality directly from joblib, which can␣
↪be installed with: pip install joblib. If this warning is raised when loading␣
↪pickled models, you may need to re-serialize those models with scikit-learn 0.21+.
  warnings.warn(msg, category=DeprecationWarning)
```

A function to generate 2 views of data for 2 classes

This function takes parameters for means, kappas (concentration parameter), and number of samples for class and generates data based on those parameters. The underlying probability distribution of the data is a von Mises-Fisher distribution.

[2]:
```
def create_data(seed, vmeans, vkappas, num_per_class=500):

    np.random.seed(seed)
    data = [[],[]]
    for view in range(2):
        for comp in range(len(vmeans[0])):
            comp_samples = sample_vMF(vmeans[view][comp],
                        vkappas[view][comp], num_per_class)
            data[view].append(comp_samples)
    for view in range(2):
        data[view] = np.vstack(data[view])

    labels = list()
    for ind in range(len(vmeans[0])):
        labels.append(ind * np.ones(num_per_class,))

    labels = np.concatenate(labels)

    return data, labels
```

Creating a function to display data and the results of clustering

The following function plots both views of data given a dataset and corresponding labels.

[3]:
```
def display_plots(pre_title, data, labels):
    plt.ion()
    # plot the views
    plt.figure()
```

(continues on next page)

(continued from previous page)

```
    fig = plt.figure(figsize=(14, 10))
    for v in range(2):
        ax = fig.add_subplot(
            1, 2, v+1, projection='3d',
            xlim=[-1.1, 1.1], ylim=[-1.1, 1.1], zlim=[-1.1, 1.1]
        )
        ax.scatter(data[v][:, 0], data[v][:, 1], data[v][:, 2], c=labels, s=8)
        ax.set_title(pre_title + ' View ' + str(v))
        plt.axis('off')

    plt.show()
```

Creating a function to perform both single-view and multi-view spherical kmeans clustering

In the following function, we will perform single-view spherical kmeans clustering on the two views separately and on them concatenated together. We also perform multi-view clustering using the multi-view algorithm. We will also compare the performance of multi-view and single-view versions of the spherical kmeans clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

[4]:
```
def perform_clustering(seed, m_data, labels, n_clusters):
    ##################Single-view spherical kmeans clustering#####################
    # Cluster each view separately
    s_kmeans = SphericalKMeans(n_clusters=n_clusters, random_state=seed, n_init=100)
    s_clusters_v1 = s_kmeans.fit_predict(m_data[0])
    s_clusters_v2 = s_kmeans.fit_predict(m_data[1])

    # Concatenate the multiple views into a single view
    s_data = np.hstack(m_data)
    s_clusters = s_kmeans.fit_predict(s_data)

    # Compute nmi between true class labels and single-view cluster labels
    s_nmi_v1 = nmi_score(labels, s_clusters_v1)
    s_nmi_v2 = nmi_score(labels, s_clusters_v2)
    s_nmi = nmi_score(labels, s_clusters)
    print('Single-view View 1 NMI Score: {0:.3f}\n'.format(s_nmi_v1))
    print('Single-view View 2 NMI Score: {0:.3f}\n'.format(s_nmi_v2))
    print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

    ##################Multi-view spherical kmeans clustering#####################

    # Use the MultiviewKMeans instance to cluster the data
    m_kmeans = MultiviewSphericalKMeans(n_clusters=n_clusters, n_init=100, random_
↪state=seed)
    m_clusters = m_kmeans.fit_predict(m_data)

    # Compute nmi between true class labels and multi-view cluster labels
    m_nmi = nmi_score(labels, m_clusters)
    print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi))

    return m_clusters
```

General experimentation procedures

For each of the experiments below, we run both single-view spherical kmeans clustering and multi-view spherical kmeans clustering. For evaluating single-view performance, we run the algorithm on each view separately as well as all views concatenated together. We evalaute performance using normalized mutual information, which is a measure of cluster purity with respect to the true labels. For both algorithms, we use an n_init value of 100, which means that we run each algorithm across 100 random cluster initializations and select the best clustering results with respect to cluster inertia.

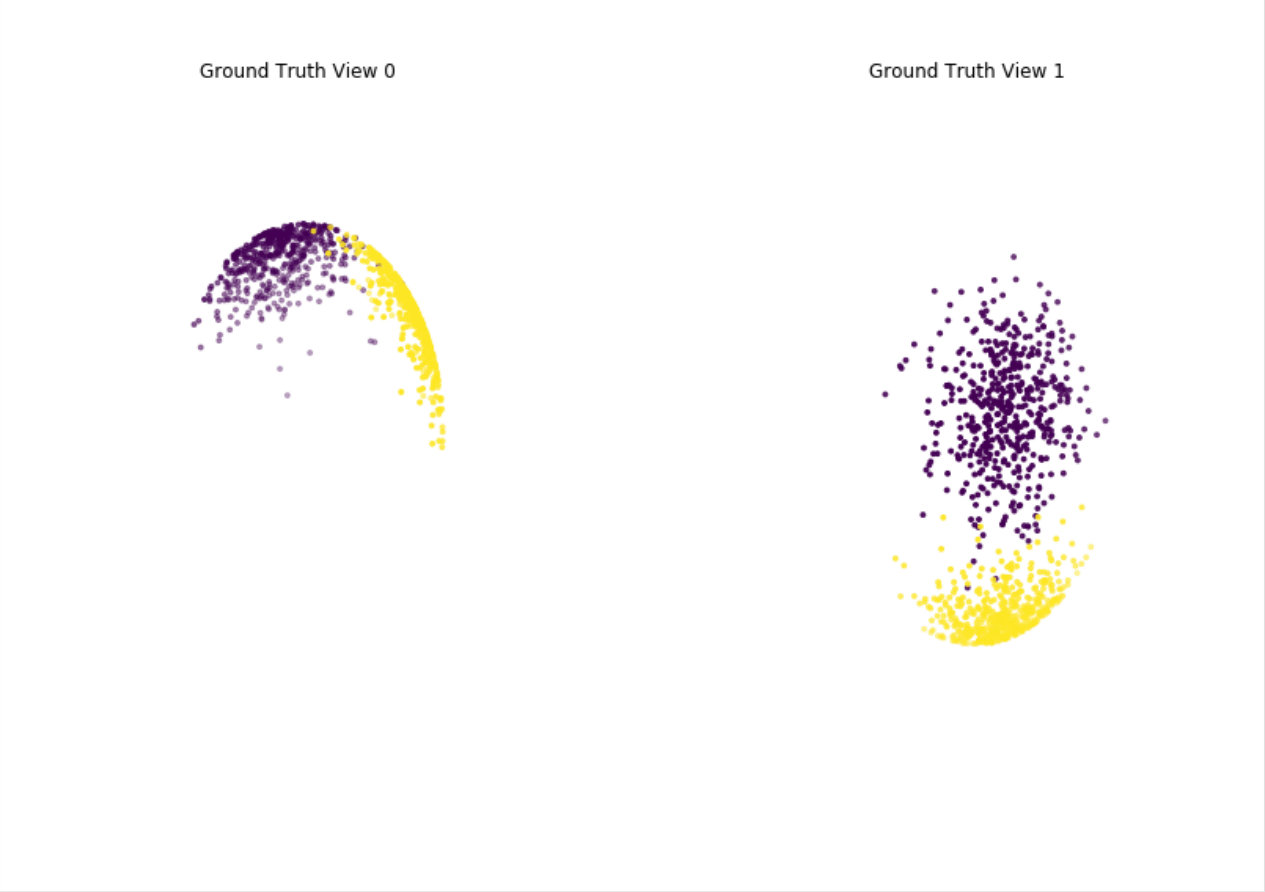
Performance when cluster components in both views are well separated

As we can see, multi-view kmeans clustering performs about as well as single-view spherical kmeans clustering for the concatenated views, and single-view spherical kmeans clustering for view 1.

[5]:
```
RANDOM_SEED=10

v1_kappas = [15, 15]
v2_kappas = [15, 15]
kappas = [v1_kappas, v2_kappas]
v1_mus = np.array([[-1, 1, 1],[1, 1, 1]])
v1_mus = normalize(v1_mus)
v2_mus = np.array([[1, -1, 1],[1, -1, -1]])
v2_mus = normalize(v2_mus)
v_means = [v1_mus, v2_mus]
data, labels = create_data(RANDOM_SEED, v_means, kappas)

m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

```
Single-view View 1 NMI Score: 0.906

Single-view View 2 NMI Score: 0.920

Single-view Concatenated NMI Score: 1.000

Multi-view NMI Score: 1.000
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are relatively inseparable (highly overlapping) in both views

As we can see, multi-view spherical kmeans clustering performs about as poorly as single-view spherical kmeans clustering across both individual views and concatenated views as inputs.

[6]:
```
v1_kappas = [15, 15]
v2_kappas = [15, 15]
kappas = [v1_kappas, v2_kappas]
v1_mus = np.array([[0.5, 1, 1],[1, 1, 1]])
v1_mus = normalize(v1_mus)
v2_mus = np.array([[1, -1, 1],[1, -1, 0.5]])
v2_mus = normalize(v2_mus)
v_means = [v1_mus, v2_mus]
data, labels = create_data(RANDOM_SEED, v_means, kappas)

m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

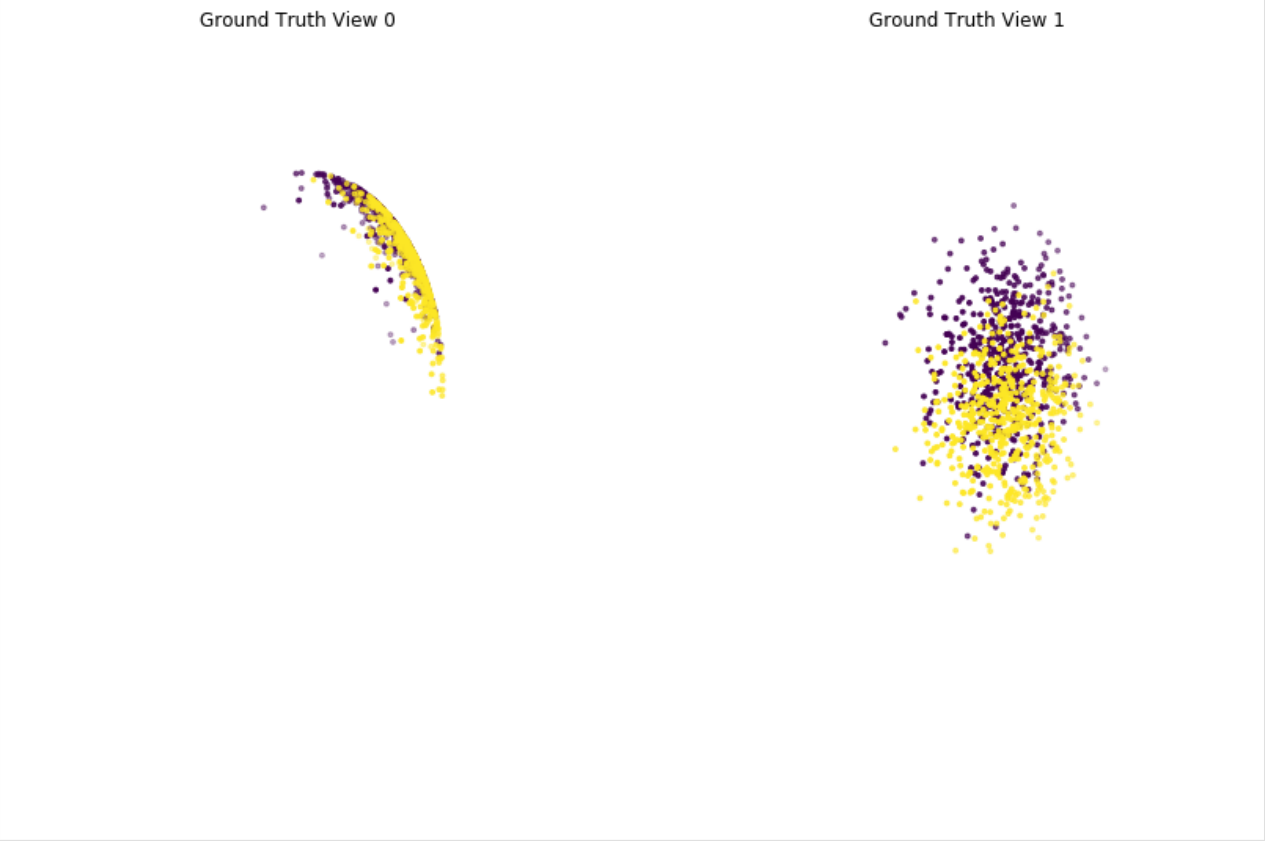
```
Single-view View 1 NMI Score: 0.102

Single-view View 2 NMI Score: 0.112

Single-view Concatenated NMI Score: 0.199
```

(continues on next page)

(continued from previous page)

```
Multi-view NMI Score: 0.204
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are somewhat separable (somewhat overlapping) in both views

Again we can see that multi-view spherical kmeans clustering performs about as well as single-view spherical kmeans clustering for the concatenated views, and both of these perform better than on single-view spherical kmeans clustering for just one view.

[7]:
```
v1_kappas = [15, 10]
v2_kappas = [10, 15]
kappas = [v1_kappas, v2_kappas]
v1_mus = np.array([[-0.5, 1, 1],[1, 1, 1]])
v1_mus = normalize(v1_mus)
v2_mus = np.array([[1, -1, 1],[1, -1, -0.2]])
v2_mus = normalize(v2_mus)
v_means = [v1_mus, v2_mus]
data, labels = create_data(RANDOM_SEED, v_means, kappas)

m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

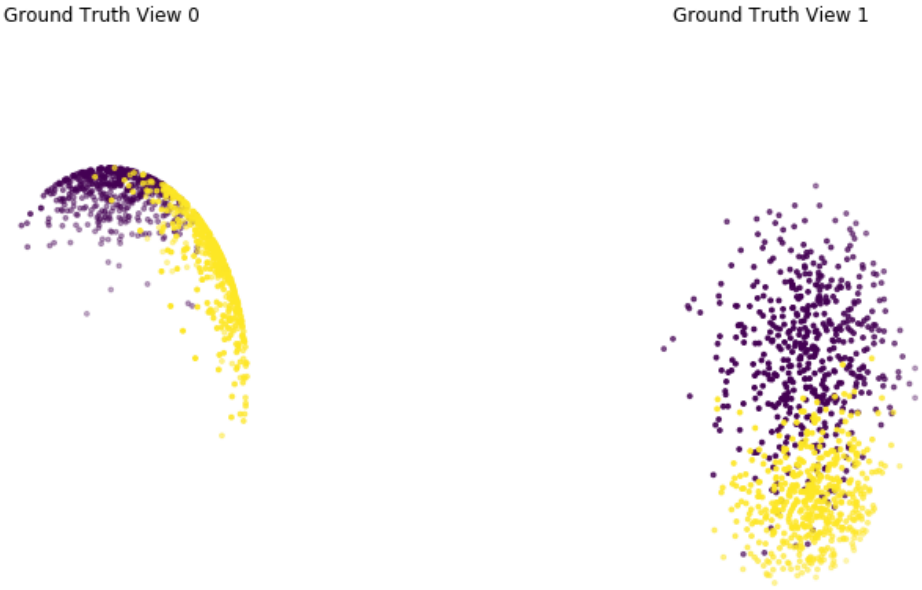
```
Single-view View 1 NMI Score: 0.677

Single-view View 2 NMI Score: 0.552
```

(continues on next page)

(continued from previous page)

```
Single-view Concatenated NMI Score: 0.827

Multi-view NMI Score: 0.831
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Performance when cluster components are highly overlapping in one view

As we can see, multi-view spherical kmeans clustering performs worse than single-view spherical kmeans clustering with concatenated views as inputs and with the best view as the input.

[8]:
```
v1_kappas = [15, 15]
v2_kappas = [15, 15]
kappas = [v1_kappas, v2_kappas]
v1_mus = np.array([[1, -0.5, 1],[1, 1, 1]])
v1_mus = normalize(v1_mus)
v2_mus = np.array([[1, -1, 1],[1, -1, 0.6]])
v2_mus = normalize(v2_mus)
v_means = [v1_mus, v2_mus]
data, labels = create_data(RANDOM_SEED, v_means, kappas)

m_clusters = perform_clustering(RANDOM_SEED, data, labels, 2)
display_plots('Ground Truth' ,data, labels)
display_plots('Multi-view Clustering' ,data, m_clusters)
```

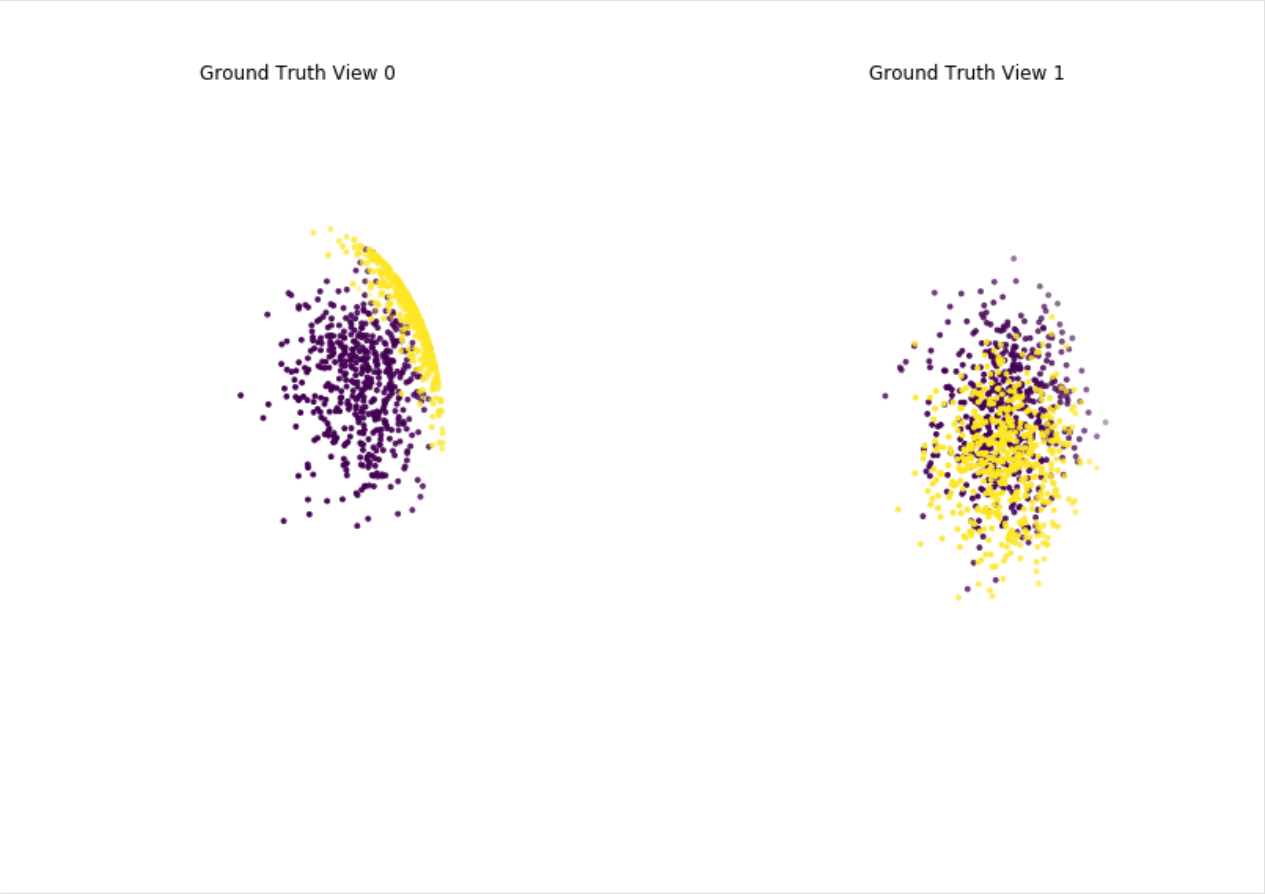
```
Single-view View 1 NMI Score: 0.740

Single-view View 2 NMI Score: 0.077

Single-view Concatenated NMI Score: 0.768
```

(continues on next page)

(continued from previous page)

```
Multi-view NMI Score: 0.741
```

```
<Figure size 432x288 with 0 Axes>
```




```
<Figure size 432x288 with 0 Axes>
```

Conclusions

Here, we have seen some of the limitations of multi-view spherical kmeans clustering. From the experiments above, it is apparent that multi-view spherical kmeans clustering performs equally as well or worse than single-view spherical kmeans clustering on concatenated data when views are informative but the data is fairly simple (i.e. only has 2 features per view). However, it is clear that the multi-view spherical kmeans algorithm does perform better on well separated cluster components than it does on highly overlapping cluster components, which does validate it's basic functionality as a clustering algorithm.

Using the Multi-view Clustering Algorithm to Cluster Data with Multiple Views

[1]:
```
from mvlearn.datasets.base import load_UCImultifeature
from mvlearn.cluster import MultiviewCoRegSpectralClustering
from mvlearn.plotting import quick_visualize
import numpy as np
from sklearn.cluster import SpectralClustering
from sklearn.metrics import normalized_mutual_info_score as nmi_score
import scipy
import warnings

warnings.simplefilter('ignore') # Ignore warnings
%matplotlib inline
RANDOM_SEED=10
```

Load the UCI Digits Multiple Features Data Set as an Example Data Set

[2]:
```
# Load dataset along with labels for digits 0 through 4
n_class = 5
m_data, labels = load_UCImultifeature(select_labeled = list(range(n_class)))
```

Running Co-Regularized Multi-view Spectral Clustering on the Data with 6 Views

Here we will compare the performance of the Co-Regularized Multi-view and Single-view versions of spectral clustering. We will evaluate the purity of the resulting clusters from each algorithm with respect to the class labels using the normalized mutual information metric.

As we can see, Co-Regularized Multi-view clustering produces clusters with higher purity compared to those produced by Single-view clustering for all 3 input types.

[3]:
```
##################Single-view spectral clustering#####################
# Cluster each view separately and compute nmi
s_spectral = SpectralClustering(n_clusters=n_class, random_state=RANDOM_SEED, n_
↪init=100)

for i in range(len(m_data)):
    s_clusters = s_spectral.fit_predict(m_data[i])
    s_nmi = nmi_score(labels, s_clusters, average_method='arithmetic')
    print('Single-view View {0:d} NMI Score: {1:.3f}\n'.format(i + 1, s_nmi))

# Concatenate the multiple views into a single view and produce clusters
s_data = np.hstack(m_data)
s_clusters = s_spectral.fit_predict(s_data)

s_nmi = nmi_score(labels, s_clusters)
print('Single-view Concatenated NMI Score: {0:.3f}\n'.format(s_nmi))

#######Co-Regularized Multi-view spectral clustering#################


# Use the MultiviewSpectralClustering instance to cluster the data
m_spectral1 = MultiviewCoRegSpectralClustering(n_clusters=n_class,
            random_state=RANDOM_SEED, n_init=100)
m_clusters1 = m_spectral1.fit_predict(m_data)

# Compute nmi between true class labels and multi-view cluster labels
m_nmi1 = nmi_score(labels, m_clusters1)
print('Multi-view NMI Score: {0:.3f}\n'.format(m_nmi1))
```

```
Single-view View 1 NMI Score: 0.620

Single-view View 2 NMI Score: 0.007

Single-view View 3 NMI Score: 0.004

Single-view View 4 NMI Score: -0.000

Single-view View 5 NMI Score: 0.007

Single-view View 6 NMI Score: 0.010
```

(continues on next page)

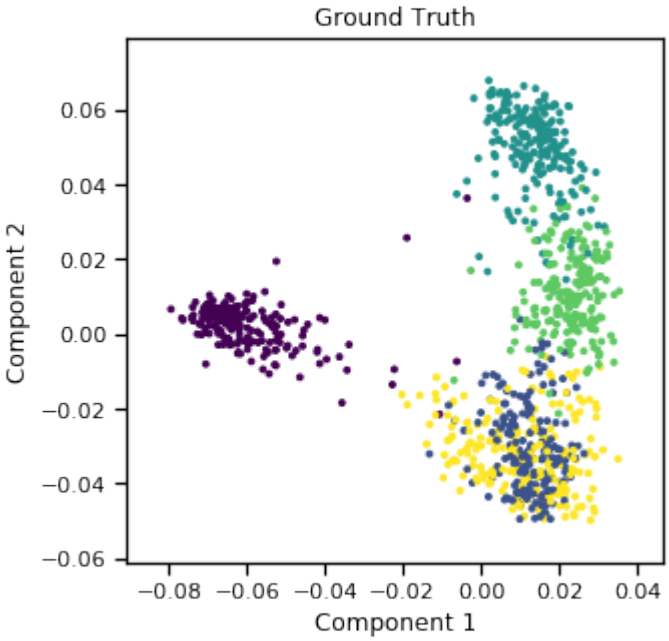
(continued from previous page)

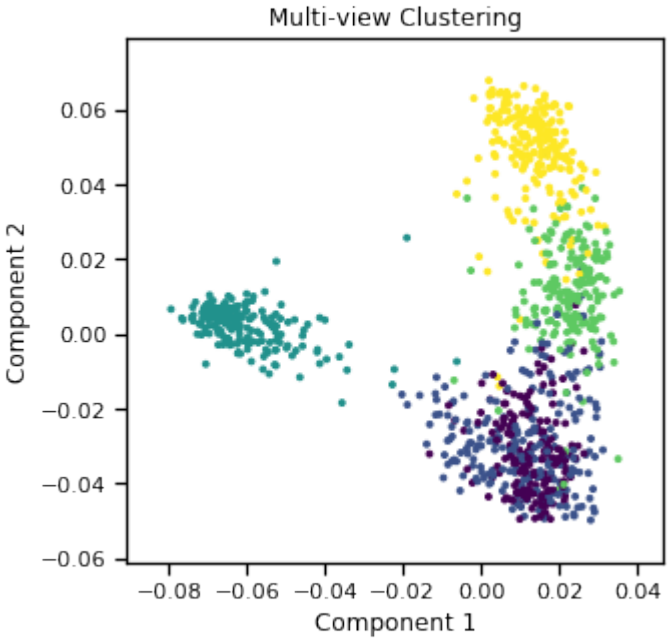
```
Single-view Concatenated NMI Score: 0.008

Multi-view NMI Score: 0.866
```

Plots of clusters produced by multi-view spectral clustering and the true clusters

We will display the clustering results of the Co-Regularized Multi-view spectral clustering algorithm below, along with the true class labels.

[4]:
```
quick_visualize(m_data, labels=labels, title='Ground Truth', scatter_kwargs={'s':8})
quick_visualize(m_data, labels=m_clusters1, title='Multi-view Clustering', scatter_
↪kwargs={'s':8})
```





Multi-view Vs Single-view Visualization and Clustering

Here, we directly compare multi-view methods available within *mvlearn* to analagous single-view methods. Using the UCI Multiple Features Dataset, we first examine the dataset by viewing it after using dimensionality reduction techniques, then we perform unsupervised clustering and compare the results to the analagous single-view methods.

[1]:
```
from mvlearn.datasets import load_UCImultifeature
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

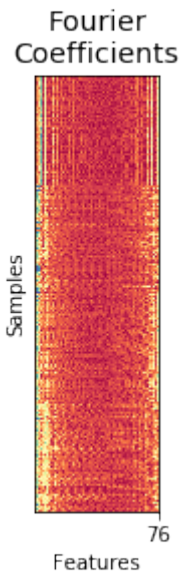
Load 6-view, 4-class data from the Multiple Features Dataset. The full 6 views with all features will be used for clustering.

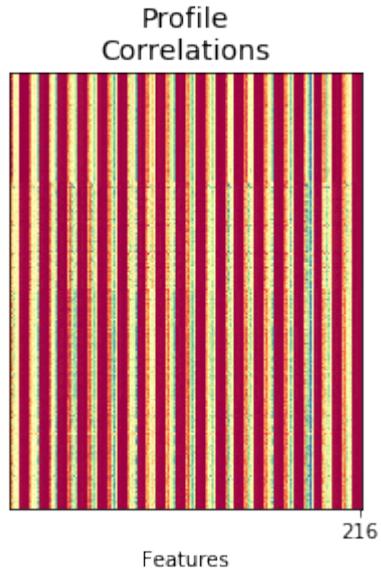
[2]:
```
# Load 4-class, multi-view data
Xs, y = load_UCImultifeature(select_labeled=[0,1,2,3])
#     Six views of handwritten digit images
#     1. 76 Fourier coefficients of the character shapes
#     2. 216 profile correlations
#     3. 64 Karhunen-Love coefficients
#     4. 240 pixel averages of the images from 2x3 windows
#     5. 47 Zernike moments
#     6. 6 morphological features
view_names = ['Fourier\nCoefficients', 'Profile\nCorrelations', 'Karhunen-\nLoeve',
              'Pixel\nAverages', 'Zernike\nMoments', 'Morphological\nFeatures']

order = np.argsort(y)
sub_samp = np.arange(0, Xs[0].shape[0], step=3)
set_aspect = 'equal'  # 'equal' or 'auto'
set_cmap = 'Spectral'

#row_orders = np.argsort(y)
```

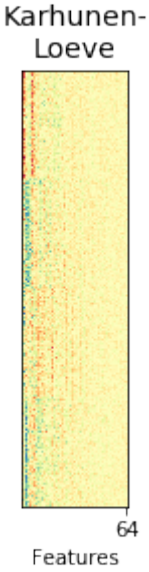
(continues on next page)

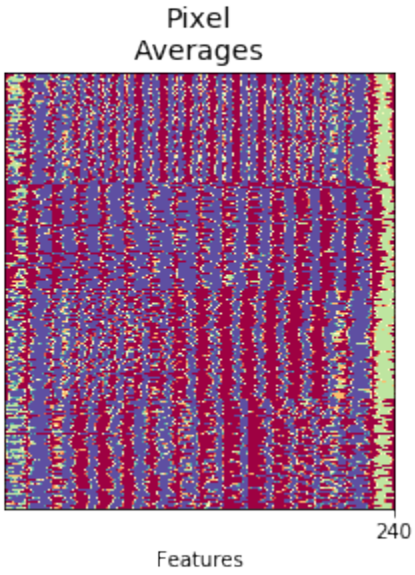
(continued from previous page)

```
for i, view in enumerate(Xs):
    sorted_view = view[order,:].copy()
    sorted_view = sorted_view[sub_samp,:]
    if set_aspect == 'auto':
        plt.figure(figsize=(1.5,4.5))
    else:
        plt.figure()

    # Scale matrix to [0, 1]
    minim = np.min(sorted_view)
    maxim = np.max(sorted_view)
    sorted_view = (sorted_view - minim) / (maxim - minim)

    plt.imshow(sorted_view, cmap=set_cmap, aspect=set_aspect)
    #plt.title('View {}'.format(i+1))
    plt.title(view_names[i], fontsize=14)
    plt.yticks([], "")
    max_dim = view.shape[1]
    plt.xticks([max_dim-1], [str(max_dim)])
    if i == 0:
        plt.ylabel('Samples')
    if i == 5:
        plt.colorbar()
    plt.xlabel('Features')
    plt.show()
```

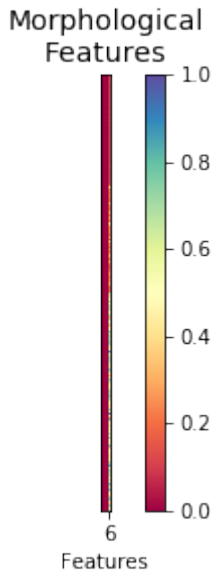



Profile
Correlations
216
Features


Karhunen-
Loeve
64
Features


Pixel
Averages
240
Features


Zernike
Moments
47
Features



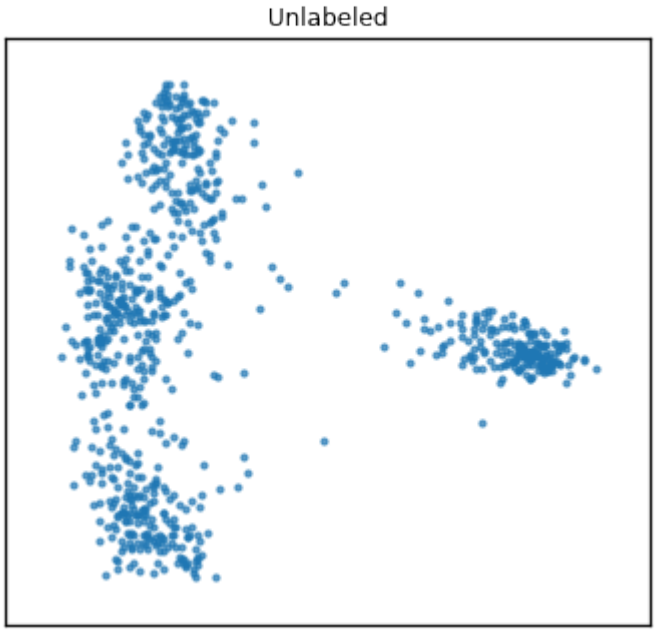
Define a function to rearrange the predicted labels so that the predicted class '0' corresponds better to the true class '0'. This is only used so that the colors generated by the labels in the prediction plots can be more easily compared to the true labels.

[3]:
```
from sklearn.metrics import confusion_matrix

def rearrange_labels(y_true, y_pred):
    conf_mat = confusion_matrix(y_true, y_pred)
    maxes = np.argmax(conf_mat, axis=0)
    y_pred_new = np.zeros_like(y_pred)
    for i, new in enumerate(maxes):
        y_pred_new[y_pred==i] = new
    return y_pred_new
```

Comparing Dimensionality Reduction Techniques

As one might do with a new dataset, we first visualize the data in 2 dimensions. For multi-view data, rather than using PCA, we use Multi-view Multi-dimensional Scaling (MVMDS) available in the package to capture the common principal components across views. This is performed automatically within the quick_visualize function. From the unlabeled plot, it is clear that there may be 4 underlying clusters, so unsupervised clustering with 4 clusters may be a natural next step in analyzing this data.

[4]:
```
from mvlearn.plotting import quick_visualize

# Use all 6 views available to reduce the dimensionality, since MVMDS is not limited
sca_kwargs = {'alpha' : 0.7, 's' : 10}

quick_visualize(Xs, title="Unlabeled", ax_ticks=False,
                ax_labels=False, scatter_kwargs=sca_kwargs)
quick_visualize(Xs, labels=y, title="True Labels", ax_ticks=False,
                ax_labels=False, scatter_kwargs=sca_kwargs)
```



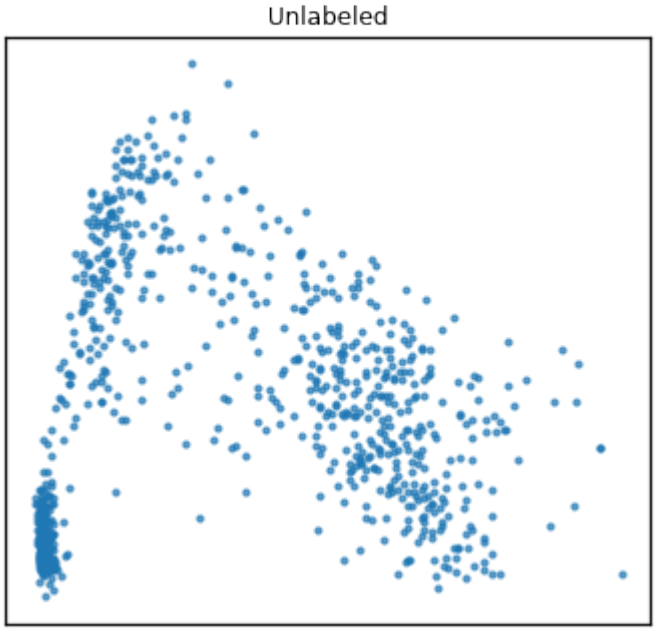
As a comparison, we concatenate the views and use PCA to reduce the dimensionality. From the unlabeled plot, it is much less clear how many underlying classes there are, so PCA was not as useful for visualizing the data if our goal was to determine underlying clusters.

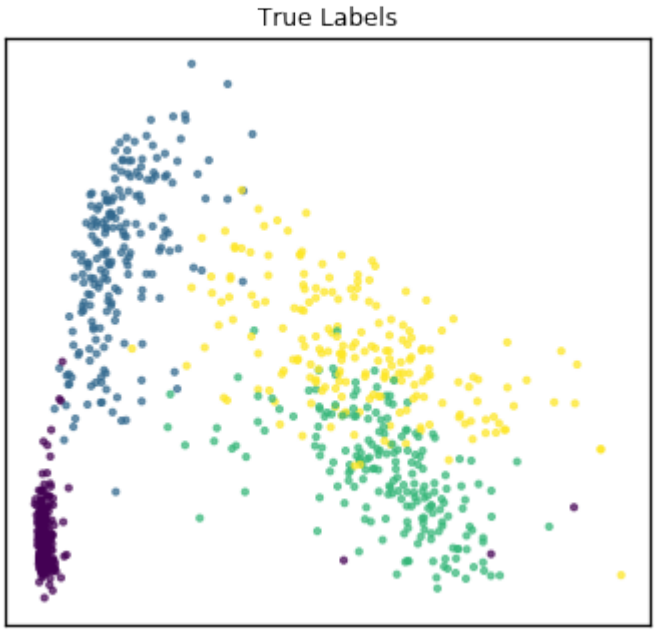
[5]:
```
from sklearn.decomposition import PCA

# Concatenate views to get naive single view
X_viewing = np.hstack([Xs[i] for i in range(len(Xs))])

# Use PCA for dimensionality reduction on the naive single view
pca = PCA(n_components=2)
```

(continues on next page)

(continued from previous page)

```
pca_X = pca.fit_transform(X_viewing)

plt.figure(figsize=(5, 5))
plt.scatter(pca_X[:,0], pca_X[:,1], **sca_kwargs)
plt.xticks([], [])
plt.yticks([], [])
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.title("Unlabeled")
plt.show()

plt.figure(figsize=(5, 5))
plt.scatter(pca_X[:,0], pca_X[:,1], c=y, **sca_kwargs)
plt.xticks([], [])
plt.yticks([], [])
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.title("True Labels")
plt.show()
```





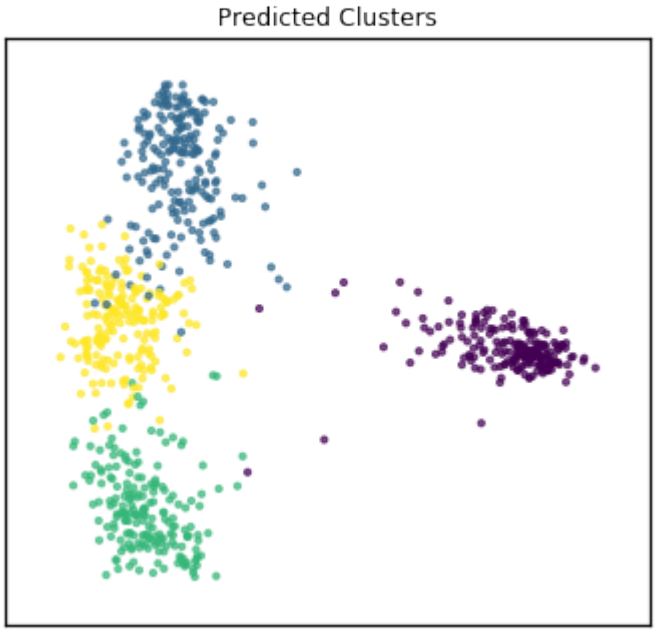
Comparing Clustering Techniques using the Full Feature Space

Now, assuming we are trying to group the samples into 4 clusters (as was much more obvious after using *mvlearn*'s dimensionality reduction viewing method), we compare multi-view clustering techniques to single-view counterparts. Specifically, we compare 6-view spectral clustering in *mvlearn* with single view spectral clustering from *scikit-learn*. For multi-view clustering, all 6 full views of data (not the dimensionality-reduced data). For single-view comparison, we concatenate these 6 full views into a single large matrix, the same as what we did before for PCA.

Since we have the true class labels, we assess the clustering accuracy with a homogeneity score.

[6]:
```
from mvlearn.cluster import MultiviewSpectralClustering

mv_clust = MultiviewSpectralClustering(n_clusters=4, affinity='nearest_neighbors')
mvlearn_cluster_labels = mv_clust.fit_predict(Xs)

# Test the accuracy of the clustering
from sklearn.metrics import homogeneity_score
mv_score = homogeneity_score(y, mvlearn_cluster_labels)
print('Multi-view homogeneity score: {0:.3f}'.format(mv_score))

# Use function defined at beginning of notebook to rearrange the labels
# for easier visual comparison to true labeled plot
mvlearn_cluster_labels = rearrange_labels(y, mvlearn_cluster_labels)

# Visualize the clusters in the 2-dimensional space
quick_visualize(Xs, labels=mvlearn_cluster_labels, title="Predicted Clusters",
                ax_ticks=False, ax_labels=False, scatter_kwargs=sca_kwargs)
```

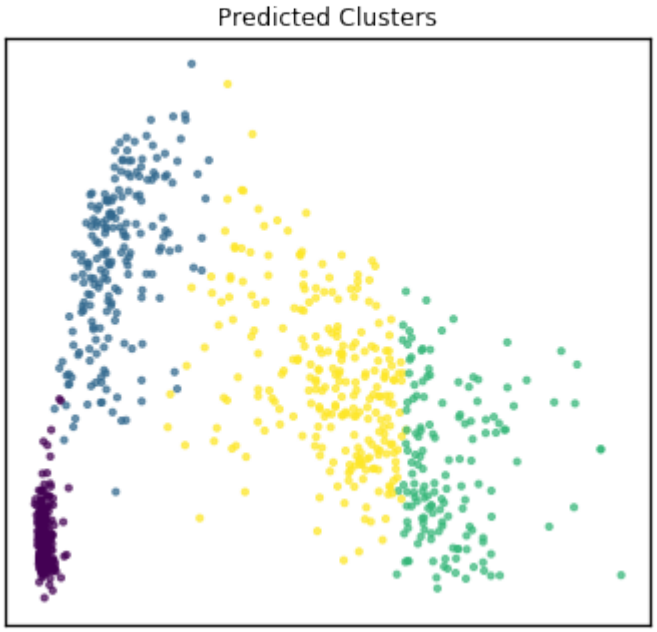
```
Multi-view homogeneity score: 0.962
```



To compare to single-view methods, we concatenate the 6 views we used for co-clustering into one data matrix, and then perform spectral clustering using the *scikit-learn* library. From the figure and cluster scores that are produced, we can see that single-view spectral clustering is unable to perform as well as the multi-view version.

[7]:
```
from sklearn.cluster import SpectralClustering

# Concatenate views and cluster
X_clustering = X_viewing
clust = SpectralClustering(n_clusters=4, affinity='nearest_neighbors')
sklearn_cluster_labels = clust.fit_predict(X_clustering)

# Test the accuracy of the clustering
sk_score = homogeneity_score(y, sklearn_cluster_labels)
print('Single-view homogeneity score: {0:.3f}'.format(sk_score))

# Rearrange for easier visual comparison to true label plot
sklearn_cluster_labels = rearrange_labels(y, sklearn_cluster_labels)

# Use PCA for dimensionality reduction on the naive single view
pca = PCA(n_components=2)
pca_X = pca.fit_transform(X_viewing)

plt.figure(figsize=(5, 5))
plt.scatter(pca_X[:,0], pca_X[:,1], c=sklearn_cluster_labels, **sca_kwargs)
plt.xticks([], [])
plt.yticks([], [])
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.title("Predicted Clusters")
plt.show()
```

```
Single-view homogeneity score: 0.703
```



5.2.2 Semi-Supervised

The following tutorials demonstrate how effectiveness of cotraining in certain multiview scenarios to boost accuracy over single view methods.

Co-Training 2-View Semi-Supervised Classification

[1]:
```
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier

from mvlearn.semi_supervised import CTClassifier
from mvlearn.datasets import load_UCImultifeature
```

Load the UCI Multiple Digit Features Dataset as an Example for Semi-Supervised Learning

To simulate a semi-supervised learning scenario, randomly remove 98% of the labels.

[2]:
```
data, labels = load_UCImultifeature(select_labeled=[0,1])

# Use only the first 2 views as an example
View0, View1 = data[0], data[1]

# Split both views into testing and training
View0_train, View0_test, labels_train, labels_test = train_test_split(View0, labels,␣
↪test_size=0.33, random_state=42)
View1_train, View1_test, labels_train, labels_test = train_test_split(View1, labels,␣
↪test_size=0.33, random_state=42)
```

(continues on next page)

(continued from previous page)

```
# Randomly remove all but 4 of the labels
np.random.seed(6)
remove_idx = np.random.rand(len(labels_train),) < 0.98
labels_train[remove_idx] = np.nan
not_removed = np.where(remove_idx==False)
print("Remaining labeled sample labels: " + str(labels_train[not_removed]))
```

```
Remaining labeled sample labels: [1. 0. 1. 0.]
```

Co-Training on 2 Views vs. Single View Semi-Supervised Learning

Here, we use the default co-training classifier, which uses Gaussian naive bayes classifiers for both views. We compare its performance to the single-view semi-supervised setting with the same basic classifiers, and with the naive technique of concatenating the two views and performing single view learning.

In this case, concatenating the two views does not improve the performance over the better view. Multiview cotraining outperforms them all.

[3]:
```
############## Single view semi-supervised learning ##############
#-----------------------------------------------------------------
gnb0 = GaussianNB()
gnb1 = GaussianNB()
gnb2 = GaussianNB()

# Train on only the examples with labels
gnb0.fit(View0_train[not_removed,:].squeeze(), labels_train[not_removed])
y_pred0 = gnb0.predict(View0_test)
gnb1.fit(View1_train[not_removed,:].squeeze(), labels_train[not_removed])
y_pred1 = gnb1.predict(View1_test)
# Concatenate the 2 views for naive "multiview" learning
View01_train = np.hstack((View0_train[not_removed,:].squeeze(), View1_train[not_
↪removed,:].squeeze()))
View01_test = np.hstack((View0_test, View1_test))
gnb2.fit(View01_train, labels_train[not_removed])
y_pred2 = gnb2.predict(View01_test)

print("Single View Accuracy on First View: {0:.3f}\n".format(accuracy_score(labels_
↪test, y_pred0)))
print("Single View Accuracy on Second View: {0:.3f}\n".format(accuracy_score(labels_
↪test, y_pred1)))
print("Naive Concatenated View Accuracy: {0:.3f}\n".format(accuracy_score(labels_test,
↪ y_pred2)))


######### Multi-view co-training semi-supervised learning #########
#-----------------------------------------------------------------
# Train a CTClassifier on all the labeled and unlabeled training data
ctc = CTClassifier()
ctc.fit([View0_train, View1_train], labels_train)
y_pred_ct = ctc.predict([View0_test, View1_test])

print("Co-Training Accuracy on 2 Views: {0:.3f}".format(accuracy_score(labels_test, y_
↪pred_ct)))
```

```
Single View Accuracy on First View: 0.568

Single View Accuracy on Second View: 0.591

Naive Concatenated View Accuracy: 0.591

Co-Training Accuracy on 2 Views: 0.992
```

Select Different Base Classifiers for the Views and Change the CTClassifier fit() parameters

Now, we use a random forest classifier with different attributes for each view. Furthermore, we manually select the number of positive (p) and negative (n) examples chosen each round in the co-training process, and we define the unlabeled pool size to draw them from and the number of iterations of training to perform.

In this case, concatenating the two views outperforms single view methods, but multiview cotraining still performs the best.

[4]:
```
############## Single view semi-supervised learning ##############
#-----------------------------------------------------------------
rfc0 = RandomForestClassifier(n_estimators=100, bootstrap=True)
rfc1 = RandomForestClassifier(n_estimators=6, bootstrap=False)
rfc2 = RandomForestClassifier(n_estimators=100, bootstrap=False)

# Train on only the examples with labels
rfc0.fit(View0_train[not_removed,:].squeeze(), labels_train[not_removed])
y_pred0 = rfc0.predict(View0_test)
rfc1.fit(View1_train[not_removed,:].squeeze(), labels_train[not_removed])
y_pred1 = rfc1.predict(View1_test)
# Concatenate the 2 views for naive "multiview" learning
View01_train = np.hstack((View0_train[not_removed,:].squeeze(), View1_train[not_
↪removed,:].squeeze()))
View01_test = np.hstack((View0_test, View1_test))
rfc2.fit(View01_train, labels_train[not_removed])
y_pred2 = rfc2.predict(View01_test)

print("Single View Accuracy on First View: {0:.3f}\n".format(accuracy_score(labels_
↪test, y_pred0)))
print("Single View Accuracy on Second View: {0:.3f}\n".format(accuracy_score(labels_
↪test, y_pred1)))
print("Naive Concatenated View Accuracy: {0:.3f}\n".format(accuracy_score(labels_test,
↪ y_pred2)))

######### Multi-view co-training semi-supervised learning #########
#-----------------------------------------------------------------
rfc0 = RandomForestClassifier(n_estimators=100, bootstrap=True)
rfc1 = RandomForestClassifier(n_estimators=6, bootstrap=False)
ctc = CTClassifier(rfc0, rfc1, p=2, n=2, unlabeled_pool_size=20, num_iter=100)
ctc.fit([View0_train, View1_train], labels_train)
y_pred_ct = ctc.predict([View0_test, View1_test])

print("Co-Training Accuracy: {0:.3f}".format(accuracy_score(labels_test, y_pred_ct)))
```

```
Single View Accuracy on First View: 0.902

Single View Accuracy on Second View: 0.871
```

(continues on next page)

(continued from previous page)

```
Naive Concatenated View Accuracy: 0.977

Co-Training Accuracy: 0.992
```

Get the prediction probabilities for all the examples

[5]:
```
y_pred_proba = ctc.predict_proba([View0_test, View1_test])
print("Full y_proba shape = " + str(y_pred_proba.shape))
print("\nFirst 10 class probabilities:\n")
print(y_pred_proba[:10,:])
```

```
Full y_proba shape = (132, 2)

First 10 class probabilities:

[[1.         0.        ]
 [0.945      0.055     ]
 [0.005      0.995     ]
 [0.09       0.91      ]
 [0.16833333 0.83166667]
 [0.995      0.005     ]
 [0.955      0.045     ]
 [0.955      0.045     ]
 [0.28       0.72      ]
 [0.925      0.075     ]]
```

Cotraining classification performance in simulated multiview scenarios

- **Experimental Setup**
- **Performance when one view is totally redundant**
- **Performance when one view is inseparable**
- **Performance when labeled data is excellent**
- **Performance when labeled data is not very separable**
- **Performance when data is overlapping**
- **Performance as labeled data proportion (essentially sample size) is varied**

[43]:
```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib

import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier
```

(continues on next page)

(continued from previous page)

```
from sklearn.decomposition import PCA

from mvlearn.semi_supervised import CTClassifier
from mvlearn.datasets import load_UCImultifeature

from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
```

Function to create 2 class data

This function is used to generate examples for 2 classes from multivariate normal distributions. Once the examples are generated, it splits them into training and testing sets and returns the needed information

[44]:
```
def create_data(seed, class2_mean_center, view1_var, view2_var, N_per_class, view2_
↪class2_mean_center=1):

    np.random.seed(seed)

    view1_mu0 = np.zeros(2,)
    view1_mu1 = class2_mean_center  * np.ones(2,) #
    view1_cov = view1_var*np.eye(2)

    view2_mu0 = np.zeros(2,)
    view2_mu1 = view2_class2_mean_center * np.ones(2,)
    view2_cov = view2_var*np.eye(2)

    view1_class0 = np.random.multivariate_normal(view1_mu0, view1_cov, size=N_per_
↪class)
    view1_class1 = np.random.multivariate_normal(view1_mu1, view1_cov, size=N_per_
↪class)

    view2_class0 = np.random.multivariate_normal(view2_mu0, view2_cov, size=N_per_
↪class)
    view2_class1 = np.random.multivariate_normal(view2_mu1, view2_cov, size=N_per_
↪class)

    View1 = np.concatenate((view1_class0, view1_class1))
    View2 = np.concatenate((view2_class0, view2_class1))
    Labels = np.concatenate((np.zeros(N_per_class,), np.ones(N_per_class,)))

    # Split both views into testing and training
    View1_train, View1_test, labels_train_full, labels_test_full = train_test_
↪split(View1, Labels, test_size=0.3, random_state=42)
    View2_train, View2_test, labels_train_full, labels_test_full = train_test_
↪split(View2, Labels, test_size=0.3, random_state=42)

    labels_train = labels_train_full.copy()
    labels_test = labels_test_full.copy()

    return View1_train, View2_train, labels_train, labels_train.copy(), View1_test,␣
↪View2_test, labels_test
```

Function to do predictions on single or concatenated view data

This function is used create classifiers for single or concatenated views and return their predictions.

[45]:
```
def single_view_class(v1_train, labels_train, v1_test, labels_test, v2_train, v2_test,
↪ v2_solver, v2_penalty):

    gnb0 = LogisticRegression()
    gnb1 = LogisticRegression(solver=v2_solver, penalty=v2_penalty)
    gnb2 = LogisticRegression()

    # Train on only the examples with labels
    gnb0.fit(v1_train, labels_train)
    y_pred0 = gnb0.predict(v1_test)

    gnb1.fit(v2_train, labels_train)
    y_pred1 = gnb1.predict(v2_test)

    accuracy_view1 = (accuracy_score(labels_test, y_pred0))
    accuracy_view2 = (accuracy_score(labels_test, y_pred1))

    # Concatenate views in naive way and train model
    combined_labeled = np.hstack((v1_train, v2_train))
    combined_test = np.hstack((v1_test, v2_test))

    gnb2.fit(combined_labeled, labels_train)
    y_pred2 = gnb2.predict(combined_test)

    accuracy_combined = (accuracy_score(labels_test, y_pred2))

    return accuracy_view1, accuracy_view2, accuracy_combined
```

Function to create 2 class scatter plots with labeled data shown

This function is used to create scatter plots of the 2 class data as well as show the samples that are labeled, making it easier to understand what distributions the simulations are dealing with

[46]:
```
def scatterplot_classes(not_removed, labels_train, labels_train_full, View1_train,␣
↪View2_train):

    idx_train_0 = np.where(labels_train_full==0)
    idx_train_1 = np.where(labels_train_full==1)

    labeled_idx_class0 = not_removed[np.where(labels_train[not_removed]==0)]
    labeled_idx_class1 = not_removed[np.where(labels_train[not_removed]==1)]

    # plot the views
    plt.figure()
    fig, ax = plt.subplots(1,2, figsize=(14,5))

    ax[0].scatter(View1_train[idx_train_0,0], View1_train[idx_train_0,1])
    ax[0].scatter(View1_train[idx_train_1,0], View1_train[idx_train_1,1])
    ax[0].scatter(View1_train[labeled_idx_class0,0], View1_train[labeled_idx_class0,
↪1], s=300, marker='X')
    ax[0].scatter(View1_train[labeled_idx_class1,0], View1_train[labeled_idx_class1,
↪1], s=300, marker='X')
```

(continues on next page)

(continued from previous page)

```
    ax[0].set_title('One Randomization of View 1')
    ax[0].legend(('Class 0', 'Class 1', 'Labeled Class 0', 'Labeled Class 1'))
    ax[0].axes.get_xaxis().set_visible(False)
    ax[0].axes.get_yaxis().set_visible(False)

    ax[1].scatter(View2_train[idx_train_0,0], View2_train[idx_train_0,1])
    ax[1].scatter(View2_train[idx_train_1,0], View2_train[idx_train_1,1])
    ax[1].scatter(View2_train[labeled_idx_class0,0], View1_train[labeled_idx_class0,
↪1], s=300, marker='X')
    ax[1].scatter(View2_train[labeled_idx_class1,0], View1_train[labeled_idx_class1,
↪1], s=300, marker='X')
    ax[1].set_title('One Randomization of View 2')
    ax[1].legend(('Class 0', 'Class 1', 'Labeled Class 0', 'Labeled Class 1'))
    ax[1].axes.get_xaxis().set_visible(False)
    ax[1].axes.get_yaxis().set_visible(False)

    plt.show()
```

Performance on simulated data

General Experimental Setup

- Below are the results from simulated data testing of the cotraining classifier with different classification problems (class distributions)
- Results are averaged over 20 randomizations, where a single randomization means using a new seed to generate examples from 2 class distributions and then randomly selecting about 1% of the training data as labeled and leaving the rest unlabeled
- 500 examples per class, with 70% used for training and 30% for testing
- For a randomization, train 4 classifiers
    1. Classifier trained on view 1 labeled data only
    2. Classifier trained on view 2 labeled data only
    3. Classifier trained on concatenation of labeled features from views 1 and 2
    4. multivew CTClassifier trained on views 1 and 2
        - For this, test classification accuracy after different numbers of cotraining iterations to see trajectory of classification accuracy
- Classification Method:
    - Logistic Regression
        * 'l2' penalty for view 1 and 'l1' penalty for view 2 to ensure independence between the classifiers in the views. This is important because a key aspect of cotraining is view independence, which can either be enforced by completely independent data, or by using an independent classifier for each view, such as using different parameters with the same type of classifier, or two different classification algorithms.

Performance when classes are well separated and labeled examples are randomly chosen

Here, the 2 class distributions are the following - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [1, 1] - Class 1 covariance: .2*eye(2)

Labeled examples are chosen randomly from the training set

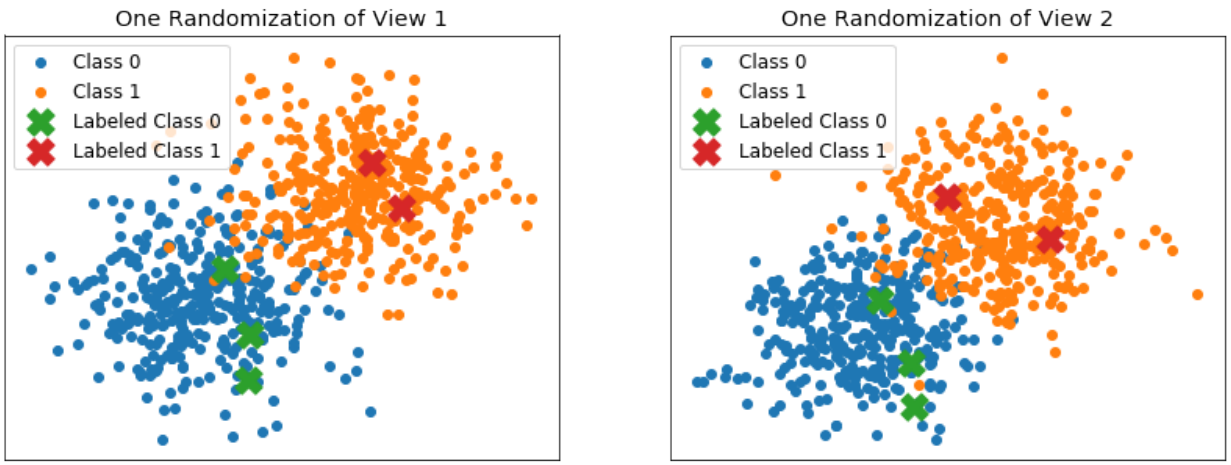
[47]:
```
randomizations = 20
N_per_class = 500
view2_penalty = 'l1'
view2_solver = 'liblinear'

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):

        ######################### Create Data ############################
        View1_train, View2_train, labels_train, labels_train_full, View1_test, View2_
↪test, labels_test = create_data(seed, 1, .2, .2, N_per_class)

        # randomly remove some labels
        np.random.seed(11)
        remove_idx = np.random.rand(len(labels_train),) < .99
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:
            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning #############
        # Only do this calculation once, since not affected by number of iterations
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
                                                                                    ␣
↪labels_train[not_removed],
                                                                                    ␣
↪View1_test,
                                                                                    ␣
↪labels_test,
                                                                                    ␣
↪View2_train[not_removed,:].squeeze(),
                                                                                    ␣
↪View2_test,
                                                                                    ␣
↪view2_solver,
                                                                                    ␣
↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
```

(continues on next page)

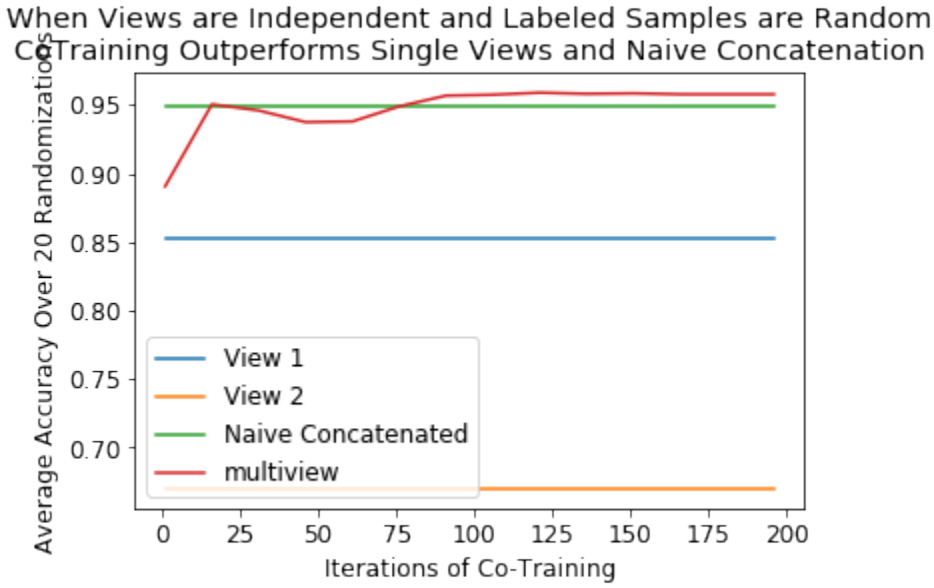
(continued from previous page)

```
            acc_combined.append(accuracy_combined)

        ################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))

acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```



[48]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
plt.plot(N_iters, acc_ct)
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When Views are Independent and Labeled Samples are Random\nCoTraining␣
↪Outperforms Single Views and Naive Concatenation')
plt.show()
```



Performance when one view is totally redundant

Here, the 2 class distributions are the following - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [1, 1] - Class 1 covariance: .2*eye(2)

Views 1 and 2 hold the exact same samples

Labeled examples are chosen randomly from the training set

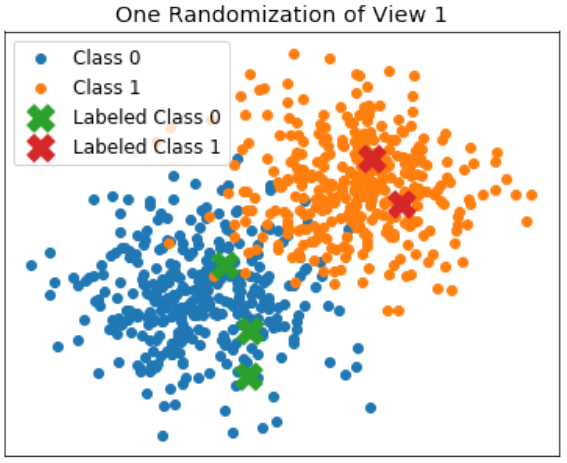
[49]:
```
randomizations = 20
N_per_class = 500
view2_penalty = 'l1'
view2_solver = 'liblinear'

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):

        ######################### Create Data ############################
        View1_train, View2_train, labels_train, labels_train_full, View1_test, View2_
↪test, labels_test = create_data(seed, 1, .2, .2, N_per_class)

        View2_train = View1_train.copy()
        View2_test = View1_test.copy()

        # randomly remove some labels
        np.random.seed(11)
```

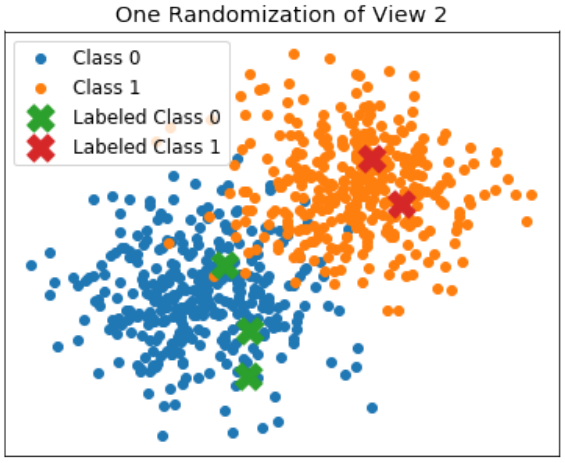
(continues on next page)

(continued from previous page)

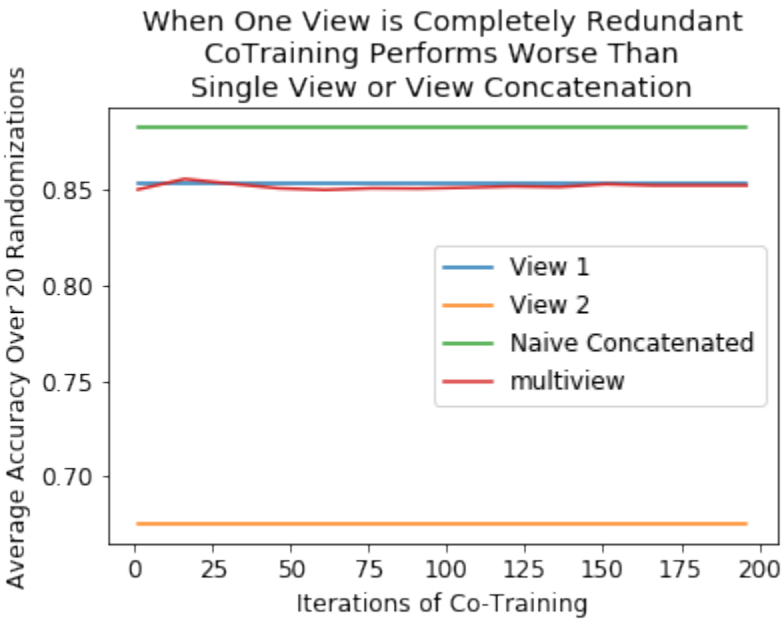
```
        remove_idx = np.random.rand(len(labels_train),) < .99
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:
            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning #############
        # Only do this calculation once, since not affected by number of iterations
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
                                                                                    ␣
↪labels_train[not_removed],
                                                                                    ␣
↪View1_test,
                                                                                    ␣
↪labels_test,
                                                                                    ␣
↪View2_train[not_removed,:].squeeze(),
                                                                                    ␣
↪View2_test,
                                                                                    ␣
↪view2_solver,
                                                                                    ␣
↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
            acc_combined.append(accuracy_combined)

        ##################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))

acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```





[50]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
plt.plot(N_iters, acc_ct)
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When One View is Completely Redundant\nCoTraining Performs Worse␣
↪Than\nSingle View or View Concatenation')
plt.show()
```



Performance when one view is inseparable

Here, the 2 class distributions are the following for the first view - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [1, 1] - Class 1 covariance: .2*eye(2)

For the second view: - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [0, 0] - Class 1 covariance: .2*eye(2)

Labeled examples are chosen randomly from the training set

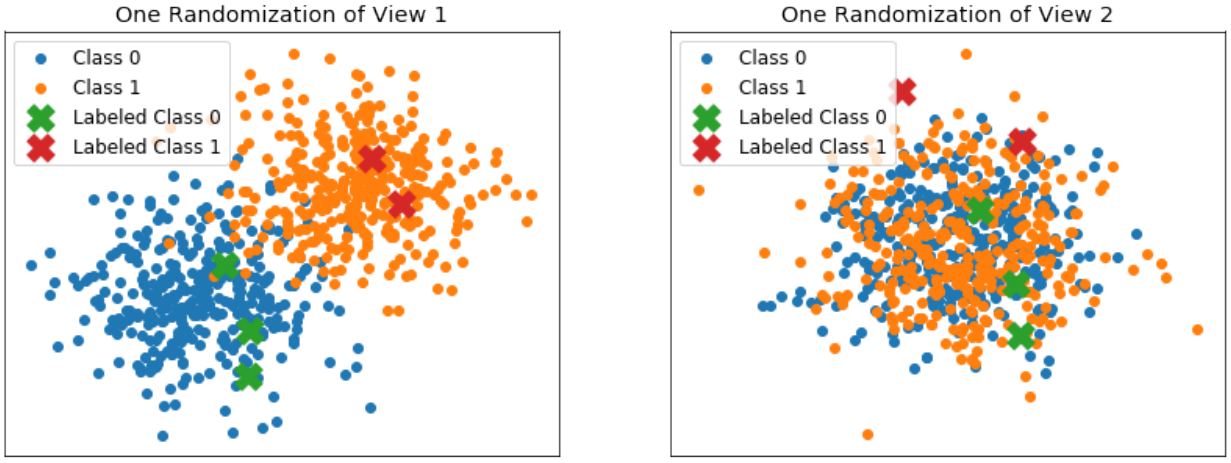
[51]:
```
randomizations = 20
N_per_class = 500
view2_penalty = 'l1'
view2_solver = 'liblinear'

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):

        ######################### Create Data ############################
        View1_train, View2_train, labels_train, labels_train_full, View1_test, View2_
↪test, labels_test = create_data(seed,
↪                                1,
↪                                .2,
↪                                .2,
↪                                N_per_class,
↪                                view2_class2_mean_center=0)

        # randomly remove some labels
        np.random.seed(11)
        remove_idx = np.random.rand(len(labels_train),) < .99
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:
            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning ##############
        # Only do this calculation once, since not affected by number of iterations
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
```

(continues on next page)

(continued from previous page)

```
↪labels_train[not_removed],

↪View1_test,

↪labels_test,

↪View2_train[not_removed,:].squeeze(),

↪View2_test,

↪view2_solver,

↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
            acc_combined.append(accuracy_combined)

        ##################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))

acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```

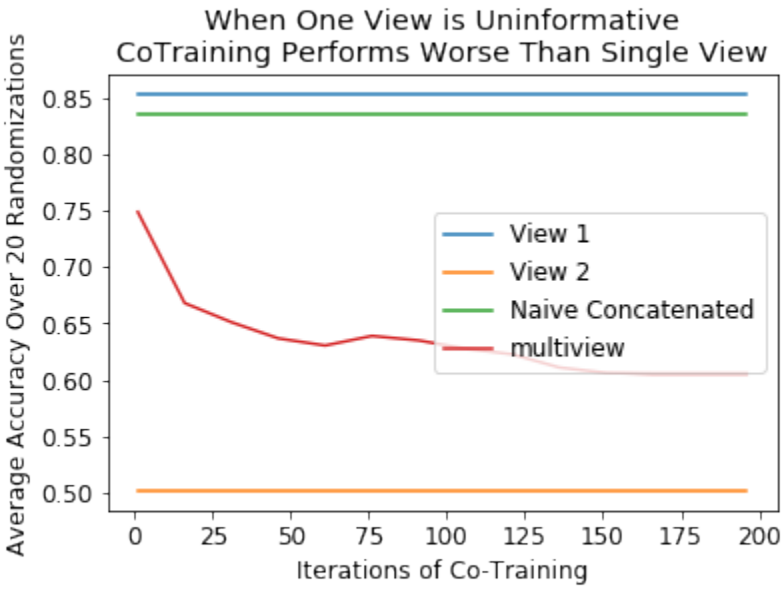


[52]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
```

(continues on next page)

(continued from previous page)

```
plt.plot(N_iters, acc_ct)
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When One View is Uninformative\nCoTraining Performs Worse Than Single View
↪')
plt.show()
```



Performance when labeled data is excellent

Here, the 2 class distributions are the following - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [1, 1] - Class 1 covariance: .2*eye(2)

Labeled examples are chosen to be very close to the mean of their respective class - Normally distributed around their class mean with standard deviation 0.05 in both dimensions

[53]:
```
randomizations = 20
N_per_class = 500
num_perfect = 3
perfect_scale = 0.05
view2_penalty = 'l1'
view2_solver = 'liblinear'

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):
```

(continues on next page)

(continued from previous page)

```
        ######################## Create Data ############################
        np.random.seed(seed)

        view1_mu0 = np.zeros(2,)
        view1_mu1 = np.ones(2,)
        view1_cov = .2*np.eye(2)

        view2_mu0 = np.zeros(2,)
        view2_mu1 = np.ones(2,)
        view2_cov = .2*np.eye(2)

        # generage perfect examples
        perfect_class0_v1 = view1_mu0 + np.random.normal(loc=0, scale=perfect_scale,␣
↪size=view1_mu0.shape)
        perfect_class0_v2 = view1_mu0 + np.random.normal(loc=0, scale=perfect_scale,␣
↪size=view1_mu0.shape)
        perfect_class1_v1 = view1_mu1 + np.random.normal(loc=0, scale=perfect_scale,␣
↪size=view1_mu1.shape)
        perfect_class1_v2 = view1_mu1 + np.random.normal(loc=0, scale=perfect_scale,␣
↪size=view1_mu1.shape)
        for p in range(1, num_perfect):
            perfect_class0_v1 = np.vstack((perfect_class0_v1, view1_mu0 + np.random.
↪normal(loc=0, scale=0.01, size=view1_mu0.shape)))
            perfect_class0_v2 = np.vstack((perfect_class0_v2, view1_mu0 + np.random.
↪normal(loc=0, scale=0.01, size=view1_mu0.shape)))
            perfect_class1_v1 = np.vstack((perfect_class1_v1, view1_mu1 + np.random.
↪normal(loc=0, scale=0.01, size=view1_mu1.shape)))
            perfect_class1_v2 = np.vstack((perfect_class1_v2, view1_mu1 + np.random.
↪normal(loc=0, scale=0.01, size=view1_mu1.shape)))
        perfect_labels = np.zeros(num_perfect,)
        perfect_labels = np.concatenate((perfect_labels, np.ones(num_perfect,)))


        view1_class0 = np.random.multivariate_normal(view1_mu0, view1_cov, size=N_per_
↪class)
        view1_class1 = np.random.multivariate_normal(view1_mu1, view1_cov, size=N_per_
↪class)

        view2_class0 = np.random.multivariate_normal(view2_mu0, view2_cov, size=N_per_
↪class)
        view2_class1 = np.random.multivariate_normal(view2_mu1, view2_cov, size=N_per_
↪class)

        View1 = np.concatenate((view1_class0, view1_class1))
        View2 = np.concatenate((view2_class0, view2_class1))
        Labels = np.concatenate((np.zeros(N_per_class,), np.ones(N_per_class,)))


        # Split both views into testing and training
        View1_train, View1_test, labels_train_full, labels_test_full = train_test_
↪split(View1, Labels, test_size=0.3, random_state=42)
        View2_train, View2_test, labels_train_full, labels_test_full = train_test_
↪split(View2, Labels, test_size=0.3, random_state=42)

        labels_train = labels_train_full.copy()
        labels_test = labels_test_full.copy()
```

(continues on next page)

(continued from previous page)

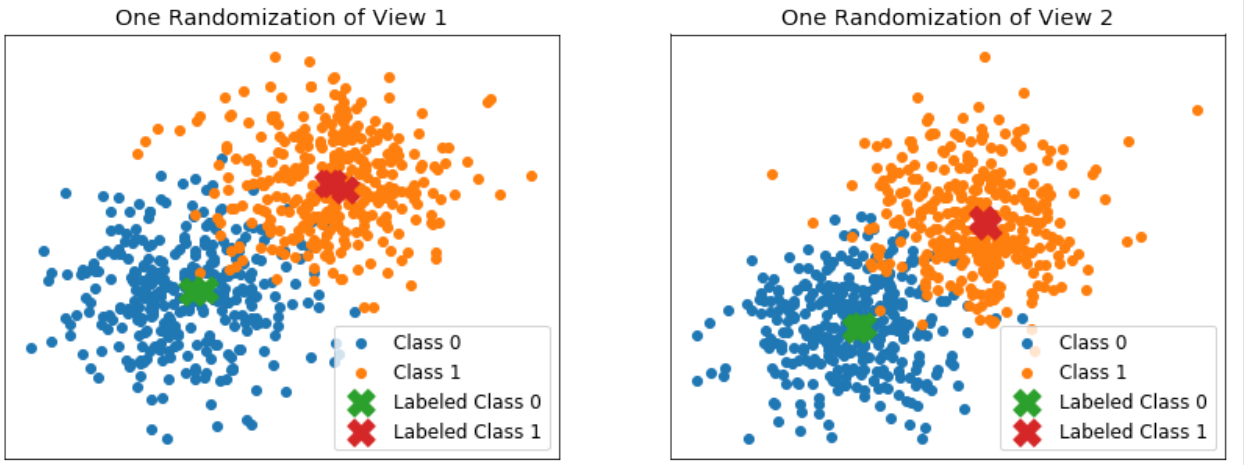
```
        # Add the perfect examples
        View1_train = np.vstack((View1_train, perfect_class0_v1, perfect_class1_v1))
        View2_train = np.vstack((View2_train, perfect_class0_v2, perfect_class1_v2))
        labels_train = np.concatenate((labels_train, perfect_labels))

        # randomly remove all but perfect labeled samples
        remove_idx = [True for i in range(len(labels_train)-2*num_perfect)]
        for i in range(2*num_perfect):
            remove_idx.append(False)

        #remove_idx = [False if i < (len(labels_train)-2*num_perfect) else True for i␣
↪in range(len(labels_train))]
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]
        not_removed = np.arange(len(labels_train)-2*num_perfect, len(labels_train))

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:
            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning #############
        # Only once, since not affected by "num iters"
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
                                                                                ␣
↪labels_train[not_removed],
                                                                                ␣
↪View1_test,
                                                                                ␣
↪labels_test,
                                                                                ␣
↪View2_train[not_removed,:].squeeze(),
                                                                                ␣
↪View2_test,
                                                                                ␣
↪view2_solver,
                                                                                ␣
↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
            acc_combined.append(accuracy_combined)


        ##################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))
```

(continues on next page)

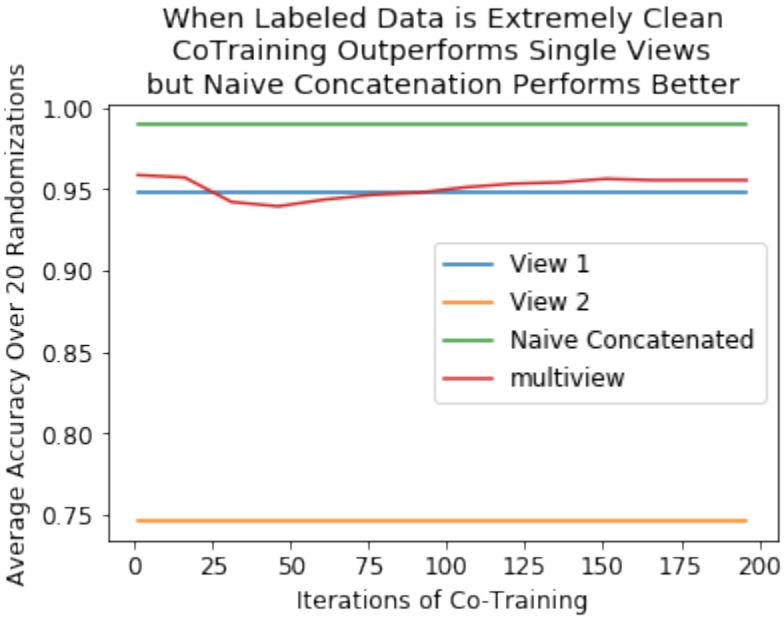
(continued from previous page)

```
acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```



[54]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
plt.plot(N_iters, acc_ct)
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When Labeled Data is Extremely Clean\nCoTraining Outperforms Single␣
↪Views\nbut Naive Concatenation Performs Better')
plt.show()
```



Performance when labeled data is not very separable

Here, the 2 class distributions are the following - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [1, 1] - Class 1 covariance: .2*eye(2)

Labeled examples are chosen to be far from their respective means according to a uniform distribution in 2 dimensions between .2 and .75 away from the x1 or x2 coordinate of the mean

[55]:
```
randomizations = 20
N_per_class = 500
num_perfect = 2
uniform_min = 0.2
uniform_max = 0.75
view2_penalty = 'l1'
view2_solver = 'liblinear'

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):

        ######################### Create Data ############################
        np.random.seed(seed)

        view1_mu0 = np.zeros(2,)
        view1_mu1 = np.ones(2,)
        view1_cov = .2*np.eye(2)
```

(continues on next page)

(continued from previous page)

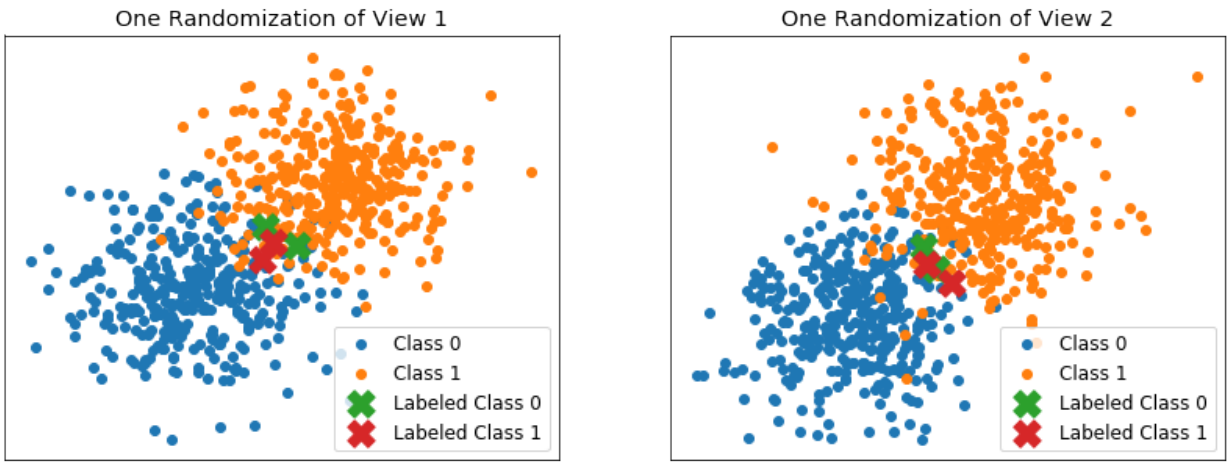
```
        view2_mu0 = np.zeros(2,)
        view2_mu1 = np.ones(2,)
        view2_cov = .2*np.eye(2)

        # generage bad examples
        perfect_class0_v1 = view1_mu0 + np.random.uniform(uniform_min, uniform_max,␣
↪size=view1_mu0.shape)
        perfect_class0_v2 = view1_mu0 + np.random.uniform(uniform_min, uniform_max,␣
↪size=view1_mu0.shape)
        perfect_class1_v1 = view1_mu1 - np.random.uniform(uniform_min, uniform_max,␣
↪size=view1_mu0.shape)
        perfect_class1_v2 = view1_mu1 - np.random.uniform(uniform_min, uniform_max,␣
↪size=view1_mu0.shape)
        for p in range(1, num_perfect):
            perfect_class0_v1 = np.vstack((perfect_class0_v1, view1_mu0 + np.random.
↪uniform(uniform_min, uniform_max, size=view1_mu0.shape)))
            perfect_class0_v2 = np.vstack((perfect_class0_v2, view1_mu0 + np.random.
↪uniform(uniform_min, uniform_max, size=view1_mu0.shape)))
            perfect_class1_v1 = np.vstack((perfect_class1_v1, view1_mu1 - np.random.
↪uniform(uniform_min, uniform_max, size=view1_mu0.shape)))
            perfect_class1_v2 = np.vstack((perfect_class1_v2, view1_mu1 - np.random.
↪uniform(uniform_min, uniform_max, size=view1_mu0.shape)))
        perfect_labels = np.zeros(num_perfect,)
        perfect_labels = np.concatenate((perfect_labels, np.ones(num_perfect,)))

        view1_class0 = np.random.multivariate_normal(view1_mu0, view1_cov, size=N_per_
↪class)
        view1_class1 = np.random.multivariate_normal(view1_mu1, view1_cov, size=N_per_
↪class)

        view2_class0 = np.random.multivariate_normal(view2_mu0, view2_cov, size=N_per_
↪class)
        view2_class1 = np.random.multivariate_normal(view2_mu1, view2_cov, size=N_per_
↪class)

        View1 = np.concatenate((view1_class0, view1_class1))
        View2 = np.concatenate((view2_class0, view2_class1))
        Labels = np.concatenate((np.zeros(N_per_class,), np.ones(N_per_class,)))


        # Split both views into testing and training
        View1_train, View1_test, labels_train_full, labels_test_full = train_test_
↪split(View1, Labels, test_size=0.3, random_state=42)
        View2_train, View2_test, labels_train_full, labels_test_full = train_test_
↪split(View2, Labels, test_size=0.3, random_state=42)

        labels_train = labels_train_full.copy()
        labels_test = labels_test_full.copy()


        # Add the perfect examples
        View1_train = np.vstack((View1_train, perfect_class0_v1, perfect_class1_v1))
        View2_train = np.vstack((View2_train, perfect_class0_v2, perfect_class1_v2))
        labels_train = np.concatenate((labels_train, perfect_labels))

        # randomly remove all but perfect labeled samples
```

(continues on next page)

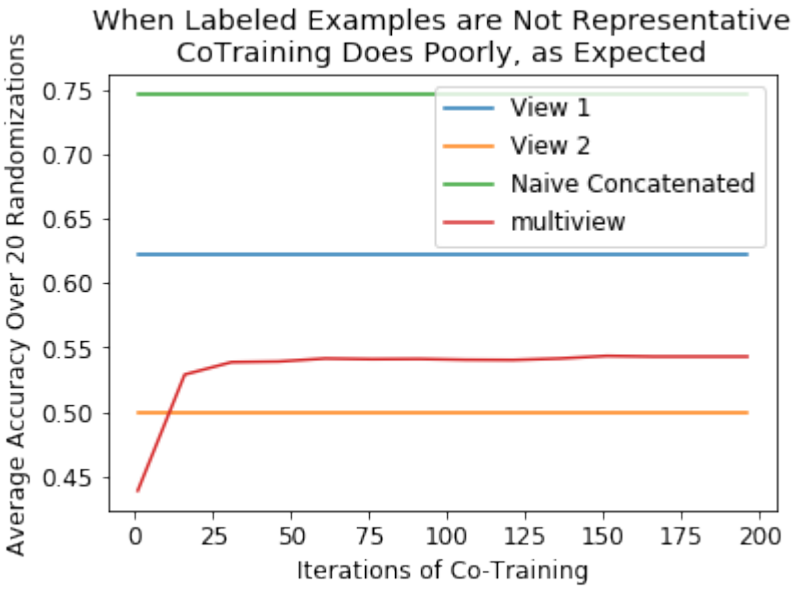
(continued from previous page)

```
        remove_idx = [True for i in range(len(labels_train)-2*num_perfect)]
        for i in range(2*num_perfect):
            remove_idx.append(False)


        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]
        not_removed = np.arange(len(labels_train)-2*num_perfect, len(labels_train))

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:

            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning #############
        # Only once, since not affected by "num iters"
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
                                                                                    ␣
↪labels_train[not_removed],
                                                                                    ␣
↪View1_test,
                                                                                    ␣
↪labels_test,
                                                                                    ␣
↪View2_train[not_removed,:].squeeze(),
                                                                                    ␣
↪View2_test,
                                                                                    ␣
↪view2_solver,
                                                                                    ␣
↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
            acc_combined.append(accuracy_combined)


        ##################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))


acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```



[56]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
plt.plot(N_iters, acc_ct)
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When Labeled Examples are Not Representative\nCoTraining Does Poorly, as␣
↪Expected')
plt.show()
```



Performance when data is overlapping

Here, the 2 class distributions are the following - Class 0 mean: [0, 0] - Class 0 covariance: *.2eye(2) - Class 1 mean: [0, 0] - Class 1 covariance: .2*eye(2)

Labeled examples are chosen randomly from the training set

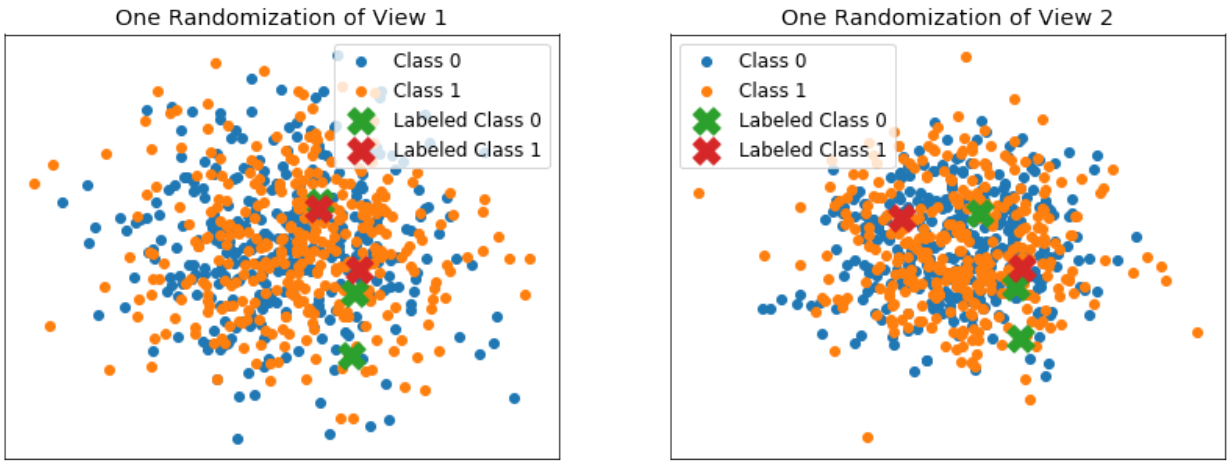
[57]:
```
randomizations = 20
N_per_class = 500
view2_penalty = 'l1'
view2_solver = 'liblinear'
class2_mean_center = 0 # 1 would make this identical to first test

N_iters = np.arange(1, 202, 15)
acc_ct = [[] for _ in N_iters]
acc_view1 = []
acc_view2 = []
acc_combined = []


for count, iters in enumerate(N_iters):

    for seed in range(randomizations):

        ######################### Create Data ############################
        View1_train, View2_train, labels_train, labels_train_full, View1_test, View2_
↪test, labels_test = create_data(seed,
                                                                                    ⌋
↪                                0,
                                                                                    ⌋
↪                                .2,
                                                                                    ⌋
↪                                .2,
                                                                                    ⌋
↪                                N_per_class,
                                                                                    ⌋
↪                                view2_class2_mean_center=class2_mean_center)

        # randomly remove some labels
        np.random.seed(11)
        remove_idx = np.random.rand(len(labels_train),) < .99
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]

        # make sure both classes have at least 1 labeled example
        if len(set(labels_train[not_removed])) != 2:
            continue

        if seed == 0 and count == 0:

            scatterplot_classes(not_removed, labels_train, labels_train_full, View1_
↪train, View2_train)

        ############## Single view semi-supervised learning #############
        # Only once, since not affected by "num iters"
        if count == 0:
            accuracy_view1, accuracy_view2, accuracy_combined = single_view_
↪class(View1_train[not_removed,:].squeeze(),
```

(continues on next page)

(continued from previous page)

```
↪labels_train[not_removed],

↪View1_test,

↪labels_test,

↪View2_train[not_removed,:].squeeze(),

↪View2_test,

↪view2_solver,

↪view2_penalty)

            acc_view1.append(accuracy_view1)
            acc_view2.append(accuracy_view2)
            acc_combined.append(accuracy_combined)

        ##################### Multiview ########################################
        gnb0 = LogisticRegression()
        gnb1 = LogisticRegression(solver=view2_solver, penalty=view2_penalty)
        ctc = CTClassifier(gnb0, gnb1, num_iter=iters)
        ctc.fit([View1_train, View2_train], labels_train)
        y_pred_ct = ctc.predict([View1_test, View2_test])
        acc_ct[count].append((accuracy_score(labels_test, y_pred_ct)))

acc_view1 = np.mean(acc_view1)
acc_view2 = np.mean(acc_view2)
acc_combined = np.mean(acc_combined)
acc_ct = [sum(row) / float(len(row)) for row in acc_ct]
```

```
<Figure size 432x288 with 0 Axes>
```

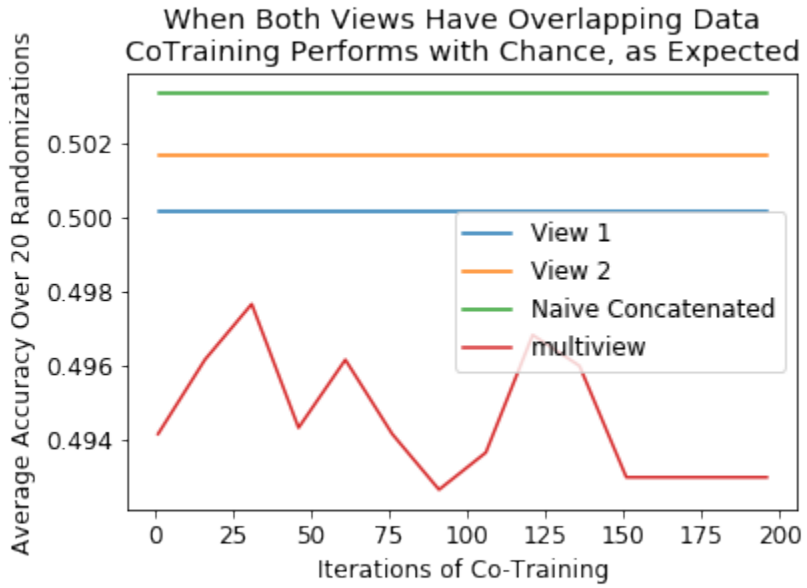


[58]:
```
# make a figure from the data
plt.figure()
plt.plot(N_iters, acc_view1*np.ones(N_iters.shape))
plt.plot(N_iters, acc_view2*np.ones(N_iters.shape))
plt.plot(N_iters, acc_combined*np.ones(N_iters.shape))
plt.plot(N_iters, acc_ct)
```

(continues on next page)

(continued from previous page)

```
plt.legend(('View 1', 'View 2', 'Naive Concatenated', 'multiview'))
plt.ylabel("Average Accuracy Over {} Randomizations".format(randomizations))
plt.xlabel('Iterations of Co-Training')
plt.title('When Both Views Have Overlapping Data\nCoTraining Performs with Chance, as␣
↪Expected')
plt.show()
```



Performance as labeled data proportion (essentially sample size) is varied

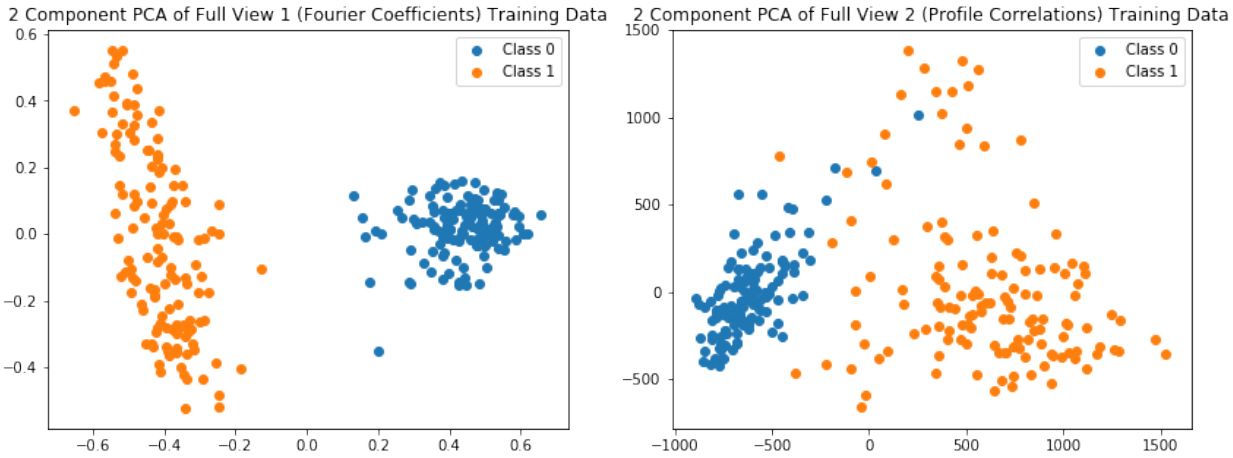
[16]:
```
data, labels = load_UCImultifeature(select_labeled=[0,1])

# Use only the first 2 views as an example
View0, View1 = data[0], data[1]

# Split both views into testing and training
View0_train, View0_test, labels_train_full, labels_test_full = train_test_split(View0,
↪ labels, test_size=0.33, random_state=42)
View1_train, View1_test, labels_train_full, labels_test_full = train_test_split(View1,
↪ labels, test_size=0.33, random_state=42)

# Do PCA to visualize data
pca = PCA(n_components = 2)
View0_pca = pca.fit_transform(View0_train)
View1_pca = pca.fit_transform(View1_train)

View0_pca_class0 = View0_pca[np.where(labels_train_full==0)[0],:]
View0_pca_class1 = View0_pca[np.where(labels_train_full==1)[0],:]
View1_pca_class0 = View1_pca[np.where(labels_train_full==0)[0],:]
View1_pca_class1 = View1_pca[np.where(labels_train_full==1)[0],:]


# plot the views
plt.figure()
```

(continues on next page)

(continued from previous page)

```
fig, ax = plt.subplots(1,2, figsize=(14,5))

ax[0].scatter(View0_pca_class0[:,0], View0_pca_class0[:,1])
ax[0].scatter(View0_pca_class1[:,0], View0_pca_class1[:,1])
ax[0].set_title('2 Component PCA of Full View 1 (Fourier Coefficients) Training Data')
ax[0].legend(('Class 0', 'Class 1'))

ax[1].scatter(View1_pca_class0[:,0], View1_pca_class0[:,1])
ax[1].scatter(View1_pca_class1[:,0], View1_pca_class1[:,1])
ax[1].set_title('2 Component PCA of Full View 2 (Profile Correlations) Training Data')
ax[1].legend(('Class 0', 'Class 1'))

plt.show()
```

```
<Figure size 432x288 with 0 Axes>
```



[23]:
```
N_labeled_full = []
acc_ct_full = []
acc_v0_full = []
acc_v1_full = []

iters = 500

for i, num in zip(np.linspace(0.03, .30, 20), (np.linspace(4, 30, 20)).astype(int)):

    N_labeled = []
    acc_ct = []
    acc_v0 = []
    acc_v1 = []

    View0_train, View0_test, labels_train_full, labels_test_full = train_test_
↪split(View0, labels, test_size=0.33, random_state=42)
    View1_train, View1_test, labels_train_full, labels_test_full = train_test_
↪split(View1, labels, test_size=0.33, random_state=42)

    for seed in range(iters):

        labels_train = labels_train_full.copy()
        labels_test = labels_test_full.copy()
```

(continues on next page)

(continued from previous page)

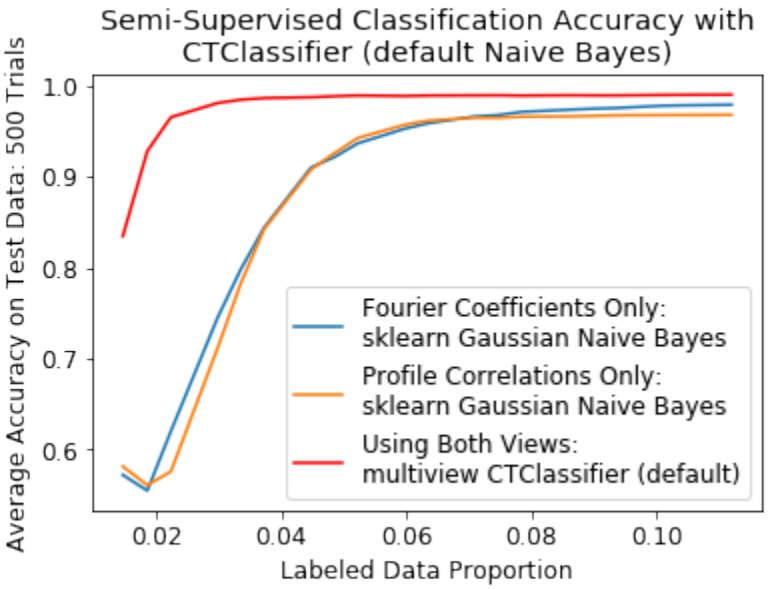
```
        # Randomly remove all but a small percentage of the labels
        np.random.seed(2*seed) #6
        remove_idx = np.random.rand(len(labels_train),) < 1-i
        labels_train[remove_idx] = np.nan
        not_removed = np.where(remove_idx==False)[0]
        not_removed = not_removed[:num]
        N_labeled.append(len(labels_train[not_removed])/len(labels_train))
        if len(set(labels_train[not_removed])) != 2:
            continue

        if Reverse_Labels:
            labels_one_idx = np.argwhere(labels_train == 1)
            labels_zero_idx = np.argwhere(labels_train == 0)

        ############## Single view semi-supervised learning #############
        #--------------------------------------------------------------------
        gnb0 = GaussianNB()
        gnb1 = GaussianNB()

        # Train on only the examples with labels
        gnb0.fit(View0_train[not_removed,:].squeeze(), labels_train[not_removed])

        y_pred0 = gnb0.predict(View0_test)
        gnb1.fit(View1_train[not_removed,:].squeeze(), labels_train[not_removed])
        y_pred1 = gnb1.predict(View1_test)

        acc_v0.append(accuracy_score(labels_test, y_pred0))
        acc_v1.append(accuracy_score(labels_test, y_pred1))

        ######### Multi-view co-training semi-supervised learning #########
        #--------------------------------------------------------------------
        # Train a CTClassifier on all the labeled and unlabeled training data
        ctc = CTClassifier()
        ctc.fit([View0_train, View1_train], labels_train)
        y_pred_ct = ctc.predict([View0_test, View1_test])
        acc_ct.append(accuracy_score(labels_test, y_pred_ct))

    acc_ct_full.append(np.mean(acc_ct))
    acc_v0_full.append(np.mean(acc_v0))
    acc_v1_full.append(np.mean(acc_v1))
    N_labeled_full.append(np.mean(N_labeled))
```

[28]:
```
matplotlib.rcParams.update({'font.size': 12})

plt.figure()
plt.plot(N_labeled_full, acc_v0_full)
plt.plot(N_labeled_full, acc_v1_full)
plt.plot(N_labeled_full, acc_ct_full,"r")
plt.legend(("Fourier Coefficients Only:\nsklearn Gaussian Naive Bayes", "Profile␣
↪Correlations Only:\nsklearn Gaussian Naive Bayes", "Using Both Views:\nmultiview␣
↪CTClassifier (default)"))
plt.title("Semi-Supervised Classification Accuracy with\nCTClassifier (default Naive␣
↪Bayes)")
plt.xlabel("Labeled Data Proportion")
plt.ylabel("Average Accuracy on Test Data: {} Trials".format(iters))
#plt.savefig('AvgAccuracy_CTClassifier.png', bbox_inches='tight')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Co-Training 2-View Semi-Supervised Regression

This tutorial demonstrates co-training regression on a semi-supervised regression task. The data only has targets for 20% of its samples, and although it does not have multiple views, co-training regression can still be beneficial. In order to get this benefit, the CTRegressor object is initialized with 2 different types of KNeighborsRegressors (in this case, the power parameter for the Minkowski metric is different in each view). Then, the single view of data (X) is passed in twice as if it shows two different views. The MSE of the predictions on test data from the resulting CTRegressor is compared to the MSE from using each of the individual KNeighborsRegressor objects after fitting on the labeled samples of the training data. The MSE shows that the CTRegressor does better than using either KNeighborsRegressor alone in this semi-supervised case.

[1]:
```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
from mpl_toolkits import mplot3d
%matplotlib inline
from mvlearn.semi_supervised import CTRegressor
```

Generating 3D Mexican Hat Data

[2]:
```
N_samples = 3750
N_test = 1250
labeled_portion = .2

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

(continues on next page)

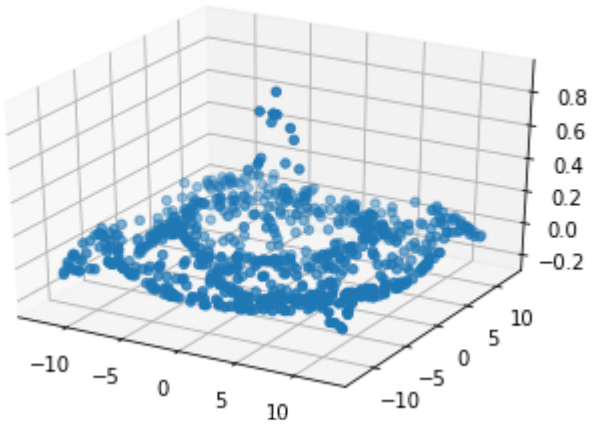
(continued from previous page)

```
# Generating the 3D Mexican Hat data
X = np.random.uniform(-4*np.pi, 4*np.pi, size=(N_samples,2))
y = ((np.sin(np.linalg.norm(X, axis=1)))/np.linalg.norm(X, axis=1)).squeeze()
X_test = np.random.uniform(-4*np.pi, 4*np.pi, size=(N_test,2))
y_test = ((np.sin(np.linalg.norm(X_test, axis=1)))/np.linalg.norm(X_test, axis=1)).
↪squeeze()

y_train = y.copy()
np.random.seed(1)

# Randomly selecting the index which are to be made nan
selector = np.random.uniform(size=(N_samples,))
selector[selector > labeled_portion] = np.nan
y_train[np.isnan(selector)] = np.nan
lab_samples = ~np.isnan(y_train)

# Indexes which are not null
not_null = [i for i in range(len(y_train)) if not np.isnan(y_train[i])]
```

Visualization of Data

Here, we plot the labeled samples that we have.

[3]:
```
fig = plt.figure()
ax = plt.axes(projection="3d")

z_points = y[lab_samples]
x_points = X[lab_samples, 0]
y_points = X[lab_samples, 1]
ax.scatter3D(x_points, y_points, z_points)
plt.show()
```



Co-Training on 2 views vs Single view training

Here, we are using the KNeighborsRegressor as the estimators for regression. We are using the default value for all the parameters except the p value in order to make the estimators independent. The same p values are used for training

the corresponding single view model.

[4]:
```
############## Single view semi-supervised learning ##############
#-----------------------------------------------------------------

knn1 = KNeighborsRegressor(p = 2)
knn2 = KNeighborsRegressor(p = 5)

# Train on only the examples with labels
knn1.fit(X[not_null], y[not_null])
pred1 = knn1.predict(X_test)
error1 = mean_squared_error(y_test, pred1)

knn2.fit(X[not_null], y[not_null])
pred2 = knn2.predict(X_test)
error2 = mean_squared_error(y_test, pred2)

print("MSE of single view with knn1 {}\n".format(error1))
print("MSE of single view with knn2 {}\n".format(error2))

######### Multi-view co-training semi-supervised learning #########
#-----------------------------------------------------------------

estimator1 = KNeighborsRegressor(p = 2)
estimator2 = KNeighborsRegressor(p = 5)
knn = CTRegressor(estimator1, estimator2, random_state = 26)

# Train a CTClassifier on all the labeled and unlabeled training data
knn.fit([X, X], y_train)
pred_multi_view = knn.predict([X_test, X_test])
error_multi_view = mean_squared_error(y_test, pred_multi_view)

print("MSE of cotraining semi supervised regression {}\n".format(error_multi_view))
```

```
MSE of single view with knn1 0.0016125954957153382

MSE of single view with knn2 0.001724891163476389

MSE of cotraining semi supervised regression 0.001508364708398609
```

[ ]:
```
```

5.2.3 Embedding

Inference on and visualization of multiview data often requires low-dimensional representations of the data, known as *embeddings*. Below are tutorials for computing such embeddings on multiview data.

Generalized Canonical Correlation Analysis (GCCA)

[23]:
```
from mvlearn.datasets import load_UCImultifeature
from mvlearn.embed import GCCA
from mvlearn.plotting import crossviews_plot
from graspy.plot import pairplot
```

(continues on next page)

(continued from previous page)

```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Load Data

We load three views from the UCI handwritten digits multi-view data set. Specificallym the Profile correlations, Karhunen-Love coefficients, and pixel averages from 2x3 windows.

[92]:
```
# Load full dataset, labels not needed
Xs, y = load_UCImultifeature()
Xs = [Xs[1], Xs[2], Xs[3]]
```

[93]:
```
# Check data
print(f'There are {len(Xs)} views.')
print(f'There are {Xs[0].shape[0]} observations')
print(f'The feature sizes are: {[X.shape[1] for X in Xs]}')
```

```
There are 3 views.
There are 2000 observations
The feature sizes are: [216, 64, 240]
```

Embed Views

[94]:
```
# Create GCCA object and embed the
gcca = GCCA()
Xs_latents = gcca.fit_transform(Xs)
```

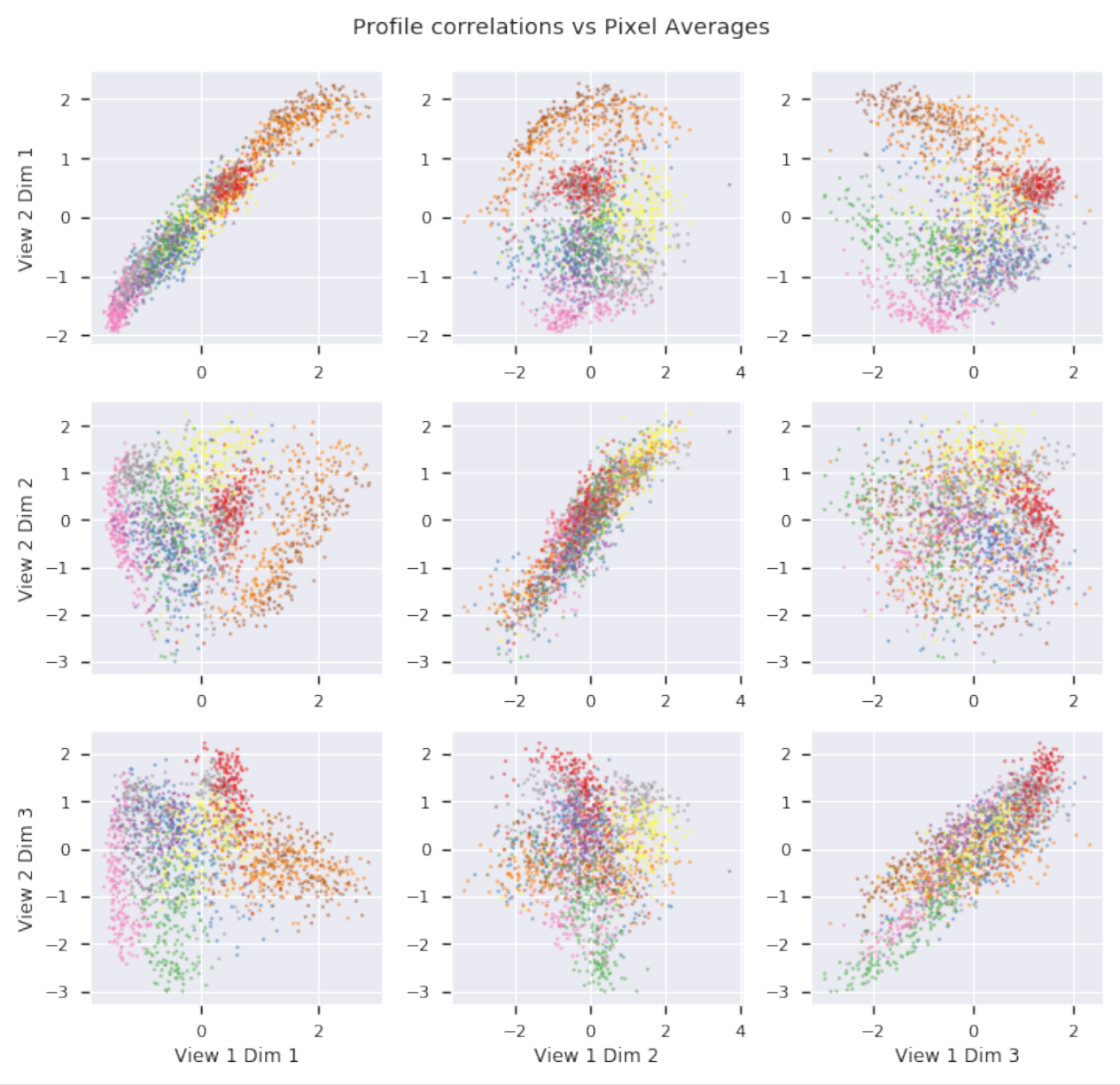
[95]:
```
print(f'The feature sizes are: {[X.shape[1] for X in Xs_latents]}')
```

```
The feature sizes are: [5, 5, 5]
```

Plot the first two views against each other

The top three dimensions from the latents spaces of the profile correlation and pixel average views are plotted against each other. However, their latent spaces are influenced the the Karhunen-Love coefficients, not plotted.

[106]:
```
crossviews_plot(Xs_latents[[0,2]], dimensions=[0,1,2], labels=y, cmap='Set1', title=f
↪'Profile correlations vs Pixel Averages', scatter_kwargs={'alpha':0.4, 's':2.0})
```



GCCA vs PCA

[1]:
```
from mvlearn.embed import GCCA
import matplotlib.pyplot as plt
import numpy as np
import scipy
%matplotlib inline
import seaborn as sns
from scipy.sparse.linalg import svds
```

[2]:
```
def get_train_test(n=100, mu=0, var=1, var2=1, nviews=3,m=1000):
    # Creates train and test data with a
    # - shared signal feature ~ N(mu, var1)
```

(continues on next page)

(continued from previous page)

```
    # - an independent noise feature ~ N(mu, var2)
    # - independent noise feautures ~ N(0, 1)
    np.random.seed(0)

    X_TRAIN = np.random.normal(mu,var,(n,1))
    X_TEST = np.random.normal(mu,var,(n,1))

    Xs_train = []
    Xs_test = []
    for i in range(nviews):
        X_train = np.hstack((np.random.normal(0,1,(n,i)),
                             X_TRAIN,
                             np.random.normal(0,1,(n,m-2-i)),
                             np.random.normal(0,var2,(n,1))
                            ))
        X_test = np.hstack((np.random.normal(0,1,(n,i)),
                            X_TEST,
                            np.random.normal(0,1,(n,m-2-i)),
                            np.random.normal(0,var2,(n,1))
                           ))

        Xs_train.append(X_train)
        Xs_test.append(X_test)

    return(Xs_train,Xs_test)
```

Positive Test

Setting:

1 high variance shared signal feature, 1 high variance noise feature

[3]:
```
nviews = 3
Xs_train, Xs_test = get_train_test(var=10,var2=10,nviews=nviews,m=1000)
```

[5]:
```
gcca = GCCA(n_components=2)
gcca.fit(Xs_train)
Xs_hat = gcca.transform(Xs_test)
```

Results:

- GCCA results show high correlation on testing data

[6]:
```
np.corrcoef(np.array(Xs_hat)[:,:,0])
```

[6]:
```
array([[1.        , 0.99698235, 0.99687182],
       [0.99698235, 1.        , 0.99689792],
       [0.99687182, 0.99689792, 1.        ]])
```

[7]:
```
Xs_hat = []
for i in range(len(Xs_train)):
    _,_,vt = svds(Xs_train[i],k=1)
    Xs_hat.append(Xs_test[i] @ vt.T)
```

- PCA selects shared dimension but also high noise dimension and so weaker correlation on testing data

[8]:
```
np.corrcoef(np.array(Xs_hat)[:,:,0])
```

[8]:
```
array([[ 1.        , -0.54014795,  0.51173297],
       [-0.54014795,  1.        , -0.98138902],
       [ 0.51173297, -0.98138902,  1.        ]])
```

Negative Test

Setting:

1 low variance shared feature

[9]:
```
nviews = 3
Xs_train, Xs_test = get_train_test(var=1,var2=1,nviews=nviews,m=1000)
```

[10]:
```
gcca = GCCA(n_components = 2)
gcca.fit(Xs_train)
Xs_hat = gcca.transform(Xs_test)
```

Results:

- GCCA fails to select shared feature and so shows low correlation on testing data

[11]:
```
np.corrcoef(np.array(Xs_hat)[:,:,0])
```

[11]:
```
array([[ 1.        ,  0.31254995, -0.02208907],
       [ 0.31254995,  1.        ,  0.13722633],
       [-0.02208907,  0.13722633,  1.        ]])
```

[12]:
```
Xs_hat = []
for i in range(len(Xs_train)):
    _,_,vt = svds(Xs_train[i],k=1)
    Xs_hat.append(Xs_test[i] @ vt.T)
```

- PCA fails to select shared feature and shows low correlation on testing data

[13]:
```
np.corrcoef(np.array(Xs_hat)[:,:,0])
```

[13]:
```
array([[1.        , 0.01016507, 0.0888701 ],
       [0.01016507, 1.        , 0.03812276],
       [0.0888701 , 0.03812276, 1.        ]])
```

Kernel CCA (KCCA)

This algorithm runs KCCA on two views of data. The kernel implementations, parameter 'ktype', are linear, polynomial and gaussian. Polynomial kernel has two parameters: 'constant', 'degree'. Gaussian kernel has one parameter: 'sigma'.

Useful information, like canonical correlations between transformed data and statistical tests for significance of these correlations can be computed using the get_stats() function of the KCCA object.

When initializing KCCA, you can also initialize the following parameters: the number of canonical components 'n_components', the regularization parameter 'reg', the decomposition type 'decomposition', and the decomposition method 'method'. There are two decomposition types: 'full' and 'icd'. In some cases, ICD will run faster than the full decomposition at the cost of performance. The only method as of now is 'kettenring-like'.

[1]:
```
import numpy as np
import sys
sys.path.append("../../..")

from mvlearn.embed.kcca import KCCA
from mvlearn.plotting.plot import crossviews_plot
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
import warnings
import matplotlib.cbook
warnings.filterwarnings("ignore",category=matplotlib.cbook.mplDeprecation)
```

Function creates Xs, a list of two views of data with a linear relationship, polynomial relationship (2nd degree) and a gaussian (sinusoidal) relationship.

[2]:
```
def make_data(kernel, N):
    # # # Define two latent variables (number of samples x 1)
    latvar1 = np.random.randn(N,)
    latvar2 = np.random.randn(N,)

    # # # Define independent components for each dataset (number of observations x␣
↪dataset dimensions)
    indep1 = np.random.randn(N, 4)
    indep2 = np.random.randn(N, 5)

    if kernel == "linear":
        x = 0.25*indep1 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2)).T
        y = 0.25*indep2 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2,␣
↪latvar1)).T

        return [x,y]

    elif kernel == "poly":
        x = 0.25*indep1 + 0.75*np.vstack((latvar1**2, latvar2**2, latvar1**2,␣
↪latvar2**2)).T
        y = 0.25*indep2 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2,␣
↪latvar1)).T

        return [x,y]

    elif kernel == "gaussian":
        t = np.random.uniform(-np.pi, np.pi, N)
        e1 = np.random.normal(0, 0.05, (N,2))
        e2 = np.random.normal(0, 0.05, (N,2))

        x = np.zeros((N,2))
        x[:,0] = t
        x[:,1] = np.sin(3*t)
        x += e1

        y = np.zeros((N,2))
        y[:,0] = np.exp(t/4)*np.cos(2*t)
```

(continues on next page)

(continued from previous page)

```
        y[:,1] = np.exp(t/4)*np.sin(2*t)
        y += e2

        return [x,y]
```

Linear kernel implementation

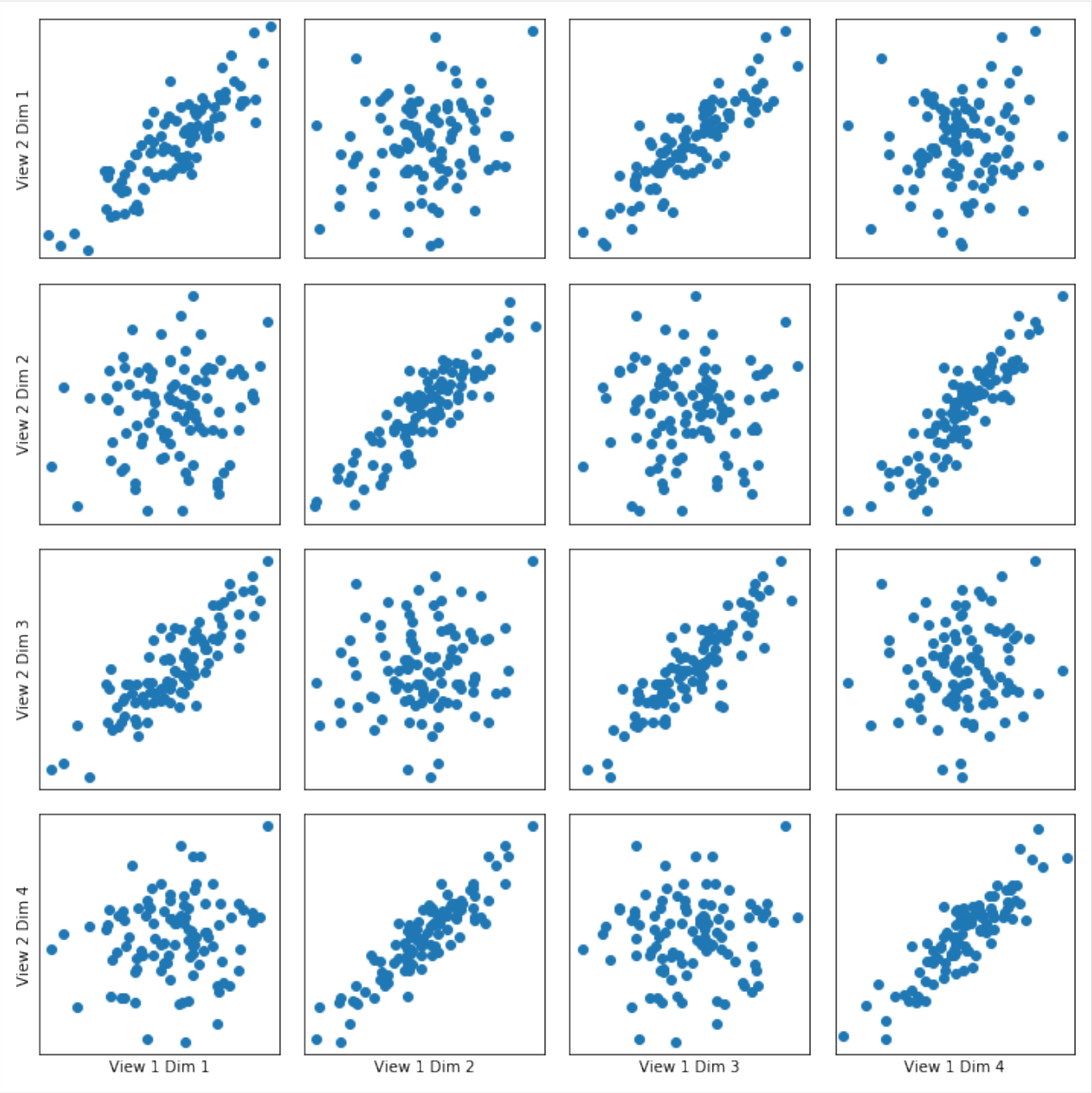
Here we show how KCCA with a linear kernel can uncover the highly correlated latent distribution of the 2 views which are related with a linear relationship, and then transform the data into that latent space. We use an 80-20, train-test data split to develop the embedding.

Also, we use statistical tests (Wilk's Lambda) to check the significance of the canonical correlations.

[3]:
```
np.random.seed(1)
Xs = make_data('linear', 100)
Xs_train = [Xs[0][:80],Xs[1][:80]]
Xs_test = [Xs[0][80:],Xs[1][80:]]

kcca_l = KCCA(n_components = 4, reg = 0.01)
kcca_l.fit(Xs_train)
linearkcca = kcca_l.transform(Xs_test)
```

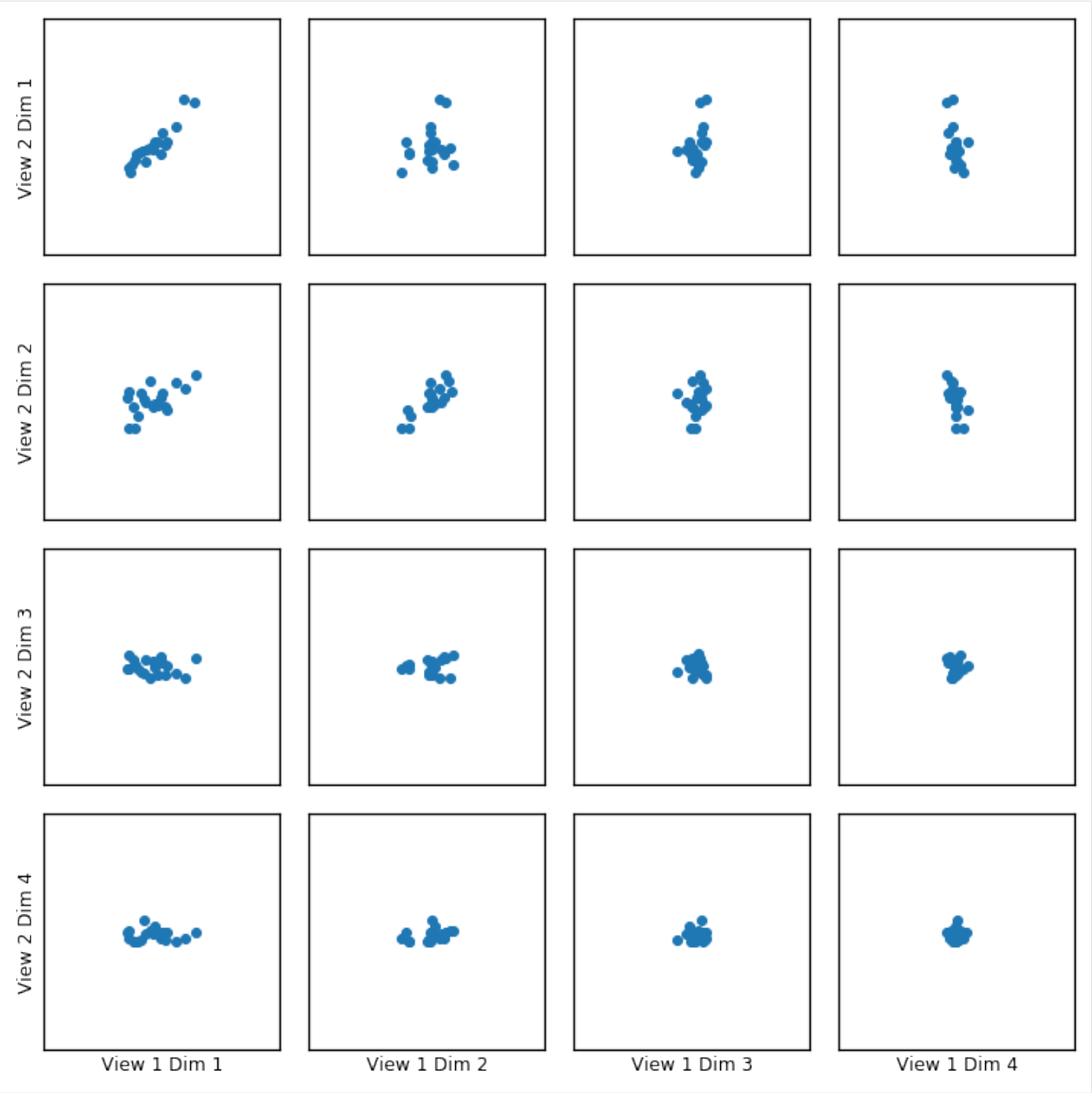
Original Data Plotted

[4]:
```
crossviews_plot(Xs, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Transformed Data Plotted

[5]:
```
crossviews_plot(linearkcca, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Now, we assess the canonical correlations achieved on the testing data, and the p-values for significance using a Wilk's Lambda test

[6]:
```
stats = kcca_l.get_stats()

print("Below are the canonical correlations and the p-values of a Wilk's Lambda test␣
↪for each components:")
print(stats['r'])
print(stats['pF'])
```

```
Below are the canonical correlations and the p-values of a Wilk's Lambda test for␣
↪each components:
[ 0.92365255  0.79419444 -0.2453487  -0.0035017 ]
```

(continues on next page)

(continued from previous page)

```
[0.00400878 0.25898906 0.99013426 0.99991417]
```

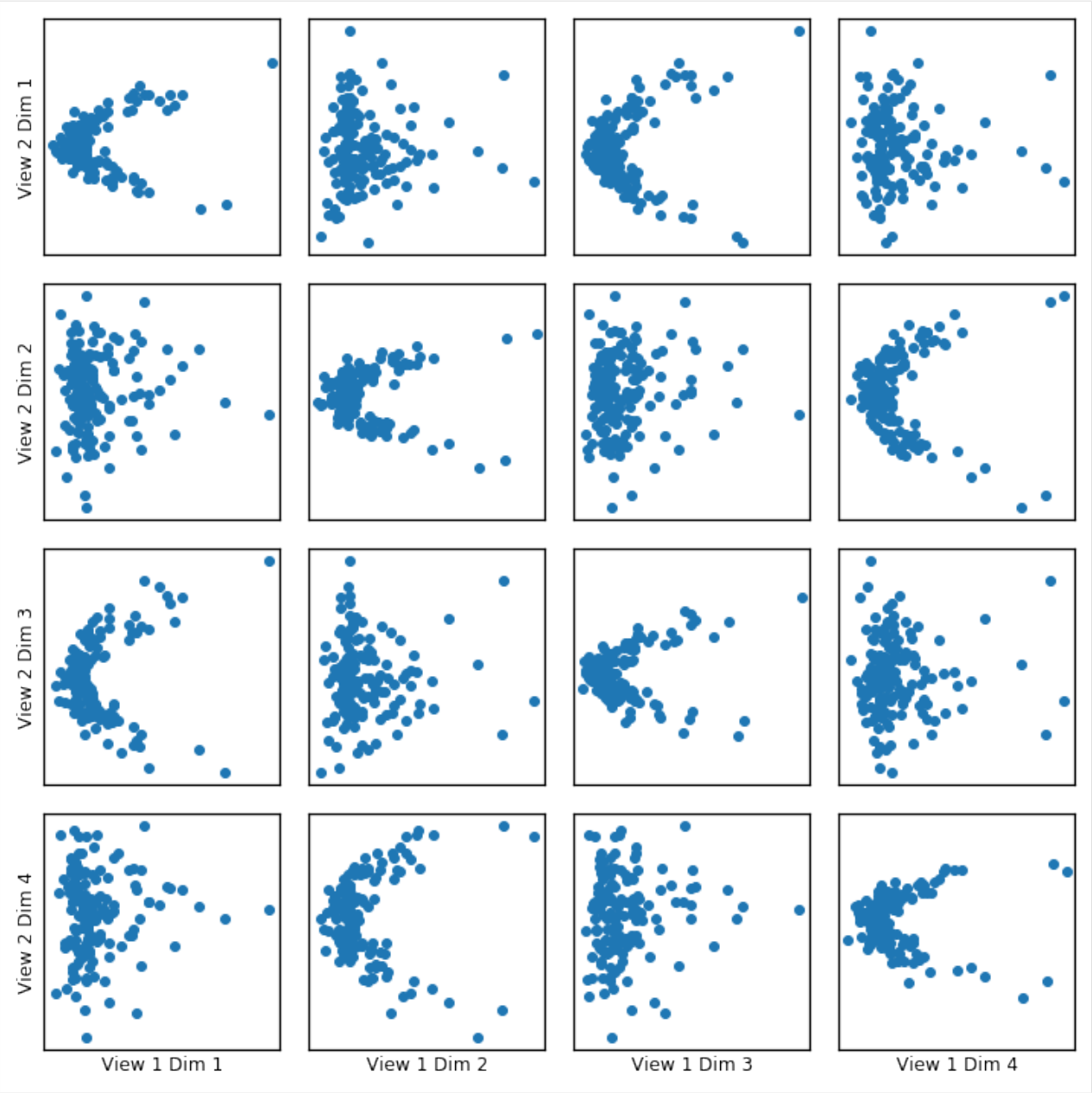
Polynomial kernel implementation

Here we show how KCCA with a polynomial kernel can uncover the highly correlated latent distribution of the 2 views which are related with a polynomial relationship, and then transform the data into that latent space.

[7]:
```
Xsp = make_data("poly", 150)
kcca_p = KCCA(ktype ="poly", degree = 2.0, n_components = 4, reg=0.001)
polykcca = kcca_p.fit_transform(Xsp)
```

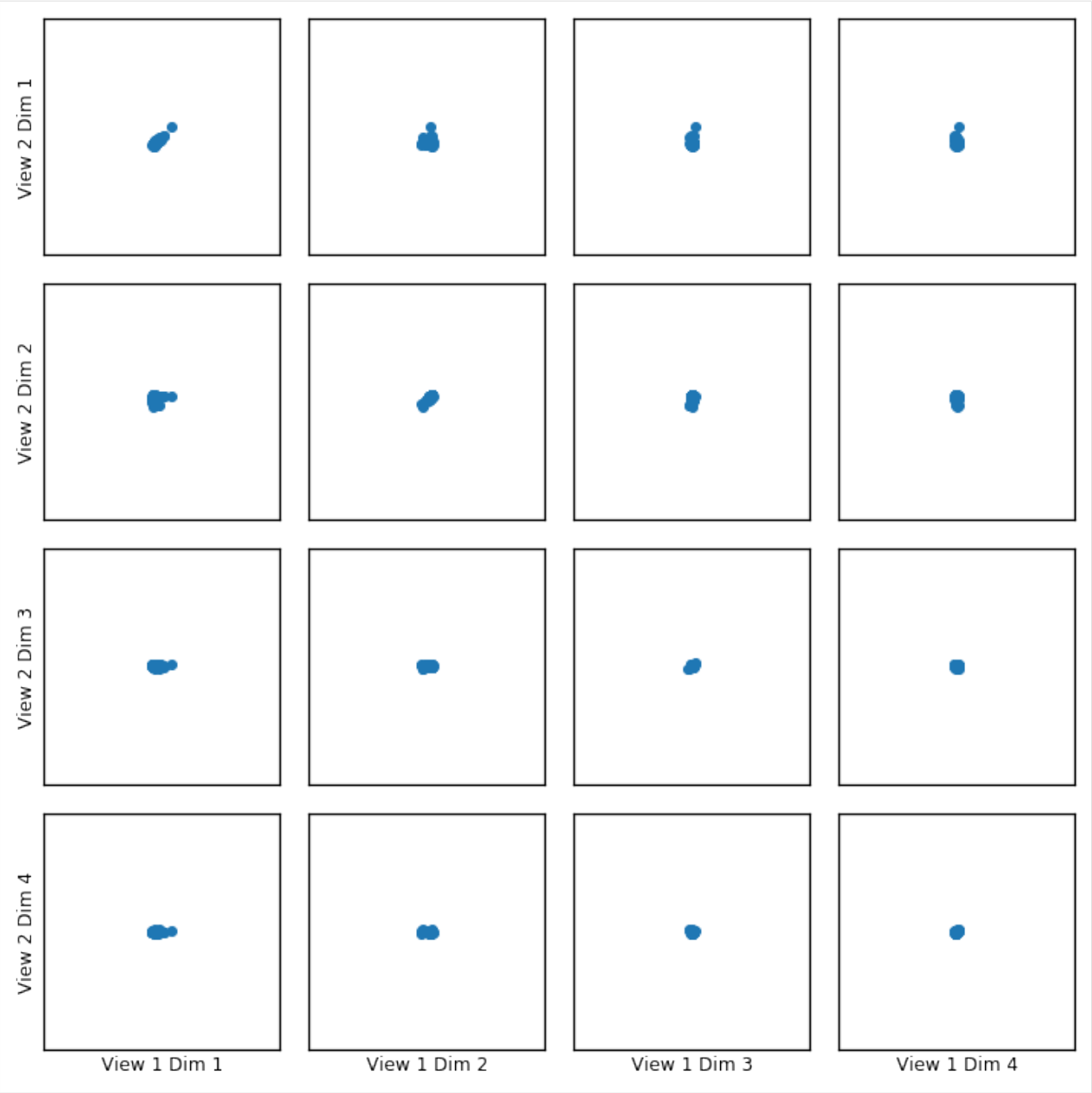
Original Data Plotted

[8]:
```
crossviews_plot(Xsp, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Transformed Data Plotted

[9]:
```
crossviews_plot(polykcca, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Now, we assess the canonical correlations achieved on the testing data

[10]:
```
stats = kcca_p.get_stats()

print("Below are the canonical correlations for each components:")
print(stats['r'])
```

```
Below are the canonical correlations for each components:
[0.96738396 0.94500285 0.63334922 0.57870821]
```

Gaussian Kernel Implementation

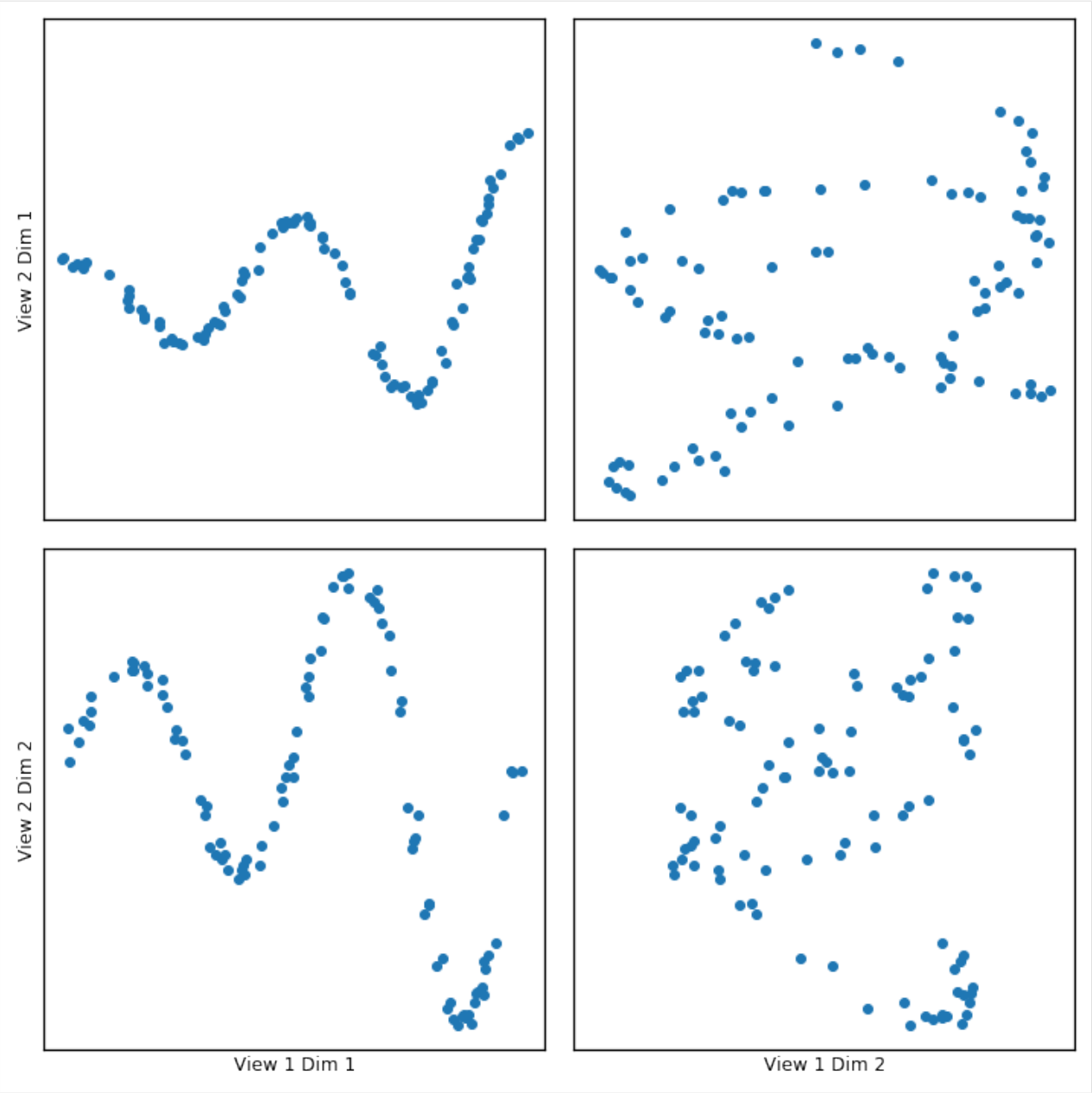
Here we show how KCCA with a gaussian kernel can uncover the highly correlated latent distribution of the 2 views which are related with a sinusoidal relationship, and then transform the data into that latent space.

[11]:
```
Xsg = make_data("gaussian", 100)
Xsg_train = [Xsg[0][:20],Xsg[1][:20]]
Xsg_test = [Xsg[0][20:],Xsg[1][20:]]
```

[12]:
```
kcca_g = KCCA(ktype ="gaussian", sigma = 1.0, n_components = 2, reg = 0.01)
kcca_g.fit(Xsg)
gausskcca = kcca_g.transform(Xsg)
```

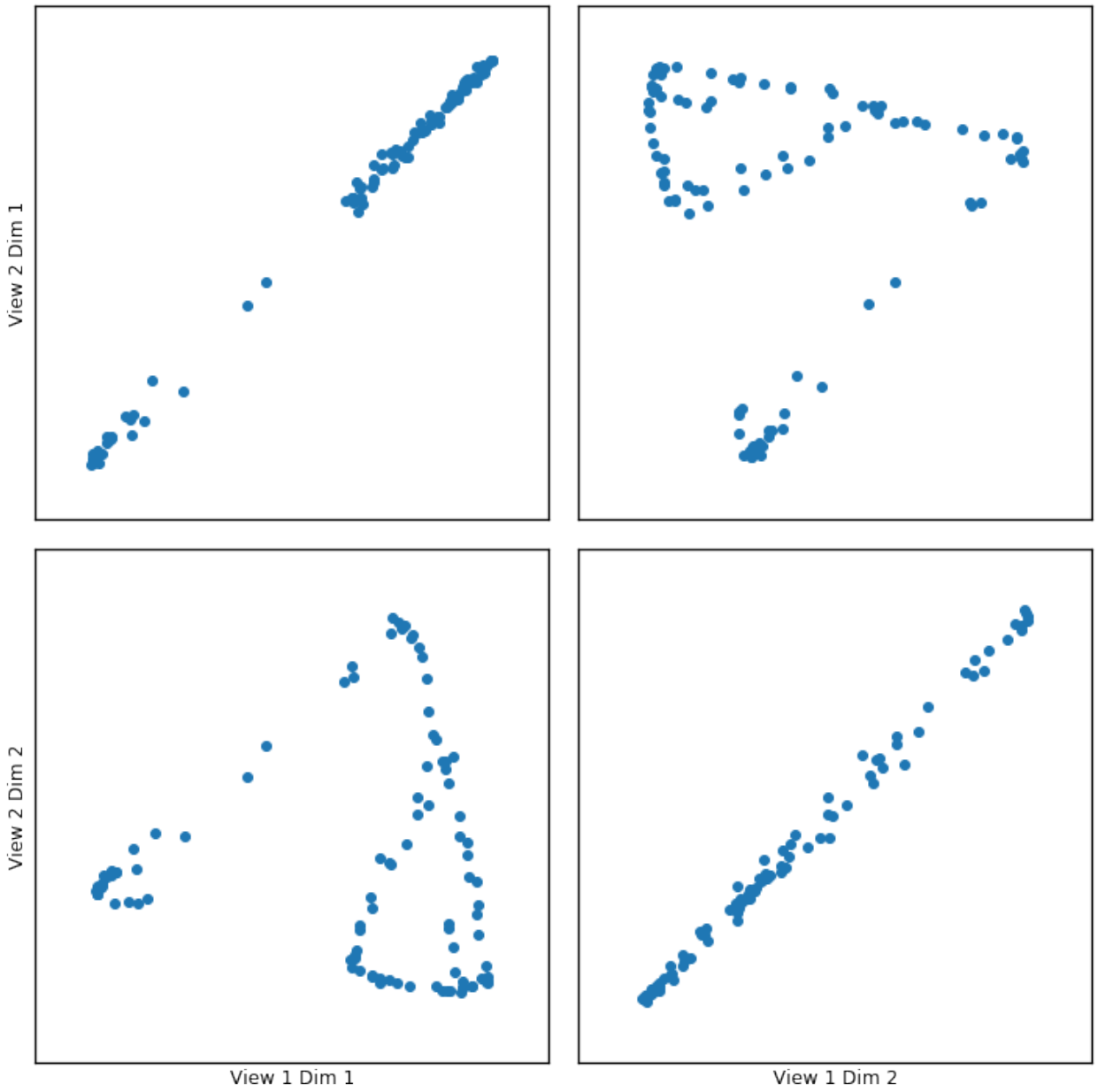
Original Data Plotted

[13]:
```
crossviews_plot(Xsg, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Transformed Data Plotted

[14]:
```
crossviews_plot(gausskcca, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Now, we assess the canonical correlations achieved on the testing data

[15]:
```
stats = kcca_g.get_stats()

print("Below are the canonical correlations for each components:")
print(stats['r'])
```

```
Below are the canonical correlations for each components:
[0.99887253 0.99762762]
```

Kernel CCA: ICD Method

Kernel matrices grow exponentially with the size of the data. There are immense storage and run-time constraints that arise when working with large datasets. The Incomplete Cholesky Decomposition (ICD) looks for a low rank approximation of the Cholesky decomposition of the kernel matrix. This reduces storage requirements from O(n^2) to O(nm), where n is the number of subjects (rows) and m is the rank of the kernel matrix. This also reduces the run-time from O(n^3) to O(nm^2).

[35]:
```
import numpy as np
import sys
sys.path.append("../../..")

from mvlearn.embed.kcca import KCCA
from mvlearn.plotting.plot import crossviews_plot
import matplotlib.pyplot as plt
%matplotlib inline
from scipy import stats
import warnings
import matplotlib.cbook
import time
warnings.filterwarnings("ignore",category=matplotlib.cbook.mplDeprecation)
```

[2]:
```
def make_data(kernel, N):
    # # # Define two latent variables (number of samples x 1)
    latvar1 = np.random.randn(N,)
    latvar2 = np.random.randn(N,)

    # # # Define independent components for each dataset (number of observations x␣
↪dataset dimensions)
    indep1 = np.random.randn(N, 4)
    indep2 = np.random.randn(N, 5)

    if kernel == "linear":
        x = 0.25*indep1 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2)).T
        y = 0.25*indep2 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2,␣
↪latvar1)).T

        return [x,y]

    elif kernel == "poly":
        x = 0.25*indep1 + 0.75*np.vstack((latvar1**2, latvar2**2, latvar1**2,␣
↪latvar2**2)).T
        y = 0.25*indep2 + 0.75*np.vstack((latvar1, latvar2, latvar1, latvar2,␣
↪latvar1)).T

        return [x,y]

    elif kernel == "gaussian":
        t = np.random.uniform(-np.pi, np.pi, N)
        e1 = np.random.normal(0, 0.05, (N,2))
        e2 = np.random.normal(0, 0.05, (N,2))

        x = np.zeros((N,2))
        x[:,0] = t
        x[:,1] = np.sin(3*t)
        x += e1
```

(continues on next page)

(continued from previous page)

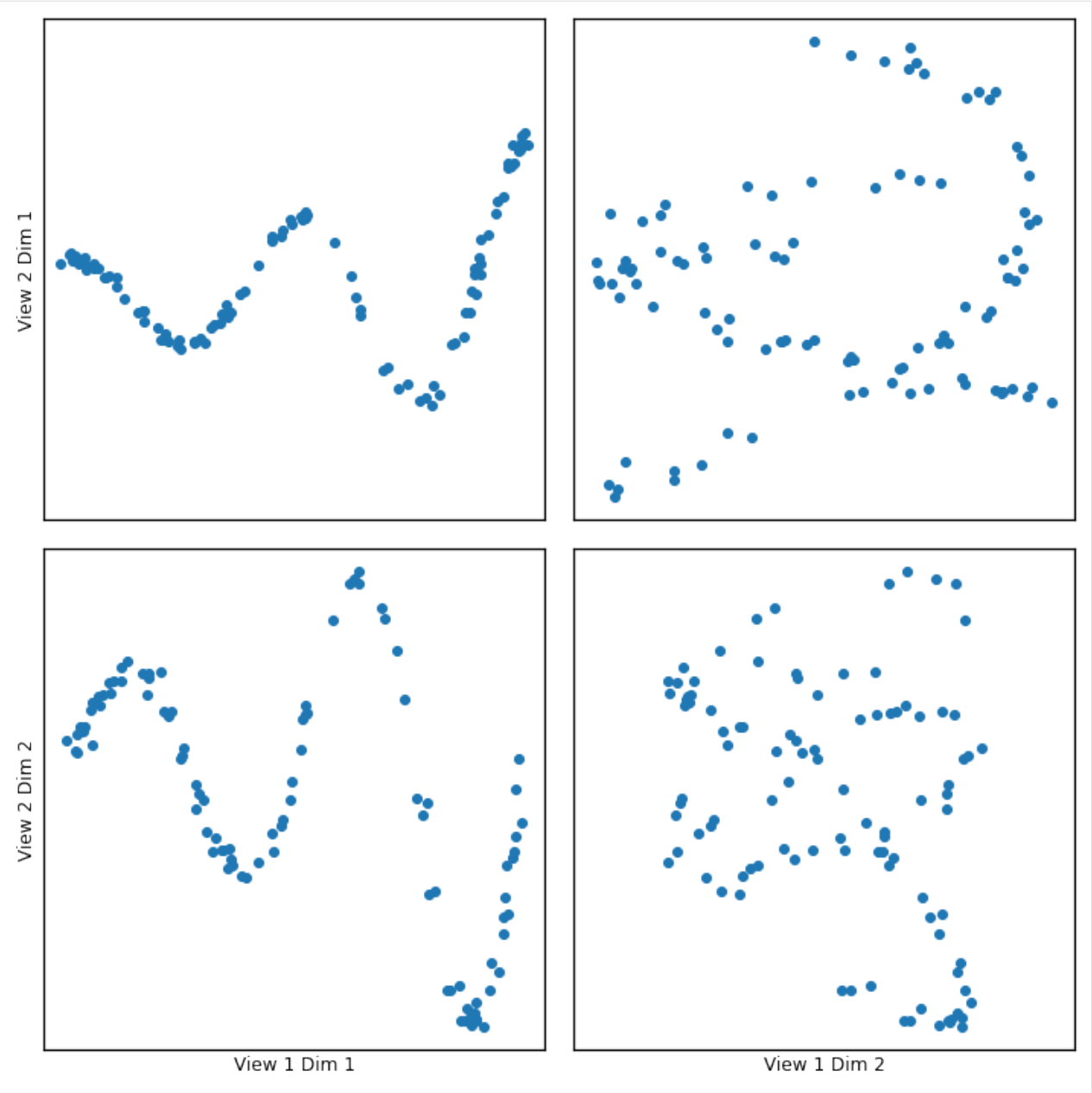
```
        y = np.zeros((N,2))
        y[:,0] = np.exp(t/4)*np.cos(2*t)
        y[:,1] = np.exp(t/4)*np.sin(2*t)
        y += e2

        return [x,y]
```

Full Decomposition vs ICD on Sample Data

ICD is run on two views of data that each have two dimensions that are sinuisoidally related. The data has 100 samples and thus the fully decomposed kernel matrix would have dimensions (100, 100). Instead we implement ICD with a kernel matrix of rank 50 (mrank = 50).

[7]:
```
np.random.seed(1)
Xsg = make_data('gaussian', 100)
```

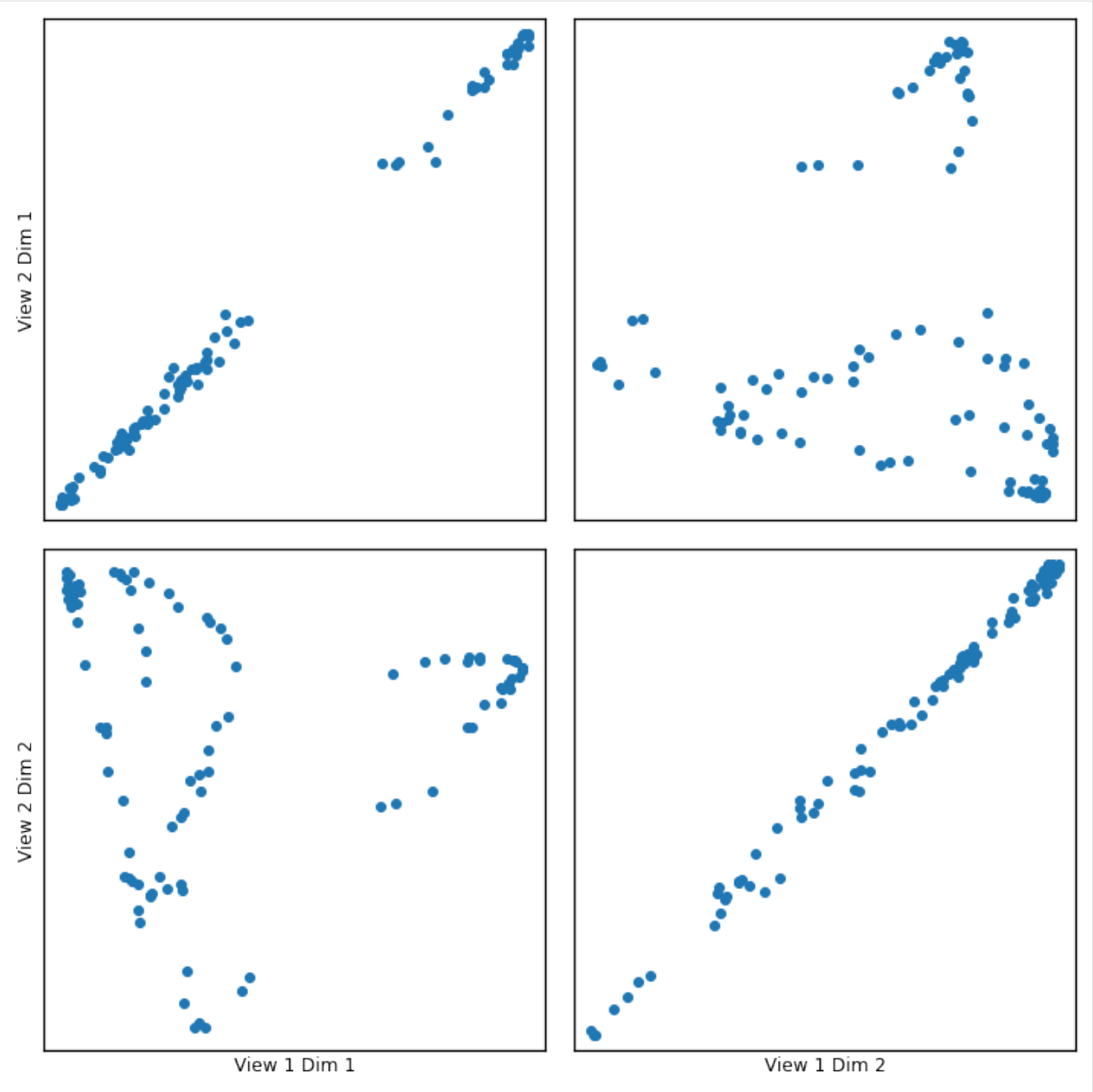
[9]:
```
crossviews_plot(Xsg, ax_ticks=False, ax_labels=True, equal_axes=True)
```



Full Decomposition

[8]:
```
kcca_g = KCCA(ktype ="gaussian", n_components = 2, reg = 0.01)
kcca_g.fit(Xsg)
gausskcca = kcca_g.transform(Xsg)
```

[10]:
```
crossviews_plot(gausskcca, ax_ticks=False, ax_labels=True, equal_axes=True)
```



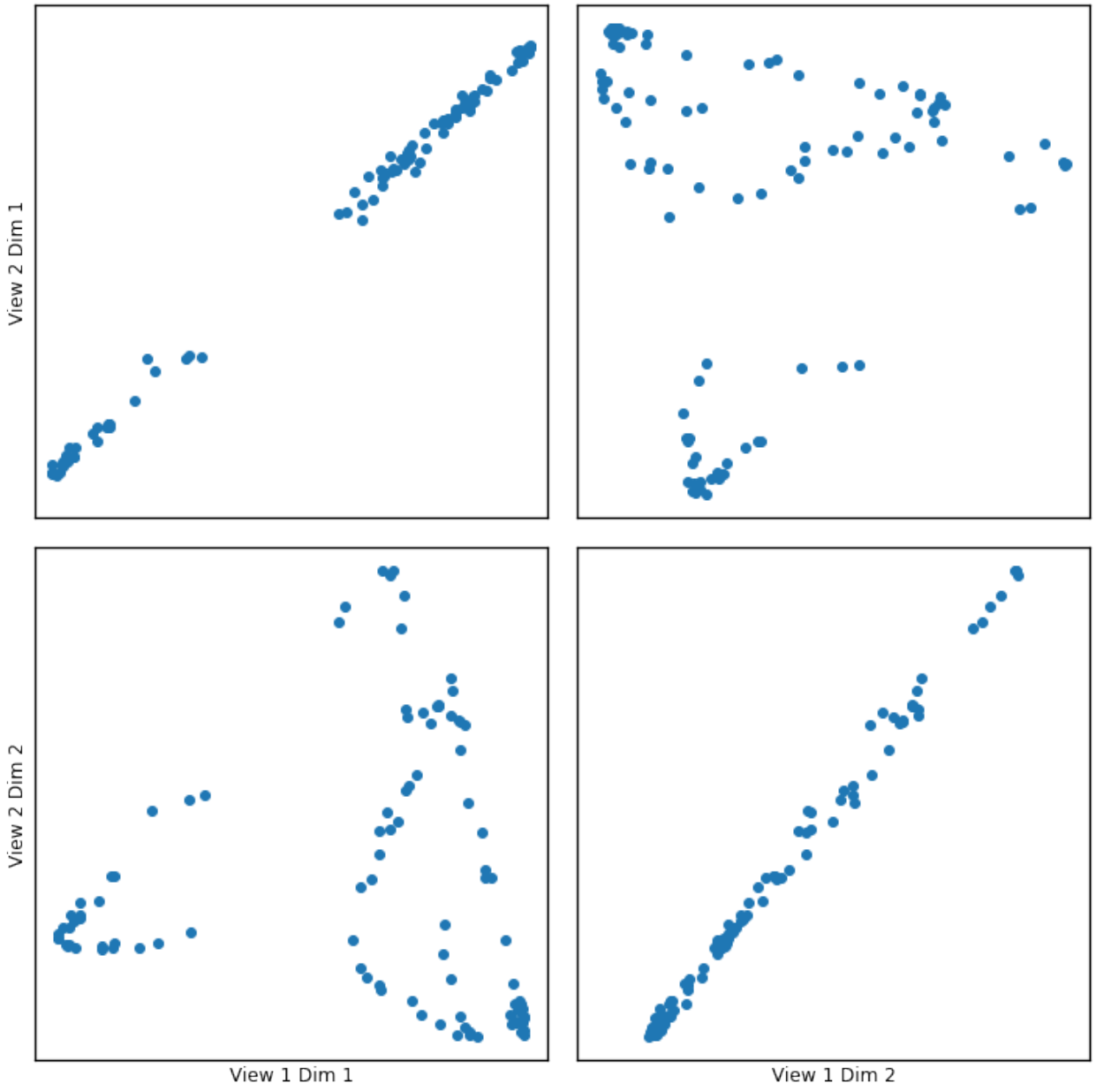
[11]:
```
(gr1, _) = stats.pearsonr(gausskcca[0][:,0], gausskcca[1][:,0])
(gr2, _) = stats.pearsonr(gausskcca[0][:,1], gausskcca[1][:,1])

print("Below are the canonical correlation of the two components:")
print(gr1,gr2)
```

```
Below are the canonical correlation of the two components:
0.9988060118791638 0.9972876357732628
```

ICD Decomposition

[12]:
```
kcca_g_icd = KCCA(ktype = "gaussian", sigma = 1.0, n_components = 2, reg = 0.01,␣
↪decomp = 'icd', mrank = 50)
icd_g = kcca_g_icd.fit_transform(Xsg)
```

[13]:
```
crossviews_plot(icd_g, ax_ticks=False, ax_labels=True, equal_axes=True)
```



[15]:
```
(icdr1, _) = stats.pearsonr(icd_g[0][:,0], icd_g[1][:,0])
(icdr2, _) = stats.pearsonr(icd_g[0][:,1], icd_g[1][:,1])

print("Below are the canonical correlation of the two components:")
print(icdr1,icdr2)
```

```
Below are the canonical correlation of the two components:
0.998805983433145 0.997287542632157
```

The canonical correlations of full vs ICD (mrank=50) are very similar!

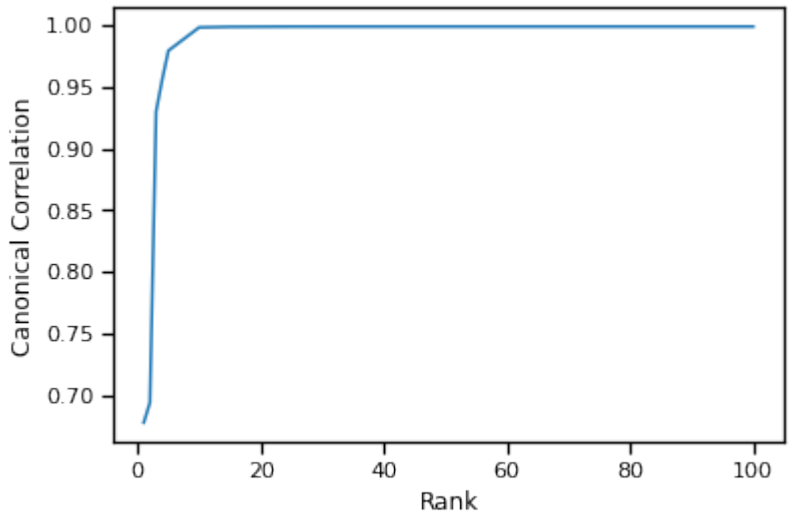
ICD Kernel Rank vs. Canonical Correlation

We can observe the relationship between the ICD kernel rank and canonical correlation of the first canonical component.

[32]:
```
can_corrs = []
rank = [1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,
↪ 90, 95, 100]

for i in rank:
    kcca_g_icd = KCCA(ktype = "gaussian", sigma = 1.0, n_components = 2, reg = 0.01,␣
↪decomp = 'icd', mrank = i)
    icd_g = kcca_g_icd.fit_transform(Xsg)
    (icdr1, _) = stats.pearsonr(icd_g[0][:,0], icd_g[1][:,0])
    can_corrs.append(icdr1)
```

[34]:
```
plt.plot(rank, can_corrs)
plt.xlabel('Rank')
plt.ylabel('Canonical Correlation')
```

[34]:
```
Text(0, 0.5, 'Canonical Correlation')
```



We observe that around rank=10-15 we achieve the same canonical correlation as the fully decomposed kernel matrix (rank=100).

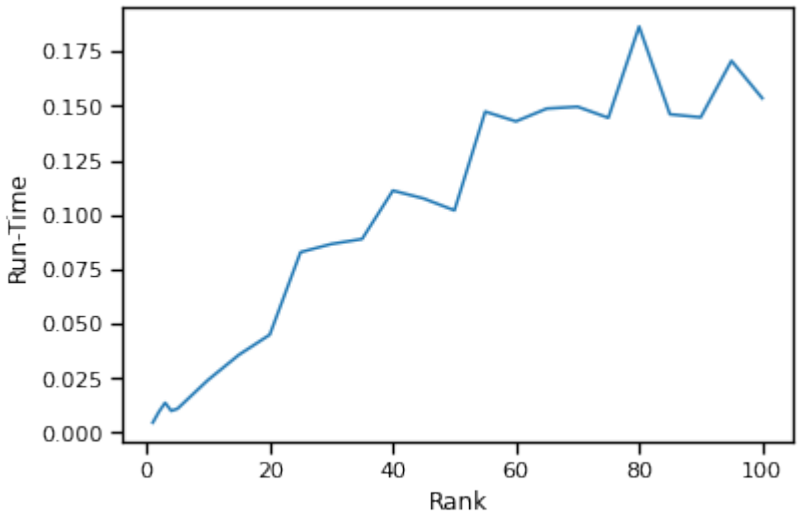
ICD Kernel Rank vs Run-Time

We can observe the relationship between the ICD kernel rank and run-time to fit and transform the two views. We average the run-time of each rank over 5 trials.

[38]:
```
run_time = []

for i in rank:
    run_time_sample = []
    for a in range(5):
        kcca_g_icd = KCCA(ktype = "gaussian", sigma = 1.0, n_components = 2, reg = 0.
↪01, decomp = 'icd', mrank = i)
        start = time.time()
        icd_g = kcca_g_icd.fit_transform(Xsg)
        run_time_sample.append(time.time()-start)
    run_time.append(sum(run_time_sample) / len(run_time_sample))
```

[39]:
```
plt.plot(rank, run_time)
plt.xlabel('Rank')
plt.ylabel('Run-Time')
```

[39]:
```
Text(0, 0.5, 'Run-Time')
```



From the rank vs canonical correlation analysis in the previous section, we discovered that a rank of 10-15 will preserve the canonical correlation (accuracy). We can see that at a rank of 10-15, we can get an order of magnitude decrease in run-time compared to a rank of 100 (full decomposition).

Deep CCA (DCCA)

In this example, we show how to used Deep CCA to uncover latent correlations between views.

[1]:
```
from mvlearn.embed import DCCA

from mvlearn.datasets import GaussianMixture
from mvlearn.plotting import crossviews_plot
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

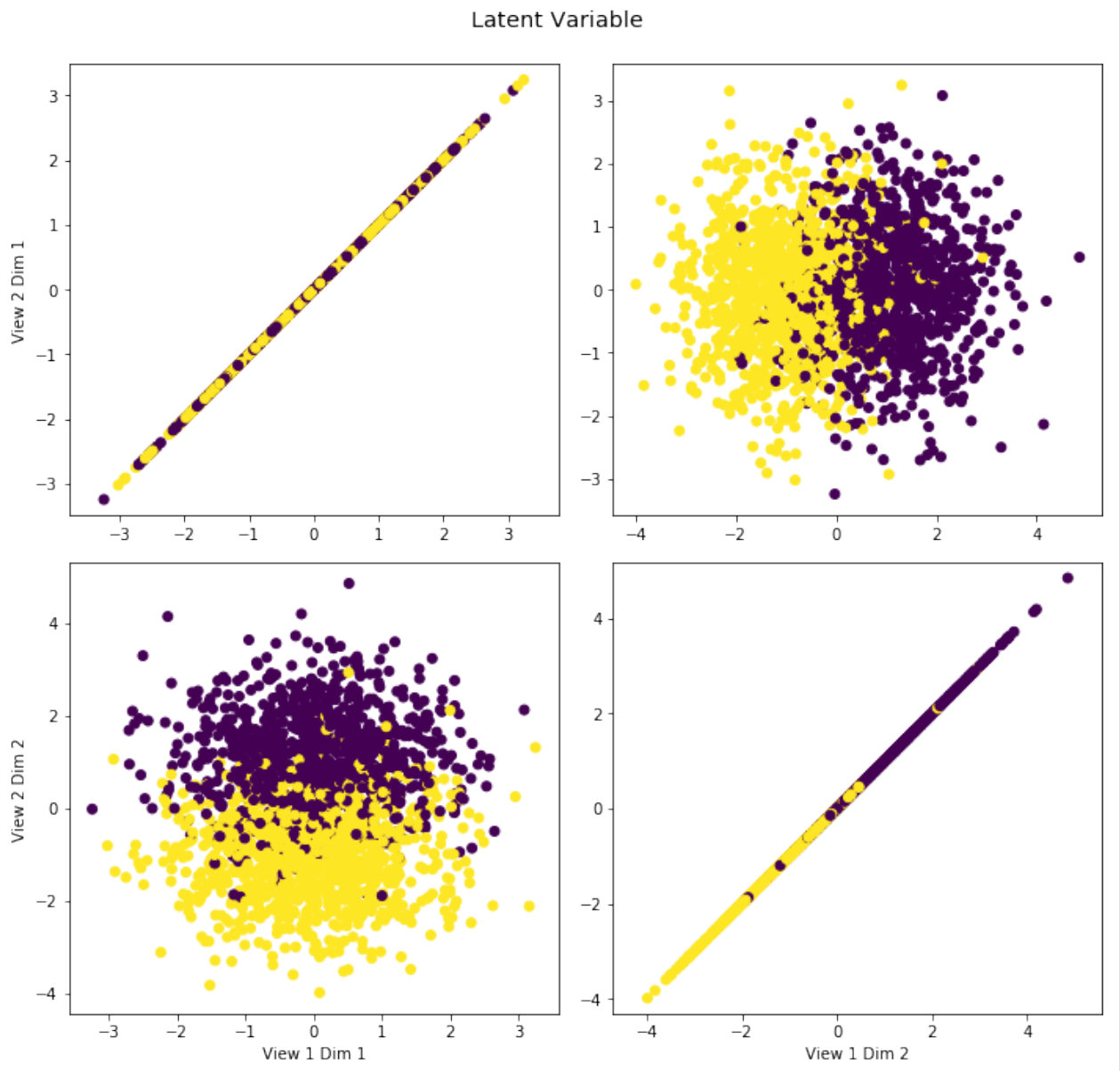
Polynomial-Transformed Latent Correlation

Latent variables are sampled from two multivariate Gaussians with equal prior probability. Then a polynomial transformation is applied and noise is added independently to both the transformed and untransformed latents.

[3]:
```
n_samples = 2000
centers = [[0,1], [0,-1]]
covariances = [np.eye(2), np.eye(2)]
GM = GaussianMixture(n_samples, centers, covariances, random_state=42,
                     shuffle=True, shuffle_random_state=42)
GM = GM.sample_views(transform='poly', n_noise=2)
```

The latent data is plotted against itself to reveal the underlying distribtution.

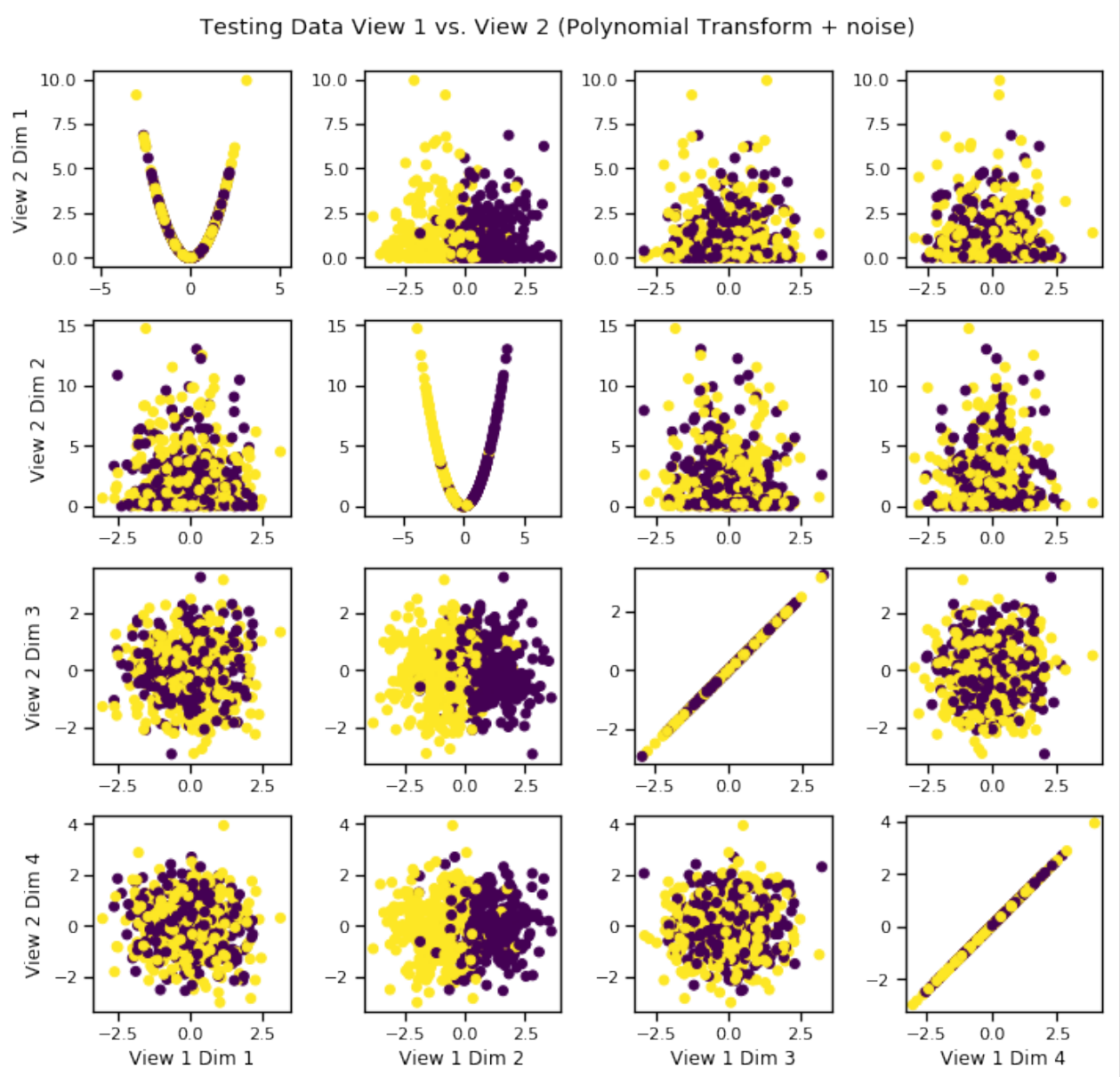
[4]:
```
crossviews_plot([GM.latent, GM.latent], labels=GM.y, title='Latent Variable', equal_
↪axes=True)
```



The noisy latent variable (view 1) is plotted against the transformed latent variable (view 2), an example of a dataset with two views.

[5]:
```
# Split data into train and test segments
Xs_train = []
Xs_test = []
max_row = int(GM.Xs[0].shape[0] * .7)
Xs, y = GM.get_Xy(latents=False)
for X in Xs:
    Xs_train.append(X[:max_row, :])
    Xs_test.append(X[max_row:, :])
y_train = y[:max_row]
y_test = y[max_row:]
```

[6]:
```
crossviews_plot(Xs_test, labels=y_test, title='Testing Data View 1 vs. View 2␣
↪(Polynomial Transform + noise)', equal_axes=True)
```



Fit DCCA model to uncover latent distribution

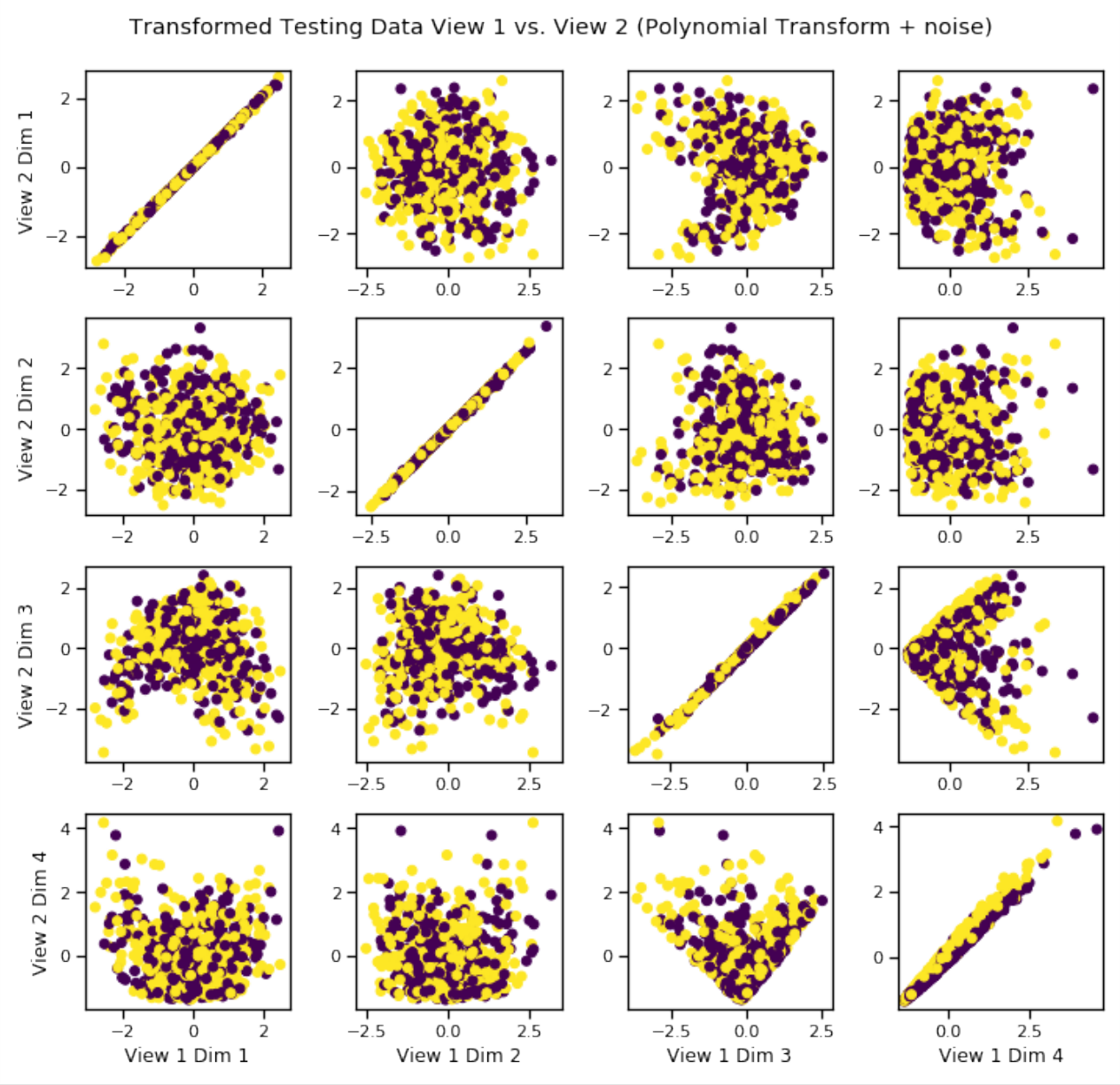
The output dimensionality is still 4.

[7]:
```
# Define parameters and layers for deep model
features1 = Xs_train[0].shape[1] # Feature sizes
features2 = Xs_train[1].shape[1]
layers1 = [1024, 512, 4] # nodes in each hidden layer and the output size
layers2 = [1024, 512, 4]

dcca = DCCA(input_size1=features1, input_size2=features2, n_components=4,
            layer_sizes1=layers1, layer_sizes2=layers2)
dcca.fit(Xs_train)
Xs_transformed = dcca.transform(Xs_test)
```

Visualize the transformed data

We can see that it has uncovered the latent correlation between views.

[8]:
```
crossviews_plot(Xs_transformed, labels=y_test, title='Transformed Testing Data View 1␣
↪vs. View 2 (Polynomial Transform + noise)', equal_axes=True)
```



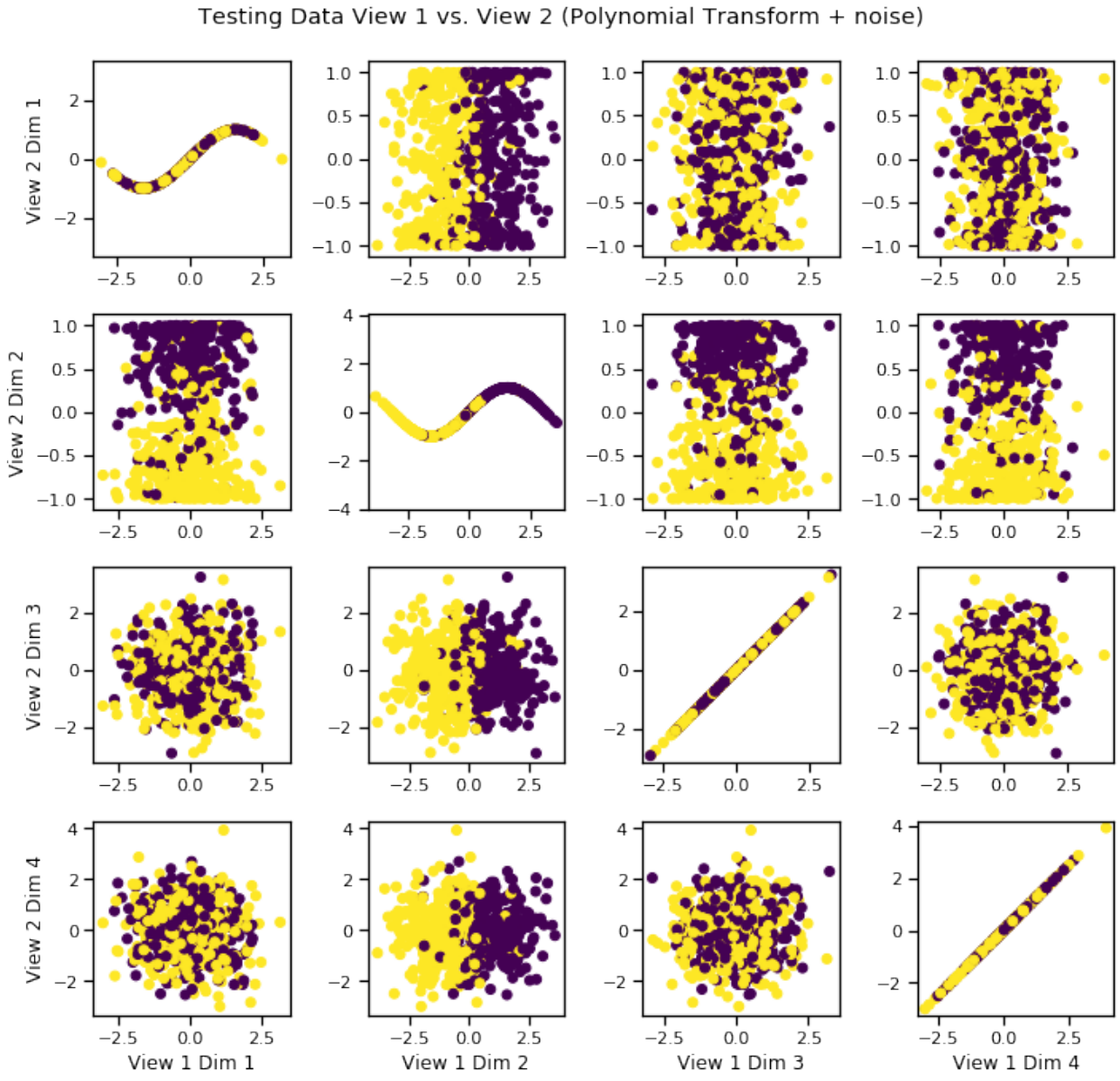
Sinusoidal-Transformed Latent Correlation

Following the same procedure as above, latent variables are sampled from two multivariate Gaussians with equal prior probability. This time, a sinusoidal transformation is applied and noise is added independently to both the transformed and untransformed latents.

[9]:
```
n = 2000
mu = [[0,1], [0,-1]]
sigma = [np.eye(2), np.eye(2)]
class_probs = [0.5, 0.5]
GM = GaussianMixture(mu,sigma,n,class_probs=class_probs, random_state=42,
                     shuffle=True, shuffle_random_state=42)
GM = GM.sample_views(transform='sin', n_noise=2)
```

[10]:
```
# Split data into train and test segments
Xs_train = []
Xs_test = []
max_row = int(GM.Xs[0].shape[0] * .7)
for X in GM.Xs:
    Xs_train.append(X[:max_row, :])
    Xs_test.append(X[max_row:, :])
y_train = GM.y[:max_row]
y_test = GM.y[max_row:]
```

[11]:
```
crossviews_plot(Xs_test, labels=y_test, title='Testing Data View 1 vs. View 2
↪(Polynomial Transform + noise)', equal_axes=True)
```



Fit DCCA model to uncover latent distribution

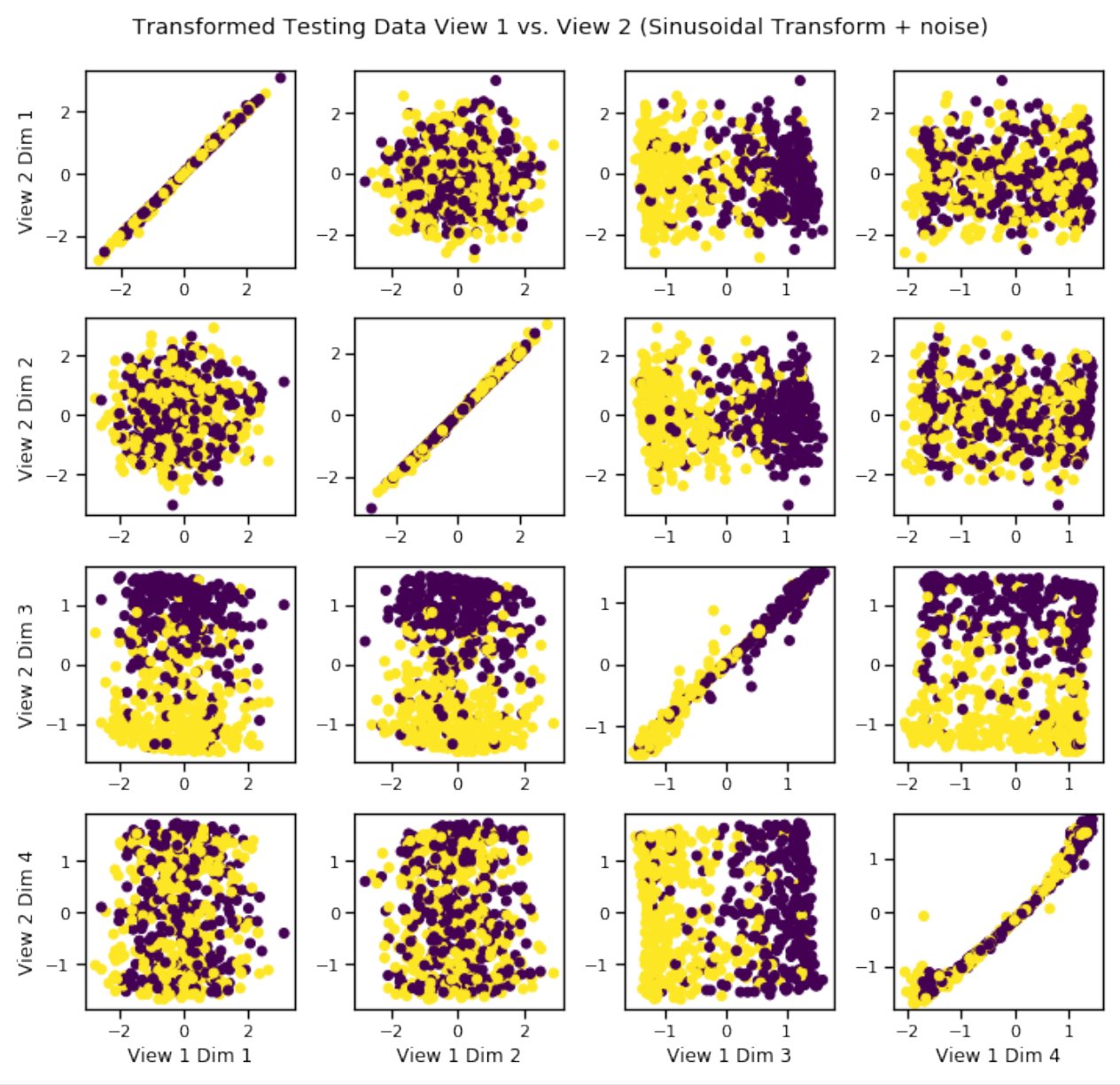
The output dimensionality is still 4.

[12]:
```
# Define parameters and layers for deep model
features1 = Xs_train[0].shape[1] # Feature sizes
features2 = Xs_train[1].shape[1]
layers1 = [1024, 512, 4] # nodes in each hidden layer and the output size
layers2 = [1024, 512, 4]

dcca = DCCA(input_size1=features1, input_size2=features2, n_components=4,
            layer_sizes1=layers1, layer_sizes2=layers2)
dcca.fit(Xs_train)
Xs_transformed = dcca.transform(Xs_test)
```

Visualize the transformed data

We can see that it has uncovered the latent correlation between views.

[13]:
```
crossviews_plot(Xs_transformed, labels=y_test, title='Transformed Testing Data View 1␣
↪vs. View 2 (Sinusoidal Transform + noise)', equal_axes=True)
```



CCA Variants Comparison

A comparison of Kernel Canonical Correlation Analysis (KCCA) with three different types of kernel to Deep Canonical Correlation Analysis (DCCA). Each learns and computes kernels suitable for different situations. The point of this tutorial is to illustrate, in toy examples, the rough intuition as to when such methods work well and generate linearly correlated projections.

The simulated latent data has two signal dimensions draw from independent Gaussians. Two views of data were derived from this.

- View 1: The latent data.
- View 2: A transformation of the latent data.

To each view, two additional independent Gaussian noise dimensions were added.

Each 2x2 grid of subplots in the figure corresponds to a transformation and either the raw data or a CCA variant. The x-axes are the data from view 1 and the y-axes are the data from view 2. Plotted are the correlations between the signal dimensions of the raw views and the top two components of each view after a CCA variant transformation. Linearly correlated plots on the diagonals of the 2x2 grids indicate that the CCA method was able to successfully learn the underlying functional relationship between the two views.

[2]:
```
from mvlearn.embed import KCCA, DCCA
from mvlearn.datasets import GaussianMixture
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
import seaborn as sns
```

[3]:
```
## Make Latents
n_samples = 200
centers = [[0,1], [0,-1]]
covariances = 2*np.array([np.eye(2), np.eye(2)])
GM_train = GaussianMixture(n_samples, centers, covariances)

## Test
GM_test = GaussianMixture(n_samples, centers, covariances)

## Make 2 views
n_noise = 2
transforms = ['linear', 'poly', 'sin']

Xs_train = []
Xs_test = []
for transform in transforms:
    GM_train.sample_views(transform=transform, n_noise=n_noise)
    GM_test.sample_views(transform=transform, n_noise=n_noise)

    Xs_train.append(GM_train.get_Xy()[0])
    Xs_test.append(GM_test.get_Xy()[0])
```

[4]:
```
## Plotting parameters
labels = GM_test.latent[:,0]
cmap = matplotlib.colors.ListedColormap(sns.diverging_palette(240, 10, n=len(labels),␣
↪center='light').as_hex())
cmap = 'coolwarm'

method_labels = ['Raw Views', 'Linear KCCA', 'Polynomial KCCA', 'Gaussian KCCA', 'DCCA
↪']
transform_labels = ['Linear Transform', 'Polynomial Transform', 'Sinusoidal Transform
↪']
```

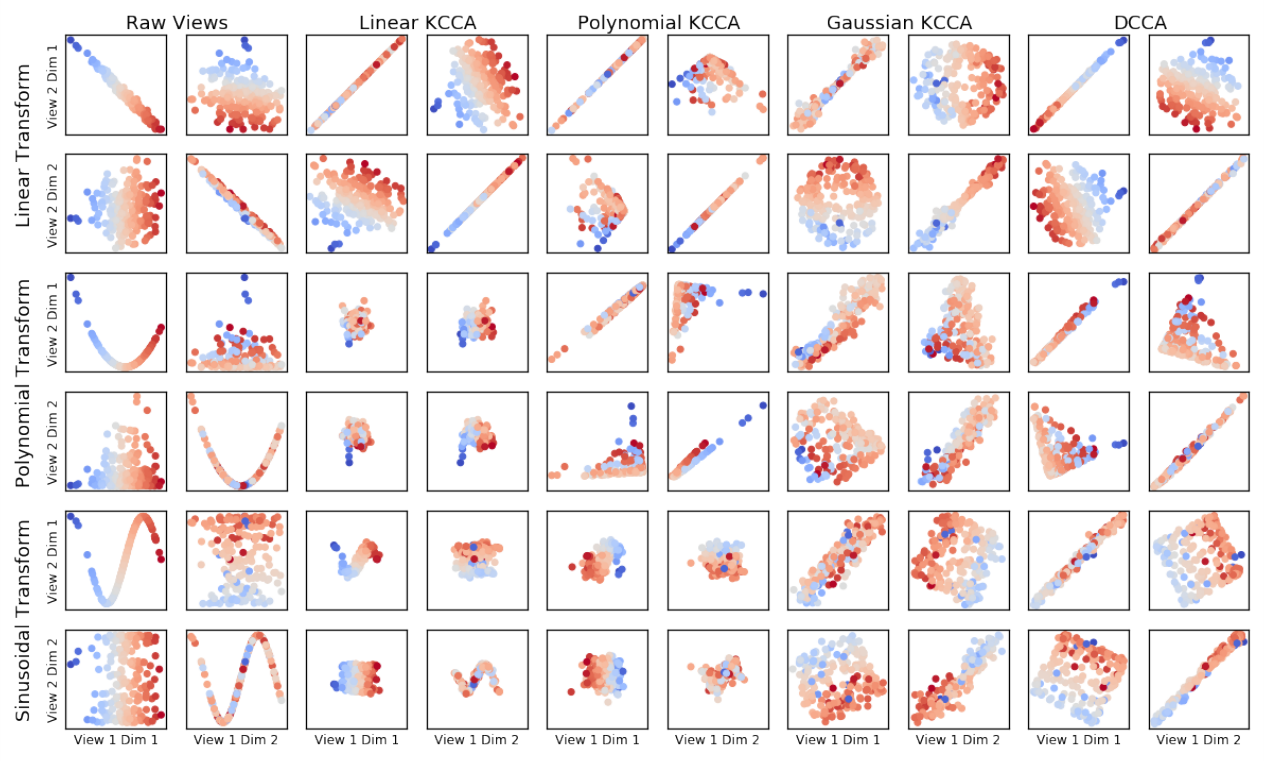
[5]:
```
input_size1, input_size2 = Xs_train[0][0].shape[1], Xs_train[0][1].shape[1]
outdim_size = min(Xs_train[0][0].shape[1], 2)
layer_sizes1 = [256, 256, outdim_size]
layer_sizes2 = [256, 256, outdim_size]
methods = [KCCA(ktype='linear', reg = 0.1, degree=2.0, constant=0.1, n_components =␣
↪2),
           KCCA(ktype='poly', reg = 0.1, degree=2.0, constant=0.1, n_components = 2),
           KCCA(ktype='gaussian', reg = 1.0, sigma=2.0, n_components = 2),
           DCCA(input_size1, input_size2, outdim_size, layer_sizes1, layer_sizes2,␣
↪epoch_num=400)
```

(continues on next page)

(continued from previous page)

```
]
```

[15]:
```
fig, axes = plt.subplots(3*2, 5*2, figsize=(20,12))
sns.set_context('notebook')

for r,transform in enumerate(transforms):
    axs = axes[2*r:2*r+2,:2]
    for i,ax in enumerate(axs.flatten()):
        dim2 = int(i / 2)
        dim1 = i % 2
        ax.scatter(
            Xs_test[r][0][:, dim1],
            Xs_test[r][1][:, dim2],
            cmap=cmap,
            c=labels,
        )
        ax.set_xticks([], [])
        ax.set_yticks([], [])
        if dim1 == 0:
            ax.set_ylabel(f"View 2 Dim {dim2+1}")
        if dim1 == 0 and dim2 == 0:
            ax.text(-0.5, -0.1, transform_labels[r], transform=ax.transAxes,␣
↪fontsize=18, rotation=90, verticalalignment='center')
        if dim2 == 1 and r == len(transforms)-1:
            ax.set_xlabel(f"View 1 Dim {dim1+1}")
        if i == 0 and r == 0:
            ax.set_title(method_labels[r], {'position':(1.11,1), 'fontsize':18})

    for c,method in enumerate(methods):
        axs = axes[2*r:2*r+2,2*c+2:2*c+4]
        Xs = method.fit(Xs_train[r]).transform(Xs_test[r])
        for i,ax in enumerate(axs.flatten()):
            dim2 = int(i / 2)
            dim1 = i % 2
            ax.scatter(
                Xs[0][:, dim1],
                Xs[1][:, dim2],
                cmap=cmap,
                c=labels,
            )
            if dim2 == 1 and r == len(transforms)-1:
                ax.set_xlabel(f"View 1 Dim {dim1+1}")
            if i == 0 and r == 0:
                ax.set_title(method_labels[c+1], {'position':(1.11,1), 'fontsize':18})
            ax.axis("equal")
            ax.set_xticks([], [])
            ax.set_yticks([], [])
```



Multiview Multidimensional Scaling (MVMDS)

MVMDS is a useful multiview dimensionaltiy reduction algorithm that allows the user to perform Multidimensional Scaling on multiple views at the same time.

[1]:
```
from mvlearn.datasets import load_UCImultifeature
from mvlearn.embed import MVMDS

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA

%matplotlib inline
```

Load Data

Data comes from UCI Digits Data. Contains 6 views and classifications of numbers 0-9

[2]:
```
# Load full dataset, labels not needed
Xs, y = load_UCImultifeature()
```

[3]:
```
# Check data
print(f'There are {len(Xs)} views.')
print(f'There are {Xs[0].shape[0]} observations')
print(f'The feature sizes are: {[X.shape[1] for X in Xs]}')
```

```
There are 6 views.
There are 2000 observations
The feature sizes are: [76, 216, 64, 240, 47, 6]
```

Plotting MVMDS vs PCA

Here we demonstrate the superior performance of MVMDS on multi-view data against the performance of PCA. To use all the views' data in PCA, we concatenate the views into a larger data matrix.

Examples of 10-class and 4 class data are shown. MVMDS learns principle components that are common across views, and end up spreading the data better.

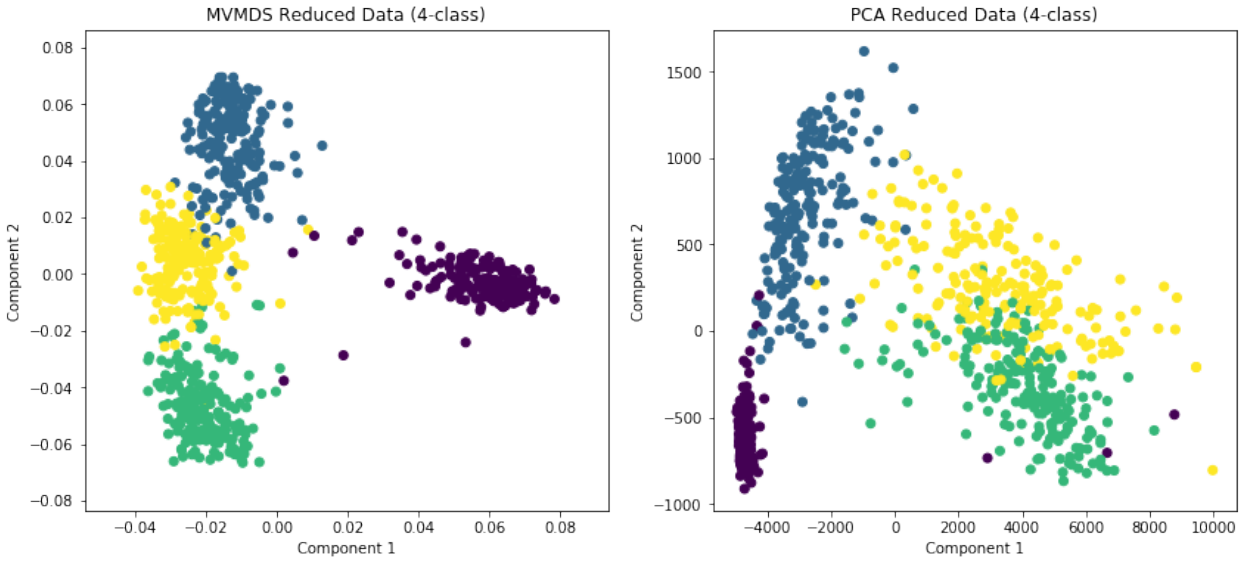
[4]:
```
# MVMDS reduction
mvmds = MVMDS(n_components=2)
Xs_mvmds_reduced = mvmds.fit_transform(Xs)

# Concatenate views then PCA for comparison
Xs_concat = Xs[0]
for X in Xs[1:]:
    Xs_concat = np.hstack((Xs_concat, X))
pca = PCA(n_components=2)
Xs_pca_reduced = pca.fit_transform(Xs_concat)
```

[5]:
```
fig, ax = plt.subplots(1, 2, figsize=(14,6))
ax[0].scatter(Xs_mvmds_reduced[:,0], Xs_mvmds_reduced[:,1], c=y)
ax[0].set_title("MVMDS Reduced Data (10-class)")
ax[0].set_xlabel("Component 1")
ax[0].set_ylabel("Component 2")
ax[1].scatter(Xs_pca_reduced[:,0], Xs_pca_reduced[:,1], c=y)
ax[1].set_title("PCA Reduced Data (10-class)")
ax[1].set_xlabel("Component 1")
ax[1].set_ylabel("Component 2")

plt.show()
```

[6]:
```
# 4-class data
Xs_4, y_4 = load_UCImultifeature(select_labeled=[0,1,2,3])
```

[7]:
```
# MVMDS reduction
mvmds = MVMDS(n_components=2)
Xs_mvmds_reduced = mvmds.fit_transform(Xs_4)

# Concatenate views then PCA for comparison
Xs_concat = Xs_4[0]
for X in Xs_4[1:]:
    Xs_concat = np.hstack((Xs_concat, X))
pca = PCA(n_components=2)
Xs_pca_reduced = pca.fit_transform(Xs_concat)
```

[8]:
```
fig, ax = plt.subplots(1, 2, figsize=(14,6))
ax[0].scatter(Xs_mvmds_reduced[:,0], Xs_mvmds_reduced[:,1], c=y_4)
ax[0].set_title("MVMDS Reduced Data (4-class)")
ax[0].set_xlabel("Component 1")
ax[0].set_ylabel("Component 2")
ax[1].scatter(Xs_pca_reduced[:,0], Xs_pca_reduced[:,1], c=y_4)
ax[1].set_title("PCA Reduced Data (4-class)")
ax[1].set_xlabel("Component 1")
ax[1].set_ylabel("Component 2")

plt.show()
```



Components of MVMDS Views Without Noise

Here we will take into account all of the views and perform MVMDS. This dataset does not contain noise and each view performs decently well in predicting the number. Therefore we will expect the common components created by MVMDS to create a strong representation of the data *(Note MVMDS only uses the fit_transform function to properly return the correct components)*

In the cell after, PCA on one view is shown for comparison. It can be seen that MVMDS seems to perform better in this instance.

Note: Each color represents a unique number class

[9]:
```
#perform mvmds
mvmds = MVMDS(n_components=5)
Components = mvmds.fit_transform(Xs)
```

[11]:
```
# Plot MVMDS images

plt.style.use('seaborn')

color_map = [sns.color_palette("Set2", 10)[int(i)] for i in y]

fig, axes = plt.subplots(4, 4, figsize = (12,12), sharey=True, sharex=True)

for i in range(4):
    for j in range(4):
        if i != j:
            axes[i,j].scatter(x = Components[:, i], y = Components[:, j], alpha = 1,␣
↪label = y, color = color_map)
        axes[3, j].set_xlabel(f'Component {j+1}')
    axes[i,0].set_ylabel(f'Component {i+1}')


ax = fig.add_subplot(111, frameon=False)
plt.tick_params(labelcolor='none', top=False, bottom=False, left=False, right=False)
ax.grid(False)
ax.set_title('First 4 MVMDS Components Computed With 6 Views (No Noise)')
```

[11]:
```
Text(0.5, 1.0, 'First 4 MVMDS Components Computed With 6 Views (No Noise)')
```

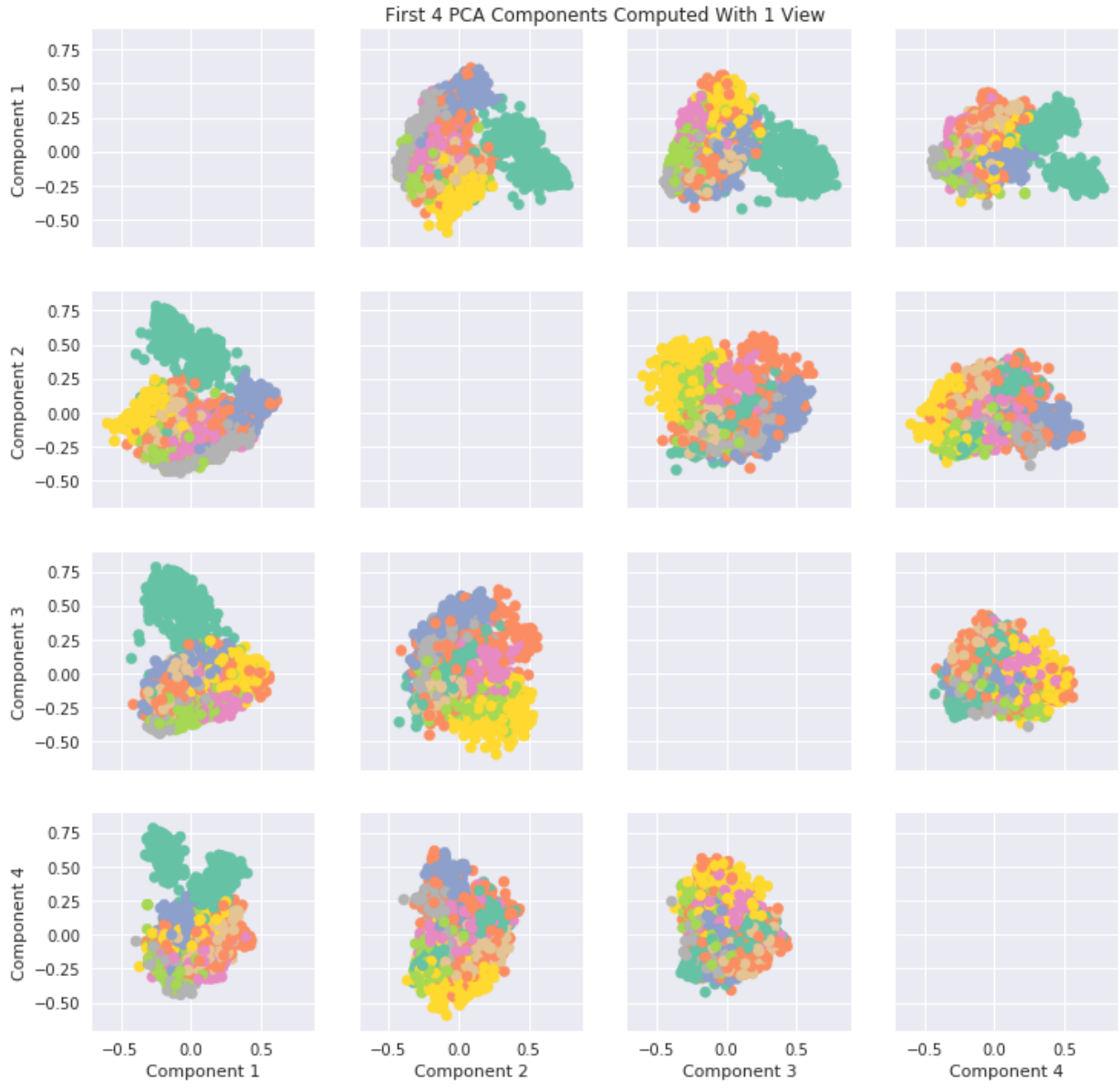
[12]:
```
#PCA Plots

pca = PCA(n_components=6)
pca_Components = pca.fit_transform(Xs[0])

fig, axes = plt.subplots(4, 4, figsize = (12,12), sharey=True, sharex=True)

for i in range(4):
    for j in range(4):
        if i != j:
            axes[i,j].scatter(x = pca_Components[:, i], y = pca_Components[:, j],␣
↪alpha = 1, label = y, color = color_map)
        axes[3, j].set_xlabel(f'Component {j+1}')
    axes[i,0].set_ylabel(f'Component {i+1}')
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(111, frameon=False)
plt.tick_params(labelcolor='none', top=False, bottom=False, left=False, right=False)
ax.grid(False)
ax.set_title('First 4 PCA Components Computed With 1 View')
```

[12]:
```
Text(0.5, 1.0, 'First 4 PCA Components Computed With 1 View')
```



MVMDS vs PCA

MVMDS is a useful multiview dimensionaltiy reduction algorithm that allows the user to perform Multidimensional Scaling on multiple views at the same time. In this notebook, we see how MVMDS performs in clustering randomly generated data and compare this to single-view classical multidimensional scaling which is equivalent to Principal Component Analysis (PCA).

Imports

[1]:
```
from mvlearn.datasets import load_UCImultifeature
from mvlearn.embed import MVMDS

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans
from sklearn.metrics.cluster import adjusted_rand_score

%matplotlib inline
```

```
C:\Users\arman\Anaconda3\envs\mvdev\lib\site-packages\sklearn\utils\deprecation.
↪py:144: FutureWarning: The sklearn.mixture.gaussian_mixture module is  deprecated␣
↪in version 0.22 and will be removed in version 0.24. The corresponding classes /␣
↪functions should instead be imported from sklearn.mixture. Anything that cannot be␣
↪imported from sklearn.mixture is now part of the private API.
  warnings.warn(message, FutureWarning)
```

Loading Data

Creates a dataset with 5 unique views. Each is represented by blobs distributed that are distributed around 6 random center points with a fixed variance.There are 100 points around each center point. The number of features of these blobs varies and the random states are assigned. Each view shares outcome values ranging from 0-5

[2]:
```
def data():

    N = 50
    D1 = 5
    D2 = 7
    D3 = 4

    np.random.seed(seed=5)
    first = np.random.rand(N,D1)
    second = np.random.rand(N,D2)
    third = np.random.rand(N,D3)
    random_views = [first, second, third]
    samp_views = [np.array([[1,4,0,6,2,3],
                            [2,5,7,1,4,3],
                            [9,8,5,4,5,6]]),
                  np.array([[2,6,2,6],
                            [9,2,7,3],
                            [9,6,5,2]])]

    first_wrong = np.random.rand(N,D1)
    second_wrong = np.random.rand(N-1,D1)
    wrong_views = [first_wrong, second_wrong]

    dep_views = [np.array([[1,2,3],[1,2,3],[1,2,3]]),
                 np.array([[1,2,3],[1,2,3],[1,2,3]])]

    return {'wrong_views' : wrong_views, 'dep_views' : dep_views,
```

(continues on next page)

(continued from previous page)

```
            'random_views' : random_views,
            'samp_views': samp_views}
```

[6]:
```
data = data
```

[13]:
```
from sklearn.metrics import euclidean_distances
```

[11]:
```
def john(data):
    print(data)
john
```

[11]:
```
<function __main__.john(data)>
```

[19]:
```
comp
```

[19]:
```
array([[-0.81330129,  0.07216426,  0.17407766],
       [ 0.34415456, -0.74042171,  0.69631062],
       [ 0.46914673,  0.66825745, -0.69631062]])
```

[20]:
```
comp2
```

[20]:
```
array([[-0.81330129,  0.07216426,  0.57735027],
       [ 0.34415456, -0.74042171,  0.57735027],
       [ 0.46914673,  0.66825745,  0.57735027]])
```

[21]:
```
mvmds = MVMDS(len(data()['samp_views'][0]))
comp = mvmds.fit_transform(data()['samp_views'])
comp2 = np.array([[-0.81330129,  0.07216426,  0.17407766],
       [0.34415456, -0.74042171,  0.69631062],
       [0.46914673,  0.66825745, -0.69631062]])

for i in range(comp.shape[0]):
    for j in range(comp.shape[1]):
        assert comp[i,j]-comp2[i,j] < .000001
```

[2]:
```
p = np.array([100,100,100,100,100,100])

#creates the blobs
j = make_blobs(n_features=12,n_samples=p, cluster_std= 4,random_state= 1)
k = make_blobs(n_features = 27,n_samples = p,cluster_std = 3,random_state=23)
l = make_blobs(n_features = 22,n_samples = p,cluster_std = 5,random_state=35)
m = make_blobs(n_features = 32,n_samples = p,cluster_std = 5,random_state=52)
n = make_blobs(n_features = 15,n_samples = p,cluster_std = 7,random_state=2)


v1 = j[0]
v2 = k[0]
v3 = l[0]
v4 = m[0]
v5 = n[0]

Views = [v1,v2,v3,v4,v5]
```

[3]:
```
# This creates a single-view dataset by concatenating the multiple views as features
↪of the first view (Naive multi-view)

arrays = []

for i in [j,k,l,m,n]:
    df = pd.DataFrame(i[0])
    df['Class'] = i[1]
    df = df.sort_values(by = ['Class'])
    y = np.array(df['Class'])
    df = df.drop(['Class'],axis = 1)
    arrays.append(np.array(df))

Views = arrays

Views_concat = np.hstack((arrays[0],arrays[1],arrays[2],arrays[3],arrays[4]))
```

Plot original Data

As you can see. The blobs are not distinguishable in 2-Dimensions

[4]:
```
ax = plt.subplot(111)
plt.scatter(v1[:,0],v1[:,1],c = y)
plt.title('First Two Features of First View')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
```

[4]:
```
Text(0, 0.5, 'Feature 2')
```

MVMDS Views Without Noise

Here we will take into account all of the views and perform MVMDS. This dataset does not contain noise and each view performs decently well in predicting the class. Therefore we will expect the common components created by MVMDS to create a strong representation of the data *(Note MVMDS only uses the fit_transform function to properly return the correct components)*

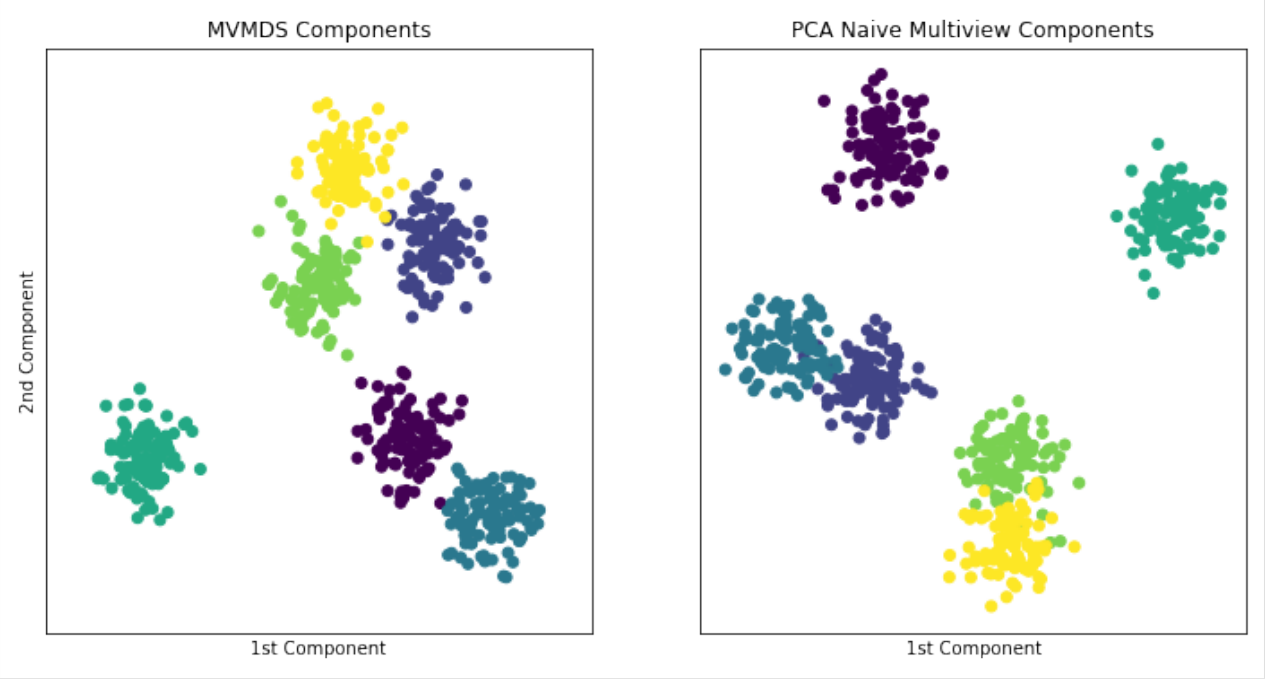
In the cell after, PCA on the concatenated single-view is shown for comparison. It can be seen that MVMDS performs better in this instance.

Note: Each color represents a unique number class

[16]:
```
#Fits MVMDS
mvmds = MVMDS(n_components=2,distance=False)
fit = mvmds.fit_transform(Views)

#Fits PCA
pca = PCA(n_components=2)
fit2 = pca.fit_transform(Views_concat)
```

[6]:
```
#Fits K-Means to MVMDS for cluster comparison
kmeans = KMeans(n_clusters=6, random_state=0).fit(fit)
labels1 = kmeans.labels_

fig, axes = plt.subplots(1,2, figsize=(12,6))

#Plots MVMDS components
axes[0].scatter(fit[:,0],fit[:,1],c = y)
axes[0].set_title('MVMDS Components')
axes[0].set_xlabel('1st Component')
axes[0].set_ylabel('2nd Component')
axes[0].set_xticks([])
axes[0].set_yticks([])

#Fits K-Means to PCA for cluster comparison
kmeans = KMeans(n_clusters=6, random_state=0).fit(fit2)
labels2 = kmeans.labels_

#Plots PCA components
axes[1].scatter(fit2[:,0],fit2[:,1],c = y)
axes[1].set_title('PCA Naive Multiview Components')
axes[1].set_xlabel('1st Component')
axes[1].set_xticks([])
axes[1].set_yticks([])

#Comparison of ARI scores

score1 = adjusted_rand_score(labels1,y)
score2 = adjusted_rand_score(labels2,y)

print('MVMDS has an ARI score of ' + str(score1) + '. while PCA has an ARI score of '␣
↪+ str(score2) +
      '. \nTherefore we can say MVMDS performs better in this instance')
```

```
MVMDS has an ARI score of 0.9840270979888018. while PCA has an ARI score of 0.
↪9344335788597602.
Therefore we can say MVMDS performs better in this instance
```



MVMDS Views With Noise

Here we will create a new variable with multiple views. This variable will contain the same 5 views from before but a 6th view of strictly noise will be added to the dataset. The concatenated single-view dataset will also have this noisy view. We can expect for the common components created by MVMDS to be less representative of the data due to the substantial noise.

As we can see compared to previous cells, the noisy MVMDS components performs worse than the MVMDS components done on views without noise. When compared to PCA on the concatenated single-view with noise, MVMDS performs worse.

Note: Each color represents a unique number class

[7]:
```
noisy_view = np.random.rand(n[0].shape[0],n[0].shape[1])

Views_Noise = Views
Views_Noise.append(noisy_view)
Views_concat_Noise = np.hstack((Views_concat,noisy_view))

#Fits MVMDS
mvmds = MVMDS(n_components=2)
fit = mvmds.fit_transform(Views_Noise)

#Fits PCA
pca = PCA(n_components=2)
fit2 = pca.fit_transform(Views_concat_Noise)
```

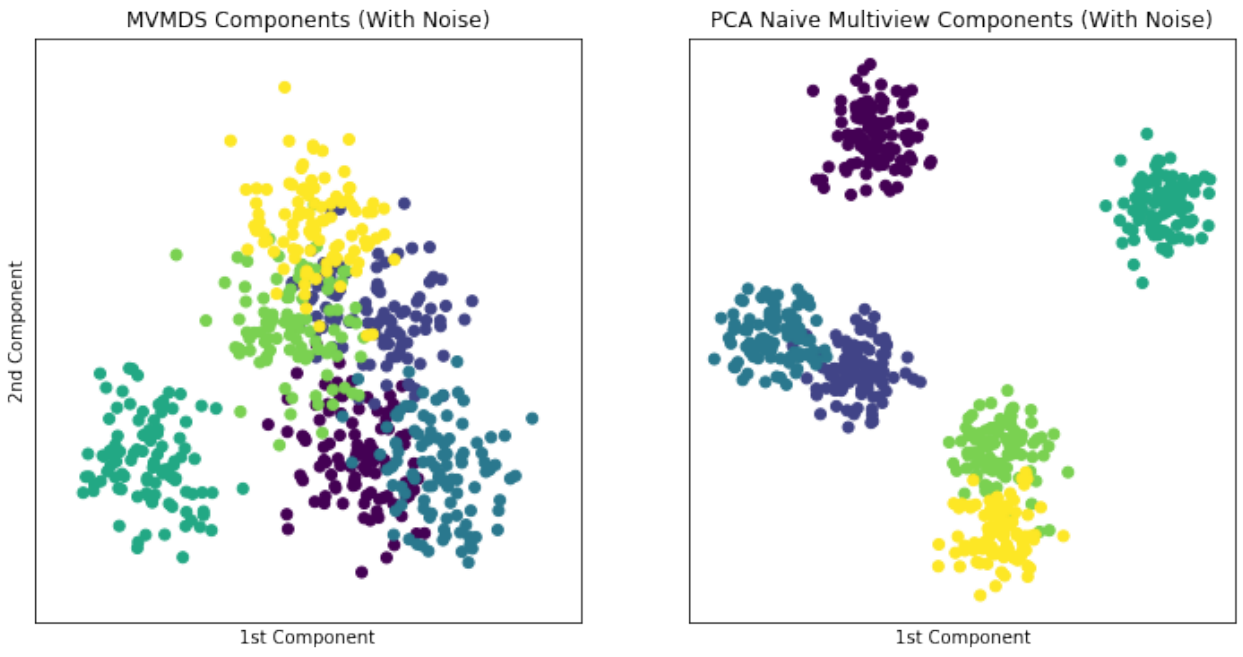
[8]:
```
#Fits K-Means to MVMDS for cluster comparison
kmeans = KMeans(n_clusters=6, random_state=0).fit(fit)
labels1_noise = kmeans.labels_

fig, axes = plt.subplots(1,2, figsize=(12,6))
```

(continues on next page)

(continued from previous page)

```
#Plots MVMDS components
axes[0].scatter(fit[:,0],fit[:,1],c = y)
axes[0].set_title('MVMDS Components (With Noise)')
axes[0].set_xlabel('1st Component')
axes[0].set_ylabel('2nd Component')
axes[0].set_xticks([])
axes[0].set_yticks([])

#Fits K-Means to PCA for cluster comparison
kmeans = KMeans(n_clusters=6, random_state=0).fit(fit2)
labels2_noise = kmeans.labels_

#Plots PCA components
axes[1].scatter(fit2[:,0],fit2[:,1],c = y)
axes[1].set_title('PCA Naive Multiview Components (With Noise)')
axes[1].set_xlabel('1st Component')
axes[1].set_xticks([])
axes[1].set_yticks([])

#Comparison of ARI scores

score1_noise = adjusted_rand_score(labels1_noise,y)
score2_noise = adjusted_rand_score(labels2_noise,y)

print('MVMDS has an ARI score of ' + str(score1_noise) + '. while PCA has an ARI␣
↪score of ' + str(score2_noise) +
      '. \nTherefore we can say PCA performs better in this instance.')
```

```
MVMDS has an ARI score of 0.6004142756032063. while PCA has an ARI score of 0.
↪9344335788597602.
Therefore we can say PCA performs better in this instance.
```



Omnibus Embedding for Multiview Data

This demo shows you how to run Omnibus Embedding on multiview data. Omnibus embedding is originally a multi-graph algorithm. The purpose of omnibus embedding is to find a Euclidean representation (latent position) of multiple graphs. The embedded latent positions live in the same canonical space allowing us to easily compare the embedded graphs to each other without aligning results. You can read more about both the implementation of Omnibus embedding used and the algorithm itself from the graspy package.

Unlike graphs however, multiview data can consist of arbitrary arrays of different dimensions. This represents an additional challenge of comparing the information contained in each view. An effective solution is to first compute the **dissimilarity matrix** for each view. Assuming each view has n samples, we will be left with an n x n matrix for each view. If the distance function used to compute these matrices is symmetric, the dissimilarity matrices will also be symmetric and we are left with "graph-like" objects. Omnibus embedding can then be applied and the resulting embeddings show whether views give similar or different information.

Below, we show the results of Omnibus embedding on multiview data when the two views are very similar and very different. We then apply Omnibus to two different views in the UCI handwritten digits dataset.

[1]:
```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
from mvlearn.embed import omnibus
```

Case 1: two identical views

For this setting, we generate two identical numpy matrices as our views. Since the information is identical in each view, the resulting embedded views should also be similar. We run omnibus on default parameters.

[2]:
```
# 100 x 50 matrices
X_1 = np.random.rand(100, 50)
X_2 = X_1.copy()

Xs = [X_1, X_2]

# Running omnibus
embedder = omnibus.Omnibus()
embeddings = embedder.fit_transform(Xs)
```

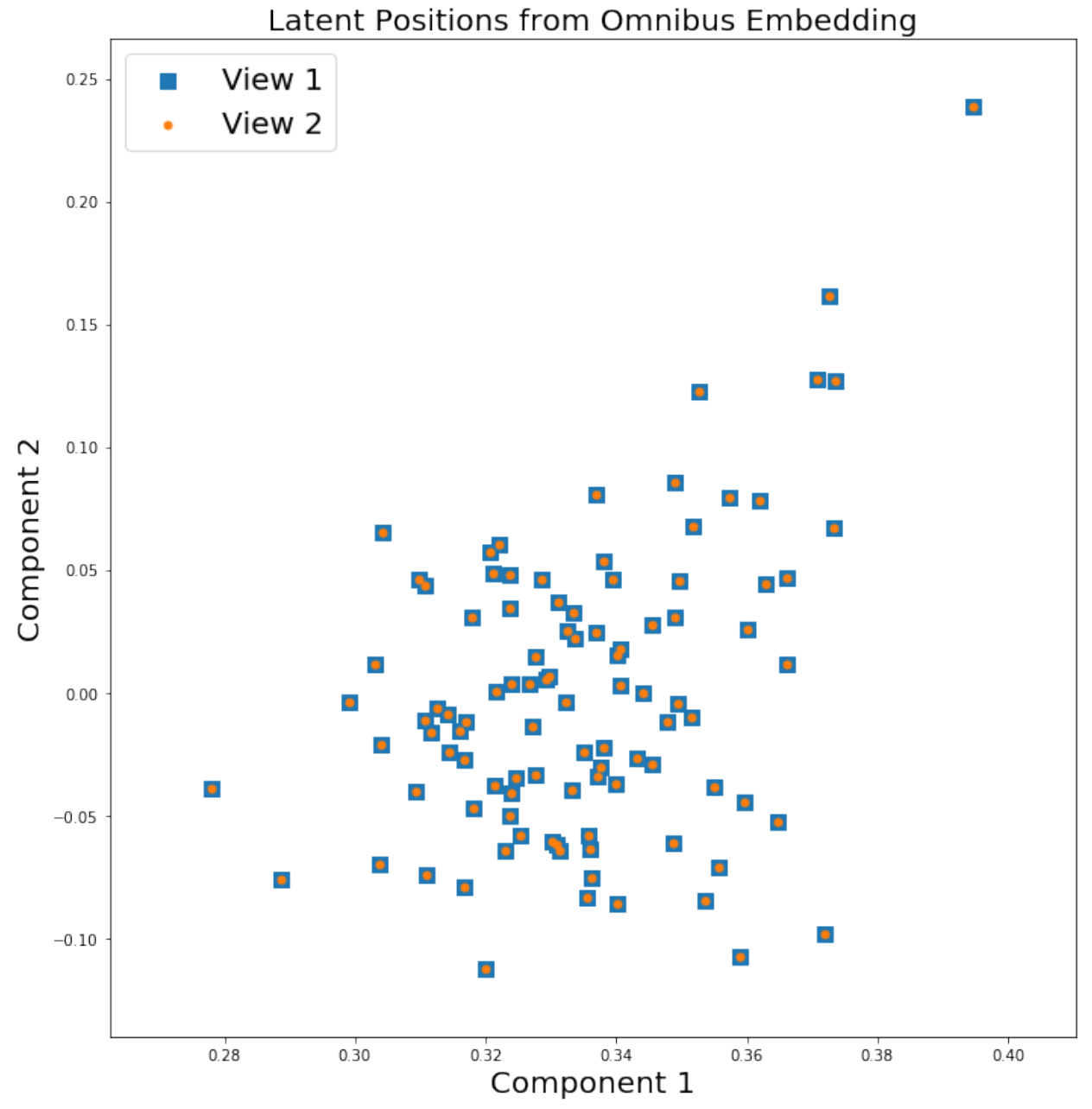
Visualizing the results

[3]:
```
Xhat1, Xhat2 = embeddings

fig, ax = plt.subplots(figsize=(10, 10))
ct = ax.scatter(Xhat1[:, 0], Xhat1[:, 1], marker='s', label = 'View 1', cmap = "tab10
↪", s = 100)
ax.scatter(Xhat2[:, 0], Xhat2[:, 1], marker='.', label= 'View 2', cmap = "tab10", s =␣
↪100)
plt.legend(fontsize=20)

# Plot lines between matched pairs of points
for i in range(50):
    idx = np.random.randint(len(Xhat1), size=1)
    ax.plot([Xhat1[idx, 0], Xhat2[idx, 0]], [Xhat1[idx, 1], Xhat2[idx, 1]], 'black',␣
↪alpha = 0.15)
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("Component 1", fontsize=20)
plt.ylabel("Component 2", fontsize=20)
plt.tight_layout()
ax.set_title('Latent Positions from Omnibus Embedding', fontsize=20)
plt.show()
```



As expected, the embeddings are identical since the views are the same.

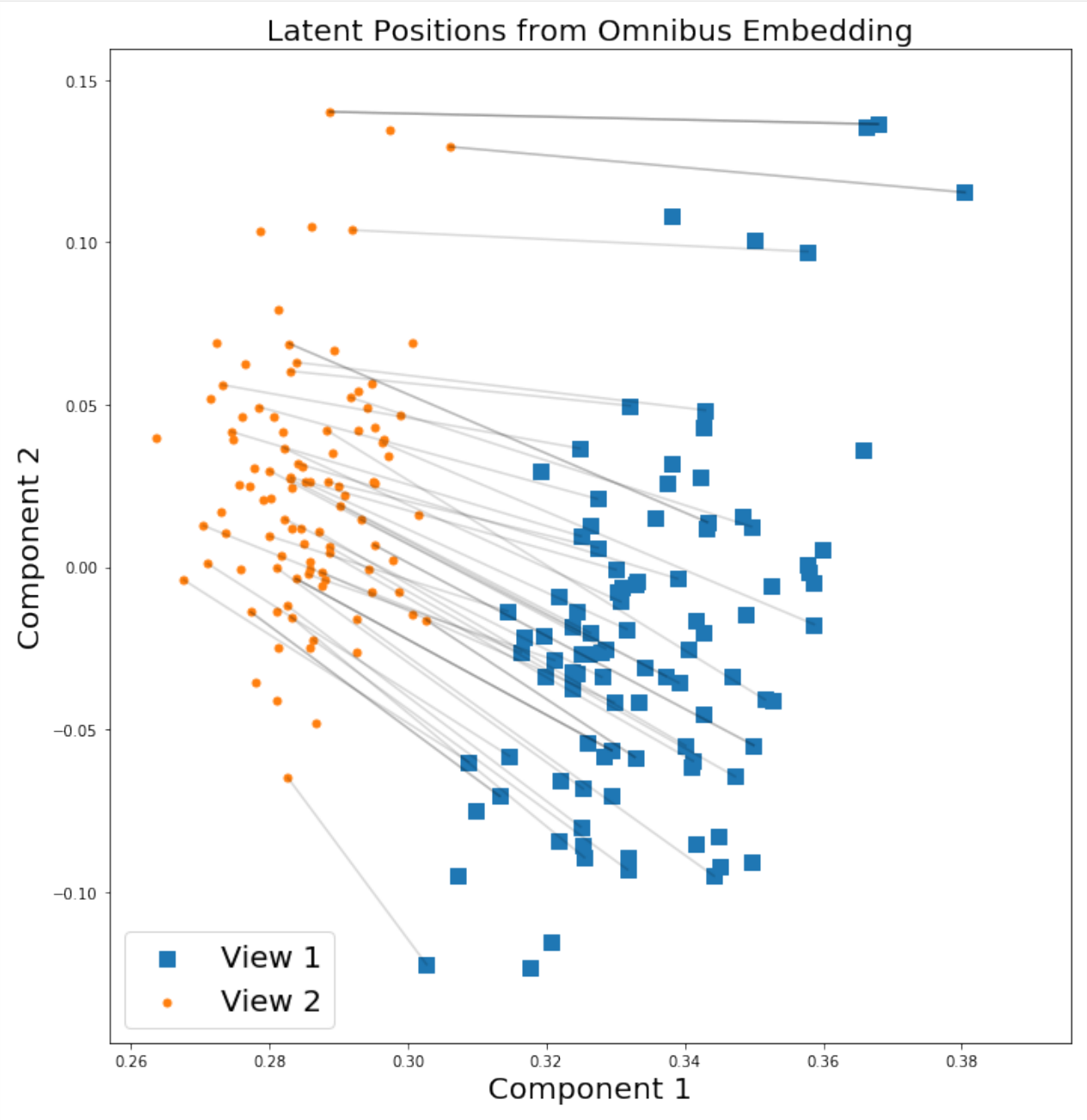
Case 2: two unidentical views

Now let's see what happens when the views are not identical.

[4]:
```
X_1 = np.random.rand(100, 50)
# Second view has different number of features
X_2 = np.random.rand(100, 100)

Xs = [X_1, X_2]

# Running omnibus
embedder = omnibus.Omnibus()
embeddings = embedder.fit_transform(Xs)
```

Visualizing the results

[5]:
```
Xhat1, Xhat2 = embeddings

fig, ax = plt.subplots(figsize=(10, 10))
ct = ax.scatter(Xhat1[:, 0], Xhat1[:, 1], marker='s', label = 'View 1', cmap = "tab10
↪", s = 100)
ax.scatter(Xhat2[:, 0], Xhat2[:, 1], marker='.', label= 'View 2', cmap = "tab10", s =␣
↪100)
plt.legend(fontsize=20)

# Plot lines between matched pairs of points
for i in range(50):
    idx = np.random.randint(len(Xhat1), size=1)
    ax.plot([Xhat1[idx, 0], Xhat2[idx, 0]], [Xhat1[idx, 1], Xhat2[idx, 1]], 'black',␣
↪alpha = 0.15)
plt.xlabel("Component 1", fontsize=20)
plt.ylabel("Component 2", fontsize=20)
plt.tight_layout()
ax.set_title('Latent Positions from Omnibus Embedding', fontsize=20)
plt.show()
```



Here, we see that the views are clearly separated suggeseting the views represent different information. Lines are drawn between corresponding samples in the two views.

UCI Digits Dataset

Finally, we run Omnibus on the UCI Multiple Features Digits Dataset. We use the Fourier coefficient and profile correlation views (View 1 and 2 respectively).

[7]:
```
from mvlearn.datasets.base import load_UCImultifeature

full_data, full_labels = load_UCImultifeature()
```

(continues on next page)

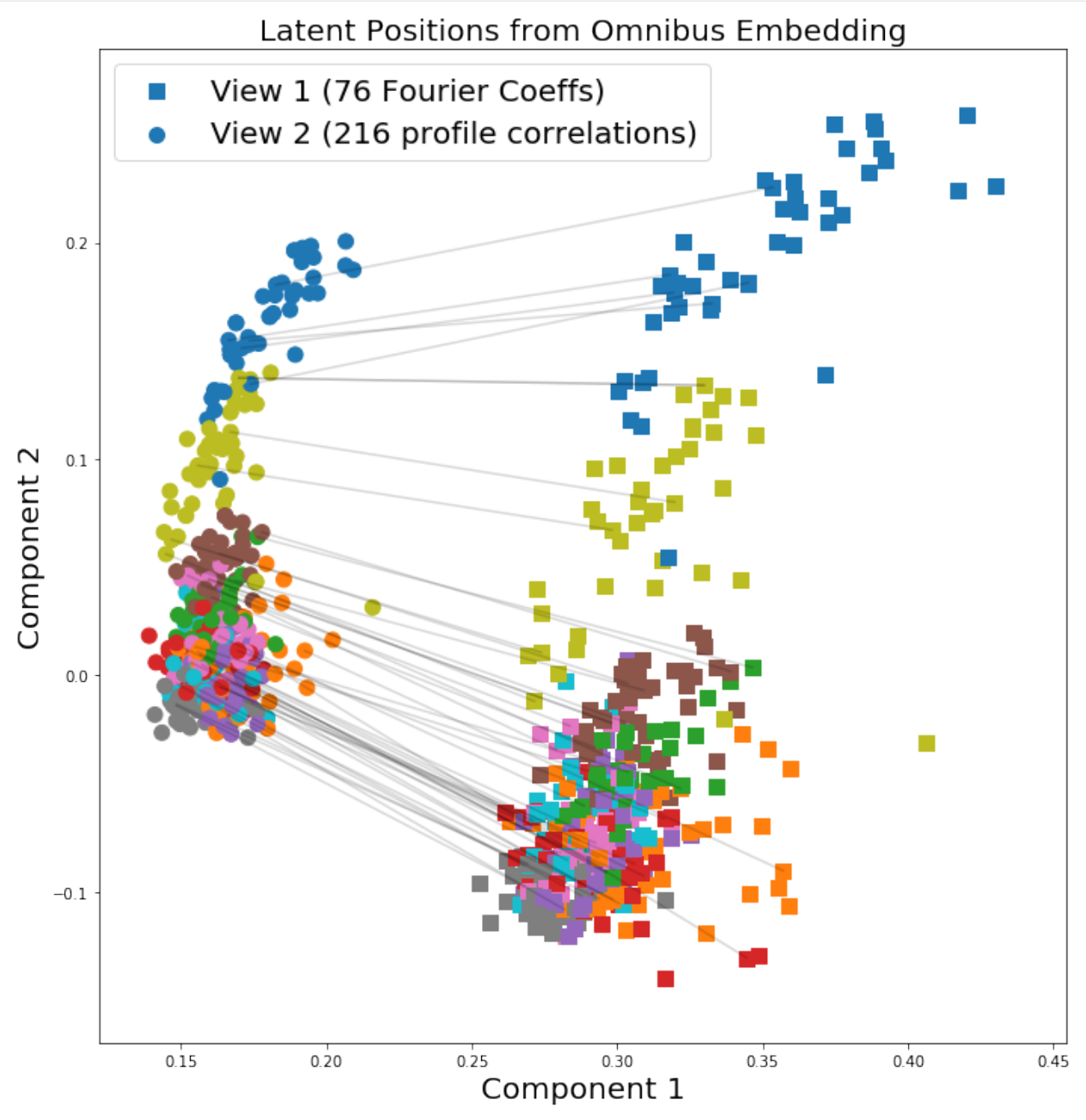
(continued from previous page)

```
view_1 = full_data[0]
view_2 = full_data[1]

Xs = [view_1, view_2]

# Running omnibus
embedder = omnibus.Omnibus()
embeddings = embedder.fit_transform(Xs)
```

Visualizing the results

This time, the points in the plot are colored by digit (0-9). The marker symbols denote which view each sample is from. We randomly plot 500 samples to make the plot more readable.

[8]:
```
Xhat1, Xhat2 = embeddings

n = 500
idxs = np.random.randint(len(Xhat1), size=n)
Xhat1 = Xhat1[idxs, :]
Xhat2 = Xhat2[idxs, :]
labels = full_labels[idxs]


fig, ax = plt.subplots(figsize=(10, 10))
ct = ax.scatter(Xhat1[:, 0], Xhat1[:, 1], marker='s', label = 'View 1 (76 Fourier␣
↪Coeffs)', c = labels, cmap = "tab10", s = 100)
ax.scatter(Xhat2[:, 0], Xhat2[:, 1], marker='o', label= 'View 2 (216 profile␣
↪correlations)', c = labels, cmap = "tab10", s = 100)
plt.legend(fontsize=20)
#fig.colorbar(ct)

# Plot lines between matched pairs of points
for i in range(50):
    idx = np.random.randint(len(Xhat1), size=1)
    ax.plot([Xhat1[idx, 0], Xhat2[idx, 0]], [Xhat1[idx, 1], Xhat2[idx, 1]], 'black',␣
↪alpha = 0.15)
plt.xlabel("Component 1", fontsize=20)
plt.ylabel("Component 2", fontsize=20)
plt.tight_layout()
ax.set_title('Latent Positions from Omnibus Embedding', fontsize=20)
plt.show()
```



SplitAE Embeddings on multiview MNIST data

[ ]:
```
!pip3 install pillow==6.2.2
!pip3 install torchvision==0.4.2
```

[5]:
```
import matplotlib.pyplot
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
from torchvision import datasets
```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
import numpy as np
import PIL

#tsnecuda is a bit harder to install, if you want to use MulticoreTSNE instead␣
↪(sklearn is too slow)
#then uncomment the below MulticoreTSNE line, comment out the tsnecuda line, and␣
↪replace
#all TSNE() lines with TSNE(n_jobs=12), where 12 is replaced with the number of cores␣
↪on your machine

#from MulticoreTSNE import MulticoreTSNE as TSNE
from tsnecuda import TSNE
from mvlearn.embed import SplitAE
```

[6]:
```
# Setup plotting

%matplotlib inline
plt.style.use("default")
%config InlineBackend.figure_format = 'svg'
```

Let's make a simple two view dataset based on MNIST as described in http://proceedings.mlr.press/v37/wangb15.pdf .

The "underlying data" of our views is a digit from 0-9 – e.g. "2" or "7" or "9".

The first view of this underlying data is a random MNIST image with the correct digit, rotated randomly +- 45 degrees.

The second view of this underlying data is another random MNIST image (not rotated) with the correct digit, but with the addition of uniform noise from [0,1]

An example point of our data is:

- view1: an MNIST image with the label "9"
- view2: a different MNIST image with the label "9" with noise added.

[7]:
```
class NoisyMnist(Dataset):

    MNIST_MEAN, MNIST_STD = (0.1307, 0.3081)

    def __init__(self, train=True):
        super().__init__()
        self.mnistDataset = datasets.MNIST("./mnist", train=train, download=True)

    def __len__(self):
        return len(self.mnistDataset)

    def __getitem__(self, idx):
        randomIndex = lambda: np.random.randint(len(self.mnistDataset))
        image1, label1 = self.mnistDataset[idx]
        image2, label2 = self.mnistDataset[randomIndex()]
        while not label1 == label2:
            image2, label2 = self.mnistDataset[randomIndex()]

        image1 = torchvision.transforms.RandomRotation((-45, 45), resample=PIL.Image.
↪BICUBIC)(image1)
        #image2 = torchvision.transforms.RandomRotation((-45, 45), resample=PIL.Image.
↪BICUBIC)(image2)
```

(continues on next page)

(continued from previous page)

```
        image1 = np.array(image1) / 255
        image2 = np.array(image2) / 255

        image2 = np.clip(image2 + np.random.uniform(0, 1, size=image2.shape), 0, 1) #␣
↪add noise to the view2 image

        # standardize both images
        image1 = (image1 - self.MNIST_MEAN) / self.MNIST_STD
        image2 = (image2 - (self.MNIST_MEAN+0.447)) / self.MNIST_STD

        image1 = torch.FloatTensor(image1).unsqueeze(0) # image1 is view1
        image2 = torch.FloatTensor(image2).unsqueeze(0) # image2 is view2

        return (image1, image2, label1)
```

Let's look at this datset we made. The first row is view1 and the second row is the corresponding view2.

[9]:
```
dataset = NoisyMnist()
print("Dataset length is", len(dataset))
dataloader = DataLoader(dataset, batch_size=8, shuffle=True, num_workers=8)
view1, view2, labels = next(iter(dataloader))

view1Row = torch.cat([*view1.squeeze()], dim=1)
view2Row = torch.cat([*view2.squeeze()], dim=1)
# make between 0 and 1 again:
view1Row = (view1Row - torch.min(view1Row)) / (torch.max(view1Row) - torch.
↪min(view1Row))
view2Row = (view2Row - torch.min(view2Row)) / (torch.max(view2Row) - torch.
↪min(view2Row))
plt.imshow(torch.cat([view1Row, view2Row], dim=0))
```

```
Dataset length is 60000
```

[9]:
```
<matplotlib.image.AxesImage at 0x131f2cdd8>
```

Sklearn API doesn't use Dataloaders (which hampers data augmentation :( ) so let's get our dataset into a different format. Each view will be an array of the shape (nSamples, nFeatures). We will do the same for the test dataset.

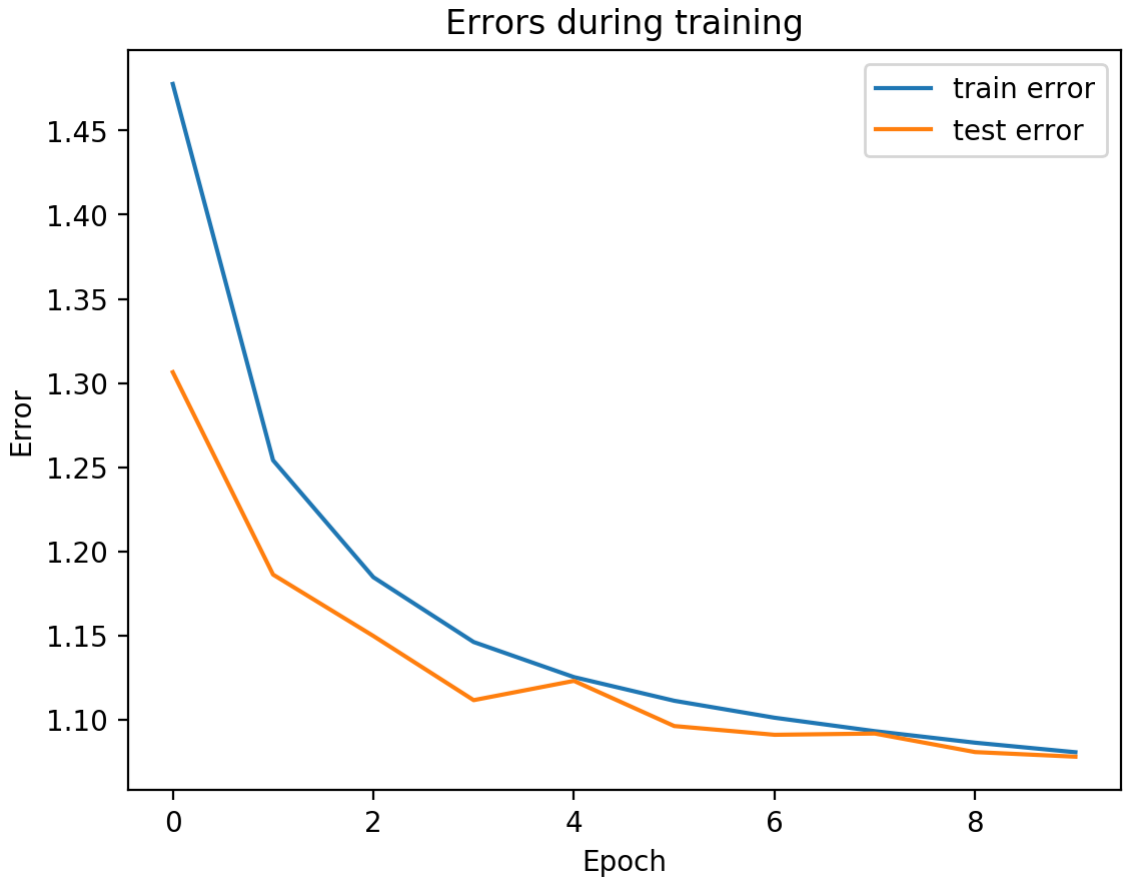
[6]:
```
# since batch_size=len(dataset), we get the full dataset with one next(iter(dataset))␣
↪call
dataloader = DataLoader(dataset, batch_size=len(dataset), shuffle=True, num_workers=8)
view1, view2, labels = next(iter(dataloader))
view1 = view1.view(view1.shape[0], -1)
view2 = view2.view(view2.shape[0], -1)

testDataset = NoisyMnist(train=False)
print("Test dataset length is", len(testDataset))
testDataloader = DataLoader(testDataset, batch_size=10000, shuffle=True, num_
↪workers=8)
testView1, testView2, testLabels = next(iter(testDataloader))
testView1 = testView1.view(testView1.shape[0], -1)
testView2 = testView2.view(testView2.shape[0], -1)
```

```
Test dataset length is 10000
```

SplitAE does two things. It creates a shared embedding for view1 and view2. And it allows predicting view2 from view1. The autoencoder network takes in view1 as input, squeezes it into a low-dimensional representation, and then from that low-dimensional representation (the embedding), it tries to recreate view1 and predict view2. Let's see that:

[19]:
```
splitae = SplitAE(hidden_size=1024, num_hidden_layers=2, embed_size=10, training_
↪epochs=10, batch_size=128,
                  learning_rate=0.001, print_info=False, print_graph=True)
splitae.fit([view1, view2], validation_Xs=[testView1, testView2])
# if the named parameter validationXs is passed with held-out data, then .fit will␣
↪print validation error as well.
```

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```



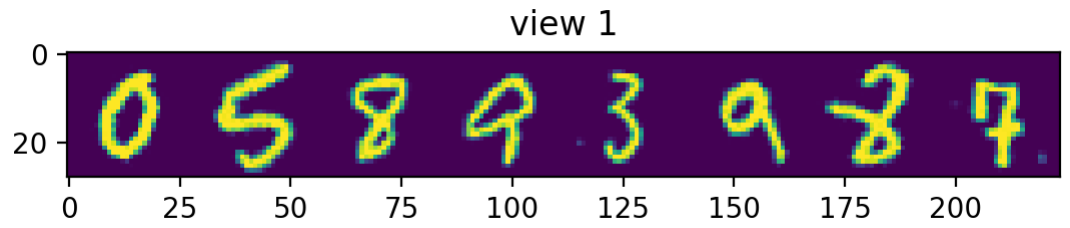
We can see from the graph that test error did not diverge from train error, which means we're not overfitting, which is good! Let's see the actual view1 recreation and the view2 prediction on test data:

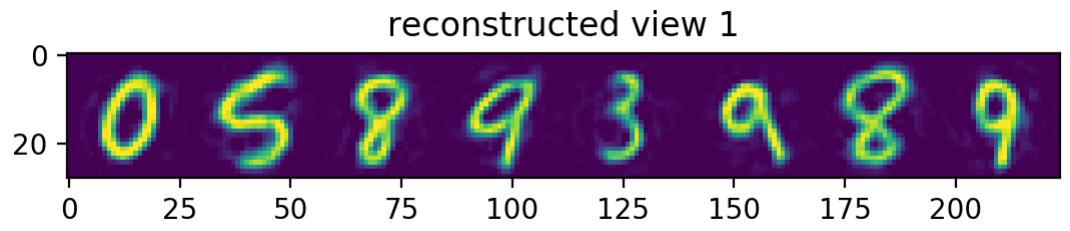
[20]:
```
MNIST_MEAN, MNIST_STD = (0.1307, 0.3081)
testEmbed, testView1Reconstruction, testView2Prediction = splitae.
↪transform([testView1, testView2])
numImages = 8
randIndices = np.random.choice(range(len(testDataset)), numImages, replace=False)
def plotRow(title, view):
    samples = view[randIndices].reshape(-1, 28, 28)
    row = np.concatenate([*samples], axis=1)
    row = np.clip(row * MNIST_STD + MNIST_MEAN, 0, 1) #denormalize
    plt.imshow(row)
    plt.title(title)
```

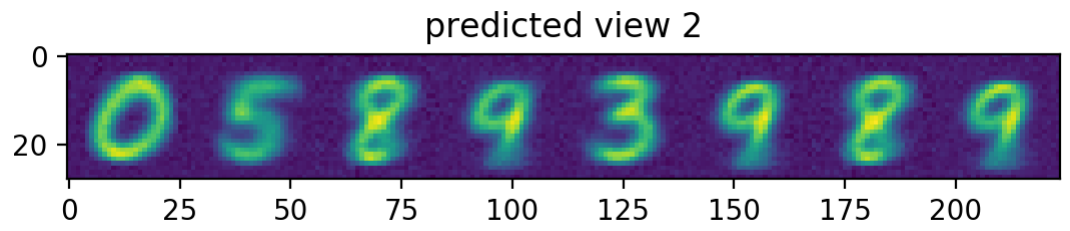
(continues on next page)

(continued from previous page)

```
    plt.show()
plotRow("view 1", testView1)
plotRow("reconstructed view 1", testView1Reconstruction)
plotRow("predicted view 2", testView2Prediction)
```







Notice the view 2 predictions. Had our view2 images been randomly rotated, the predictions would have a hazy circle, since the best guess would be the mean of all the rotated digits. Since we don't rotate our view2 images, we instead get something that's only a bit hazy around the edges – corresonding to the mean of all the non-rotated digits.

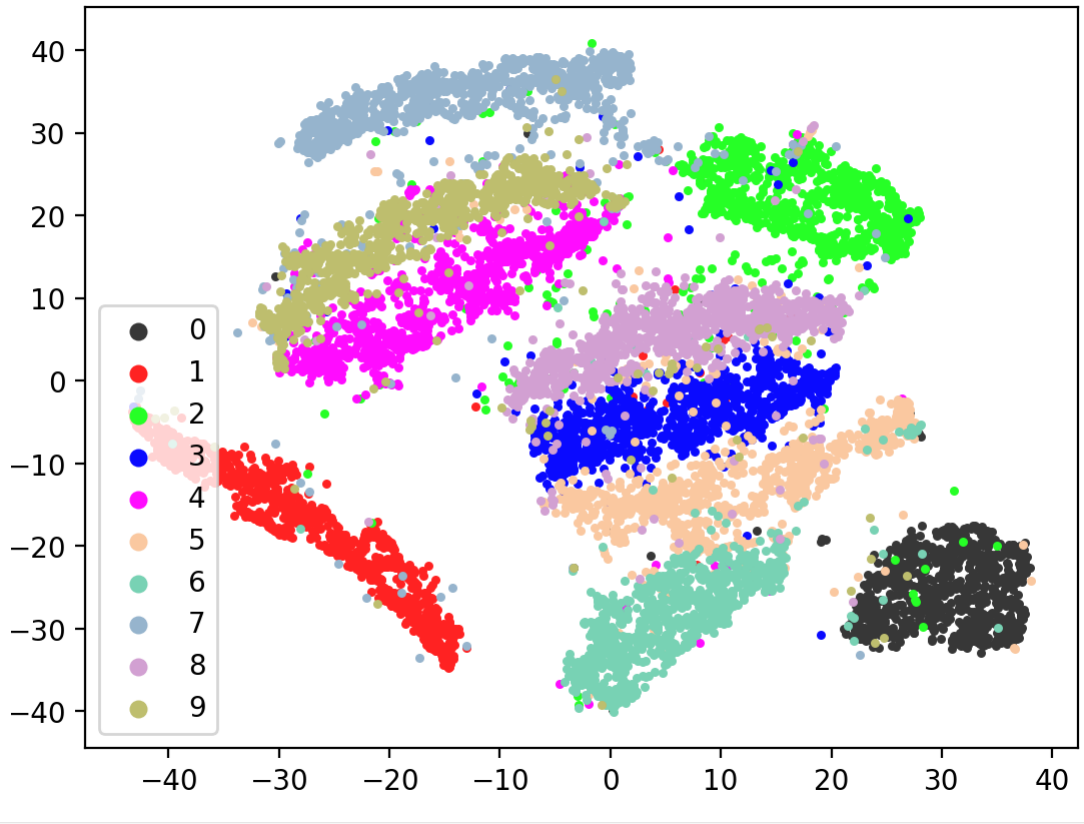
Next let's visualize our 20d test embeddings with T-SNE and see if they represent our original underlying representation – the digits from 0-9 – of which we made two views of. In the perfect scenario, each of the 10,000 vectors of our test embedding would be one of ten vectors, representing the digits from 0-9. (Our network wouldn't do this, as it tries to reconstruct each unique view1 image exactly). In lieu of this we can hope for embedding vectors corresponding to the same digits to be closer together.

[24]:
```
%config InlineBackend.figure_format = 'retina'

tsne = TSNE()
tsneEmbeddings = tsne.fit_transform(testEmbed)

def plot2DEmbeddings(embeddings, labels):
    pointColors = []
    origColors = [[55, 55, 55], [255, 34, 34], [38, 255, 38], [10, 10, 255], [255, 12,
↪ 255], [250, 200, 160], [120, 210, 180], [150, 180, 205], [210, 160, 210], [190,␣
↪190, 110]]
    origColors = (np.array(origColors)) / 255
    for l in labels.cpu().numpy():
        pointColors.append(tuple(origColors[l].tolist()))
```

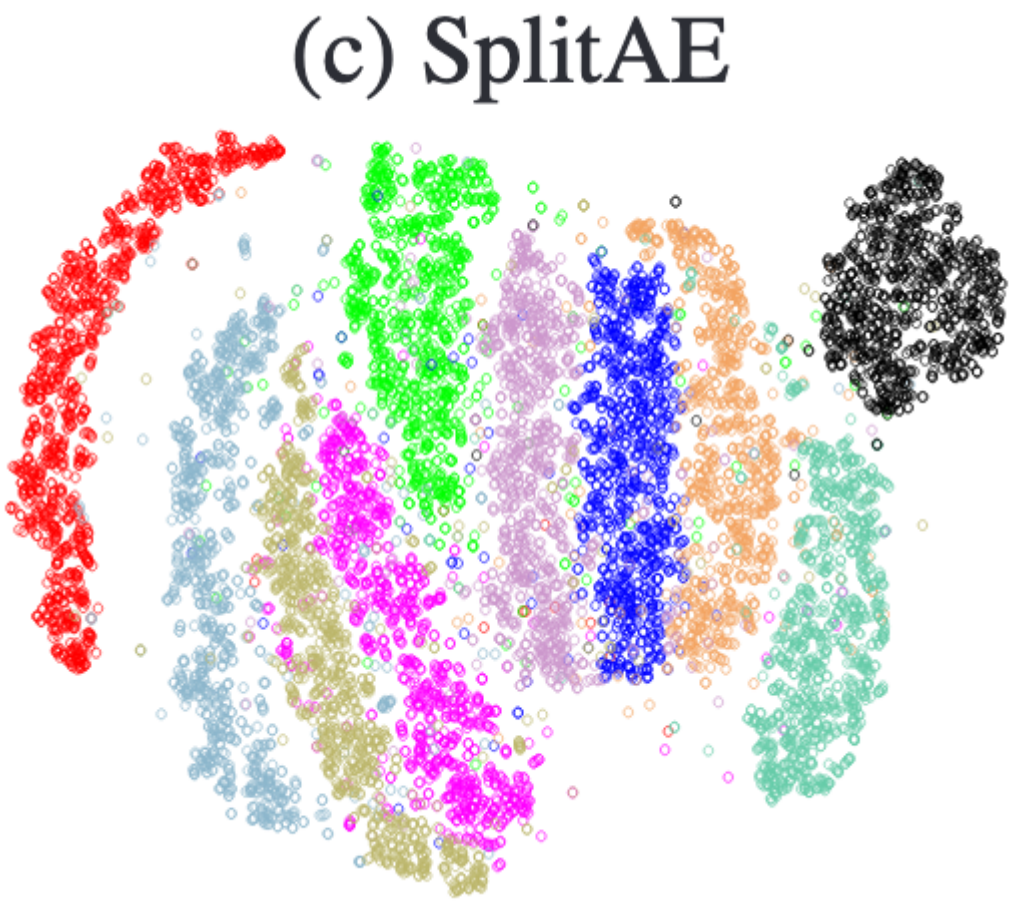
(continues on next page)

(continued from previous page)

```
    fig, ax = plt.subplots()
    #scatter = ax.scatter(*tsneEmbeddings.transpose(), c=pointColors, s=5)
    for i, label in enumerate(np.unique(labels)):
        idxs = np.where(testLabels == label)
        ax.scatter(embeddings[idxs][:, 0], embeddings[idxs][:, 1], c=[origColors[i]],␣
↪label=i, s=5)

    legend = plt.legend(loc="lower left")
    for handle in legend.legendHandles:
        handle.set_sizes([30.0])
    plt.show()

plot2DEmbeddings(tsneEmbeddings, testLabels)
```

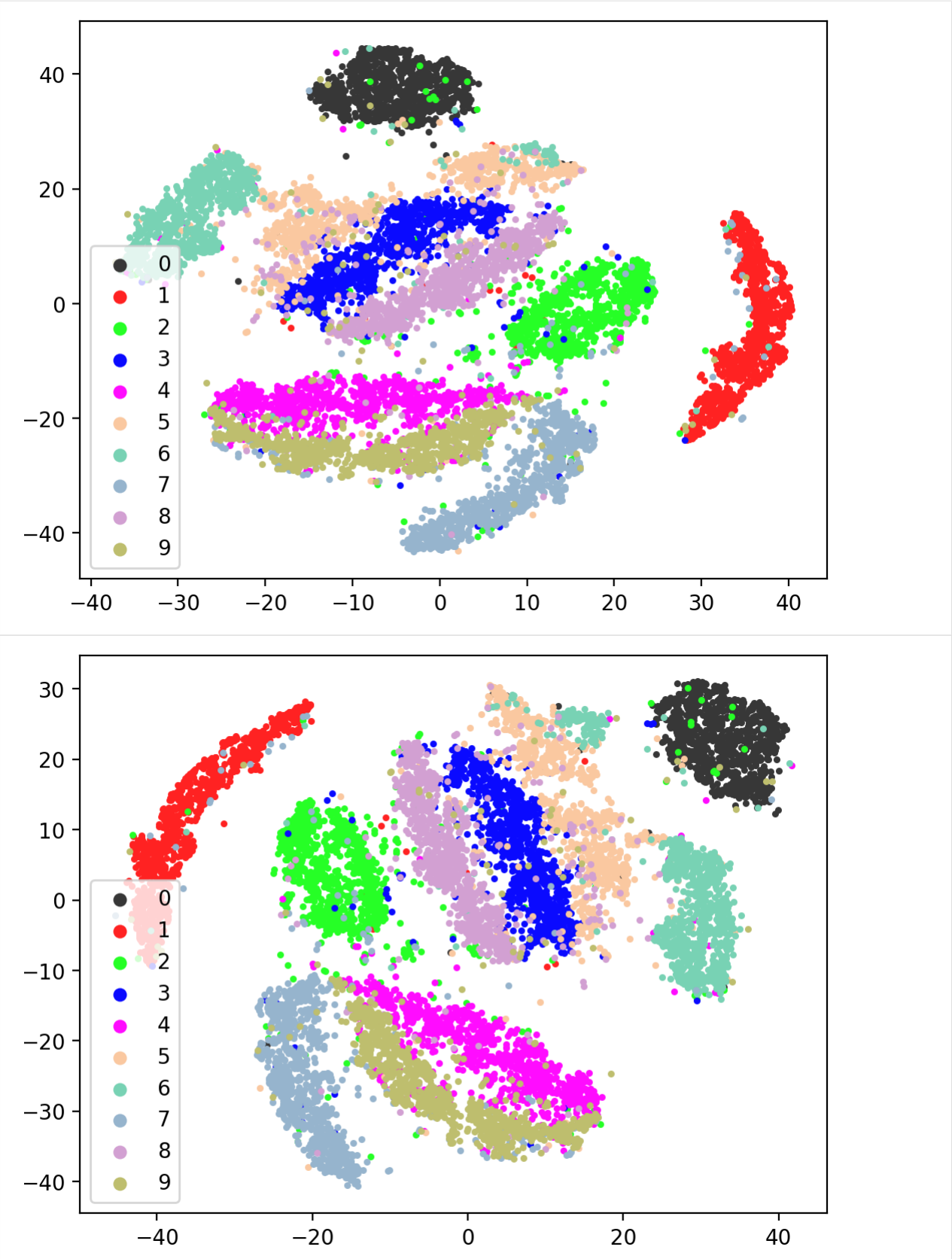


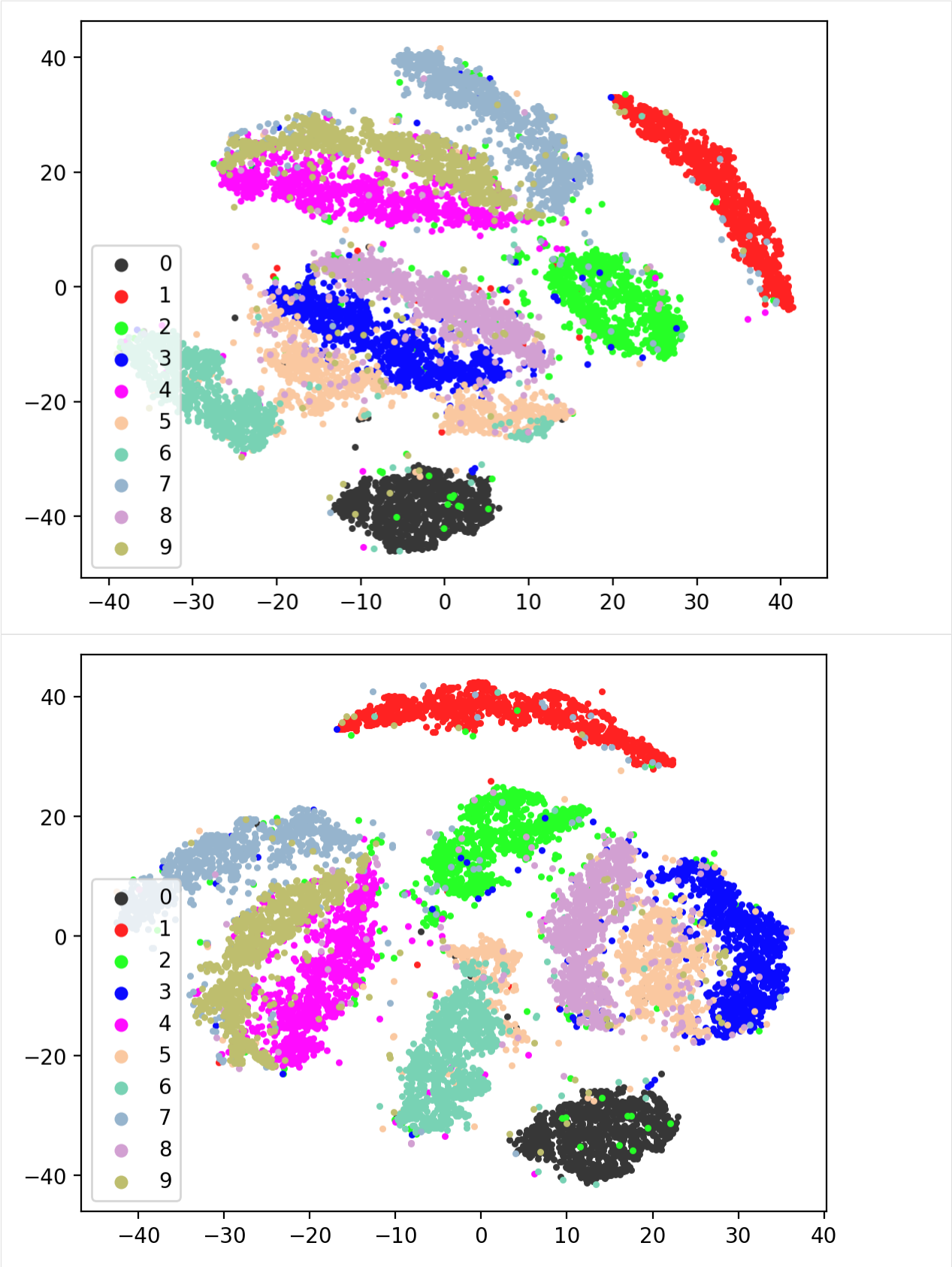
This is the image we're trying to reproduce:

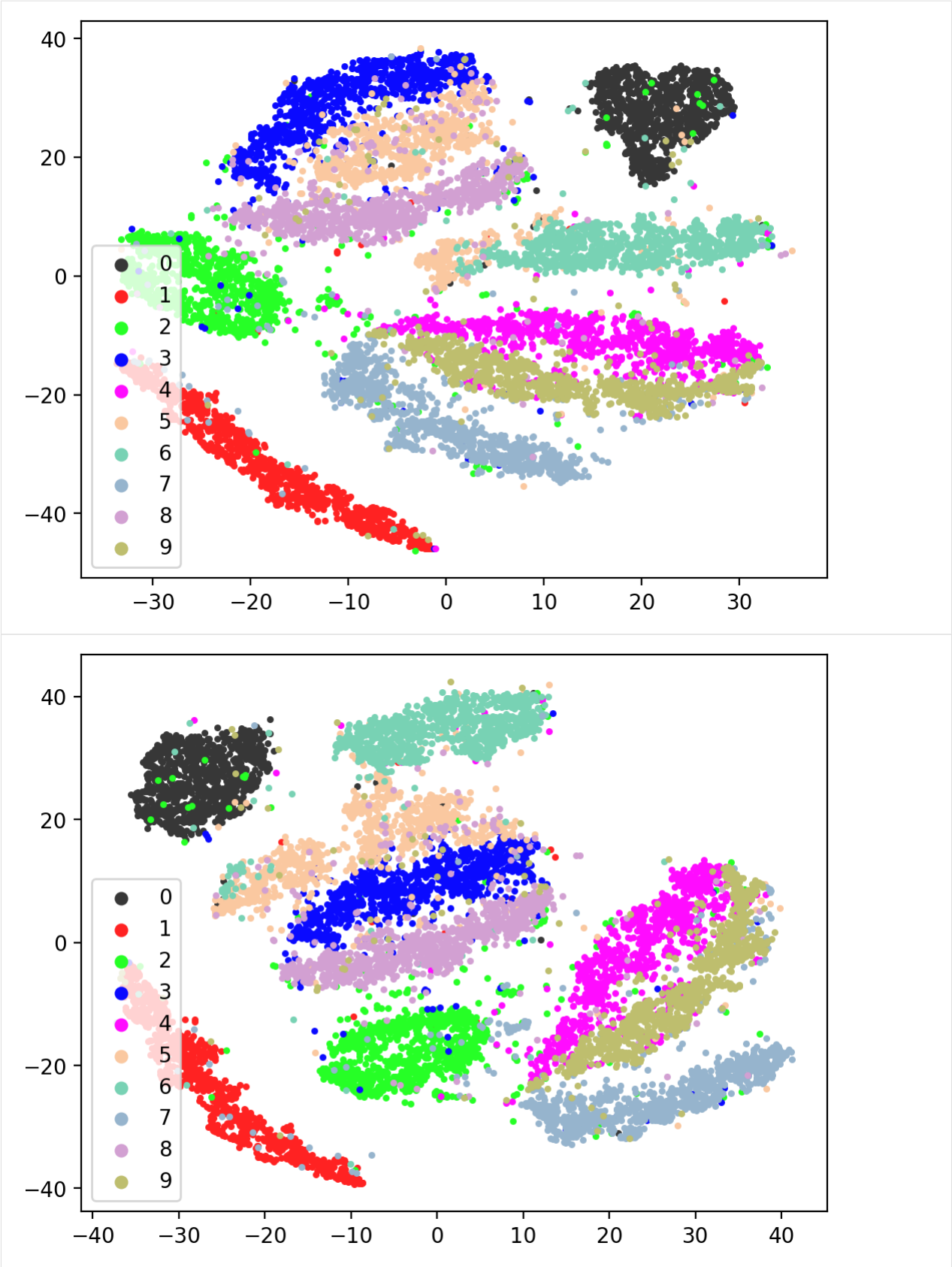


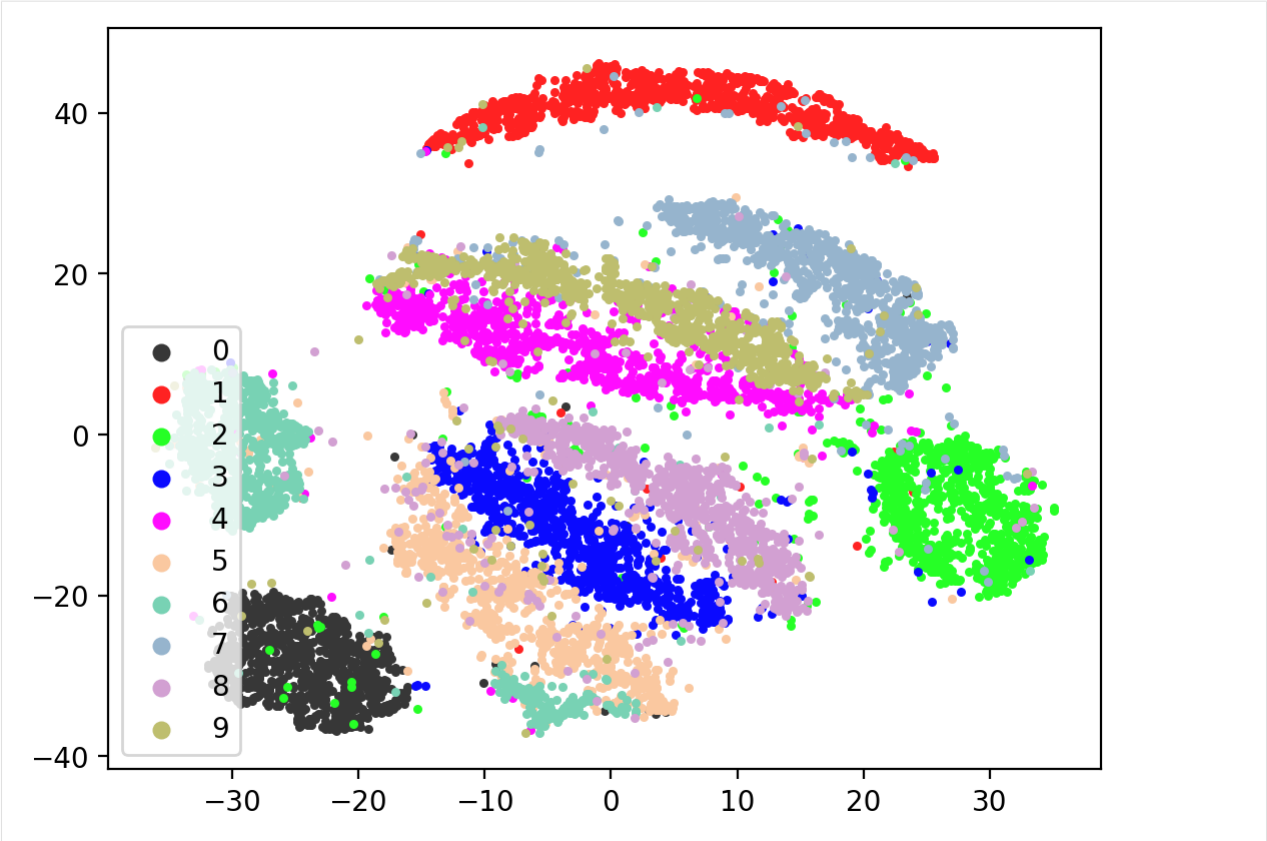
Lets check the variability of multiple TSNE runs:

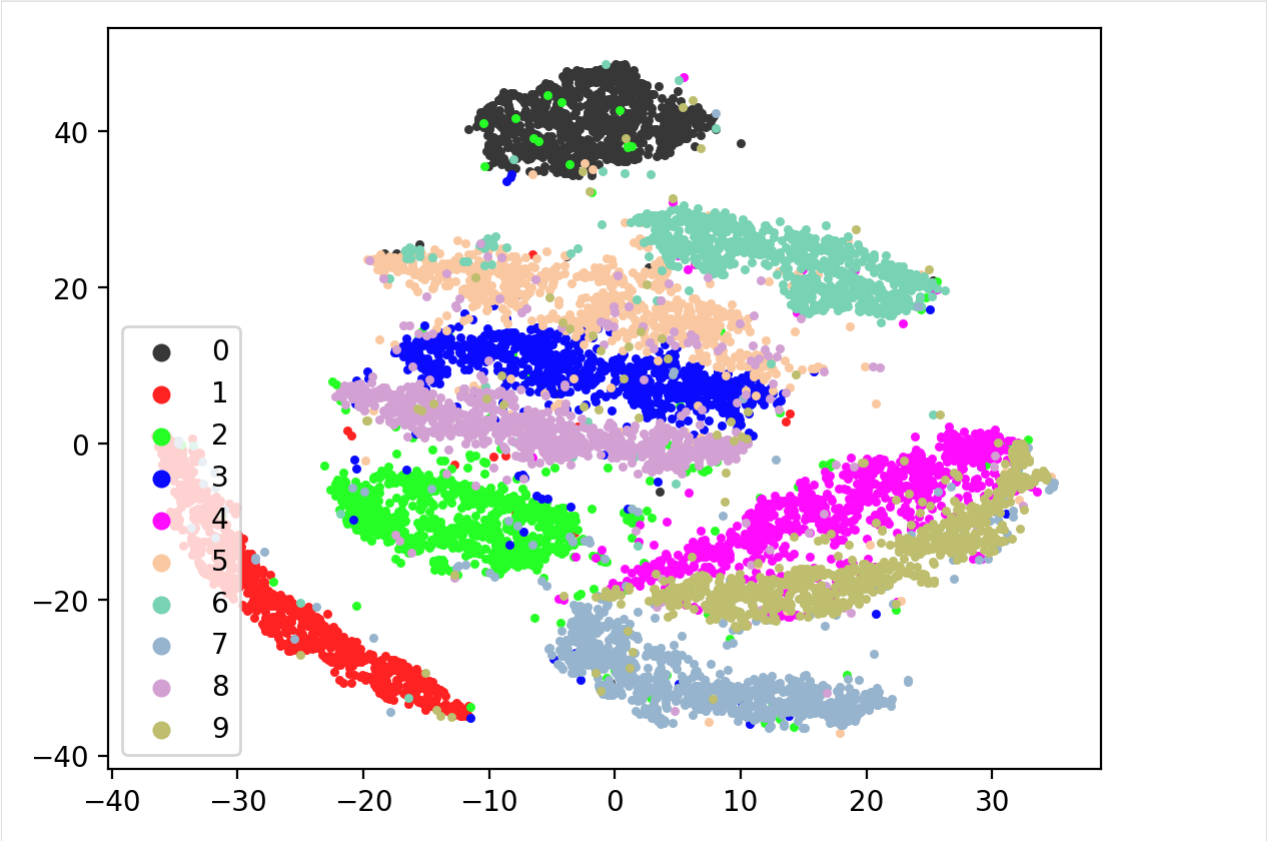
[22]:
```
for i in range(10):
    tsneEmbeddings = tsne.fit_transform(testEmbed)
    plot2DEmbeddings(tsneEmbeddings, testLabels)
```

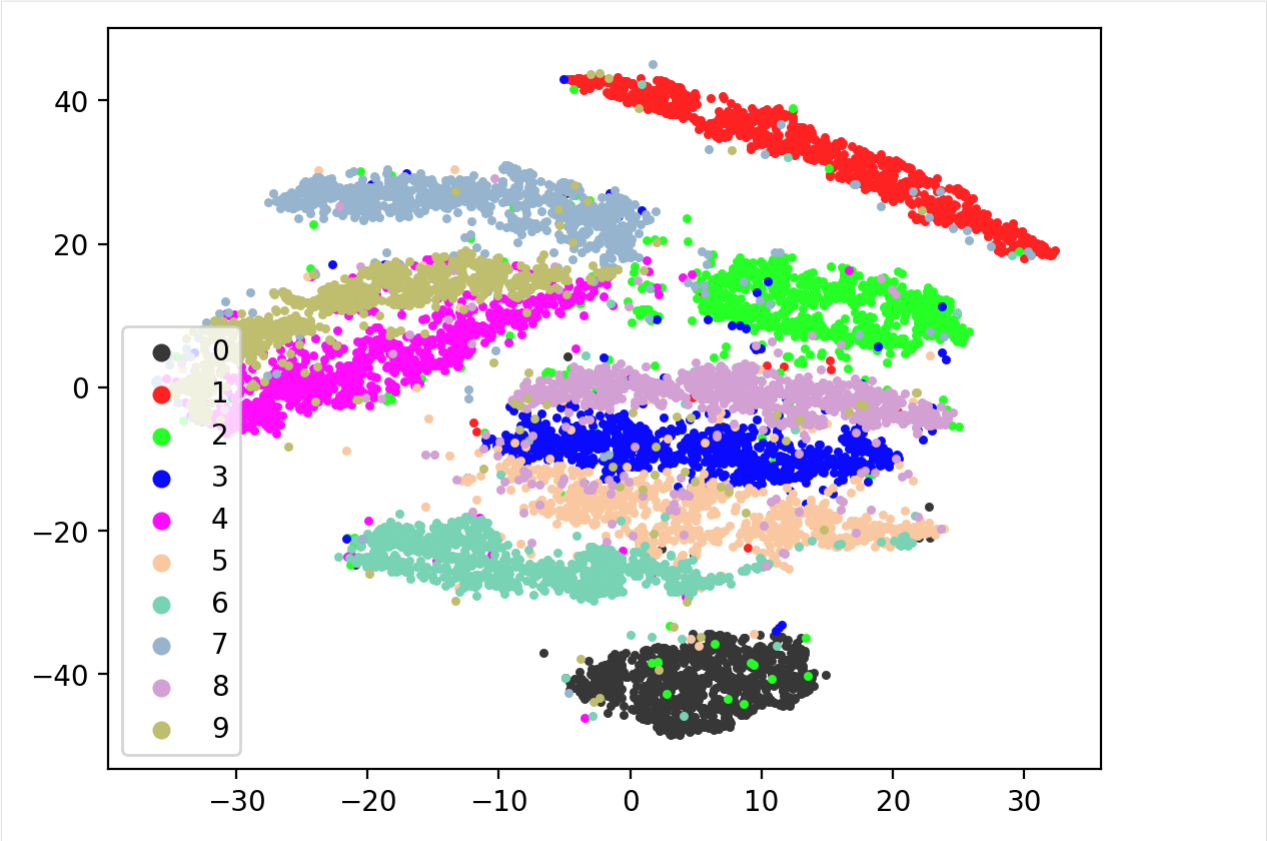

40
20
0
−20
−40
−40
−30
−20
−10
0
10
20
30
40
0
1
2
3
4
5
6
7
8
9
30
20
10
0
−10
−20
−30
−40
−40
−20
0
20
40
0
1
2
3
4
5
6
7
8
9

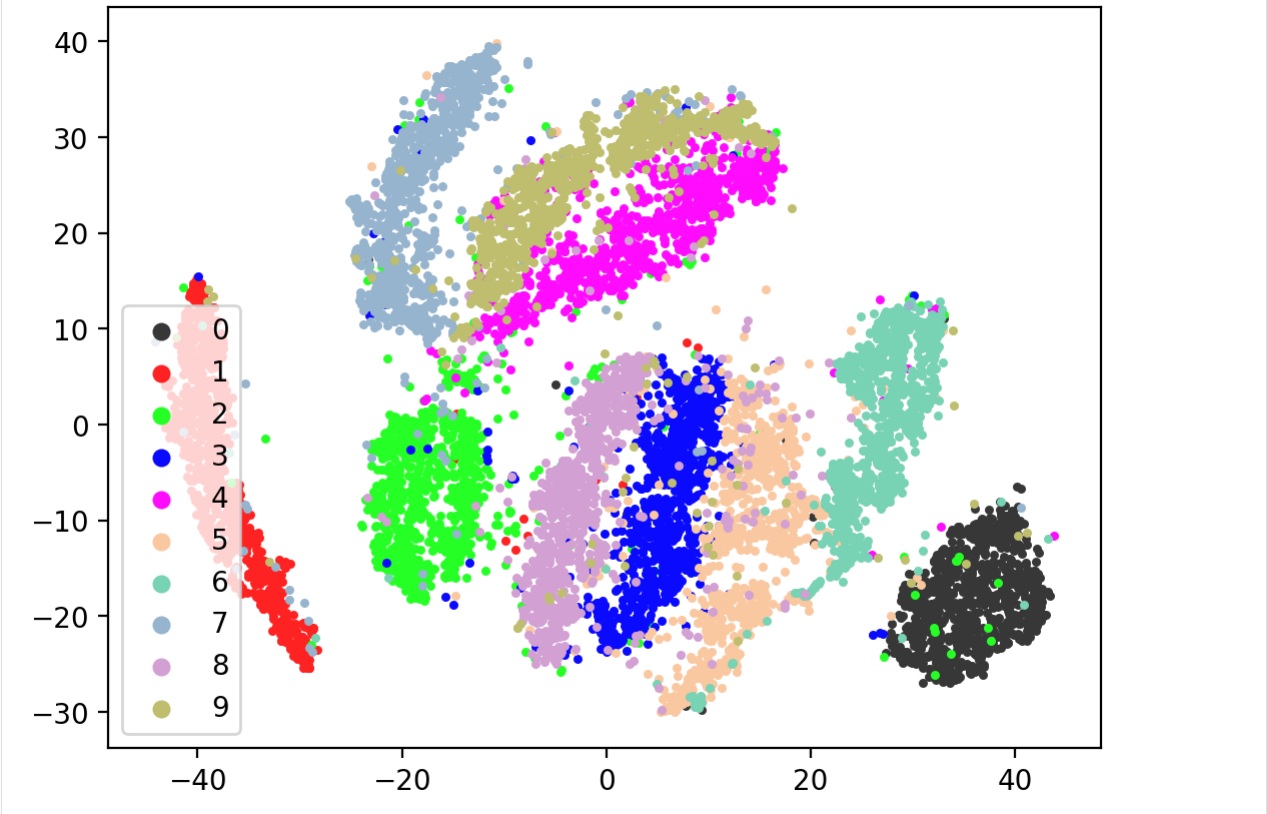

40
20
0
−20
−40
−40
−30
−20
−10
0
10
20
30
40
0
1
2
3
4
5
6
7
8
9


40
20
0
−20
−40
−30
−20
−10
0
10
20
30
0
1
2
3
4
5
6
7
8
9
40
20
0
−20
−40
−40
−30
−20
−10
0
10
20
30
40
0
1
2
3
4
5
6
7
8
9


40
20
0
−20
−40
−30
−20
−10
0
10
20
30
0
1
2
3
4
5
6
7
8
9


40
20
0
−20
−40
−40
−30
−20
−10
0
10
20
30
0
1
2
3
4
5
6
7
8
9

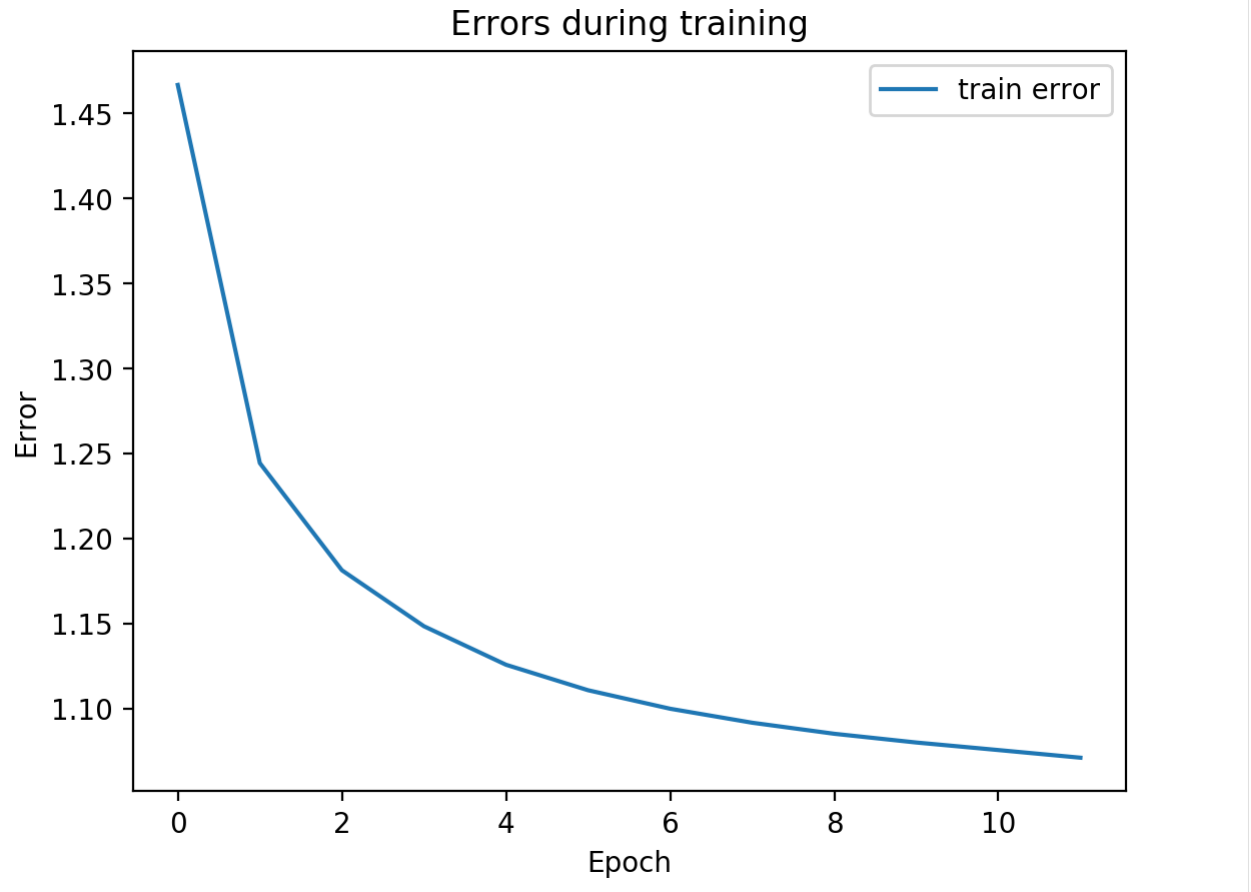

40
20
0
−20
−40
−30
−20
−10
0
10
20
30
0
1
2
3
4
5
6
7
8
9

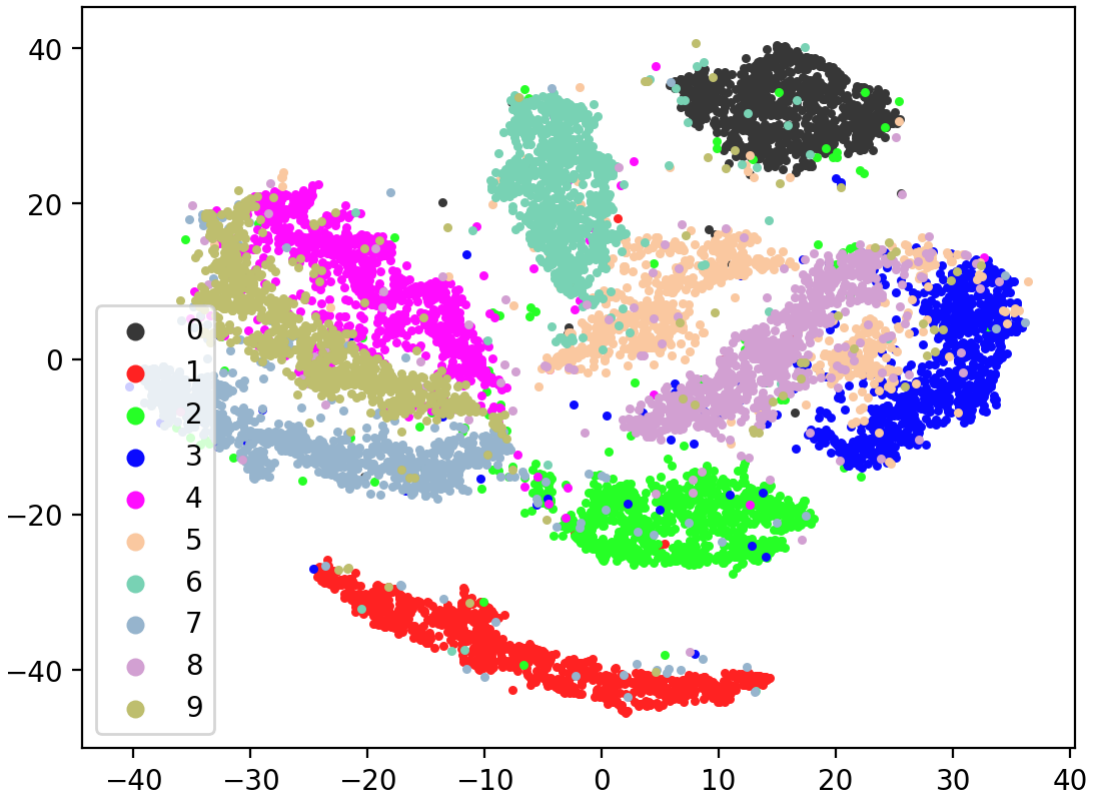

40
30
20
10
0
−10
−20
−30
−40
−20
0
20
40
0
1
2
3
4
5
6
7
8
9

Now let's check the variability of both training the model plus TSNE-ing the test embeddings.

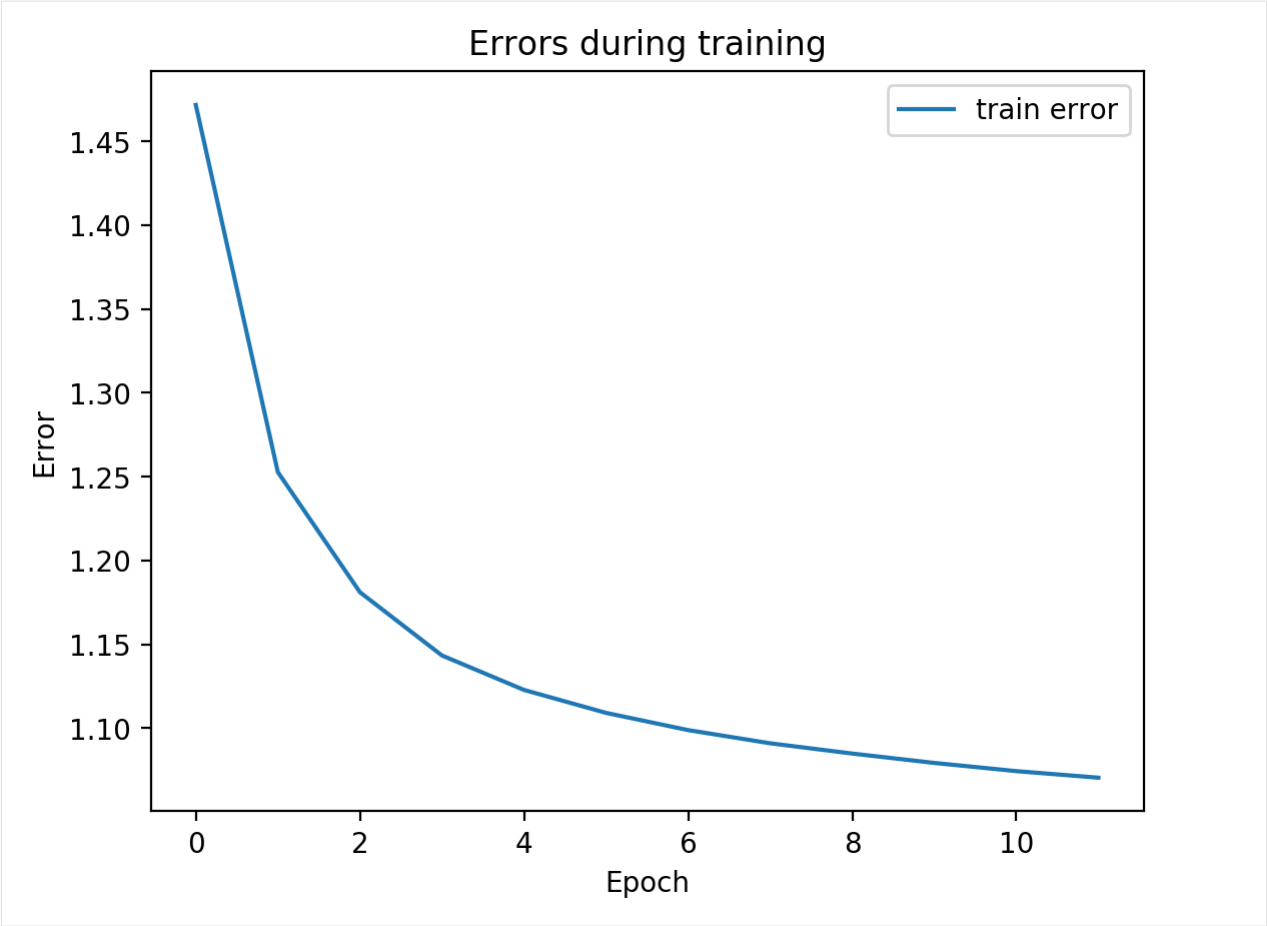
[23]:
```
for i in range(10):

    splitae = SplitAE(hidden_size=1024, num_hidden_layers=2, embed_size=10, training_
↪epochs=12, batch_size=128,
                      learning_rate=0.001, print_info=False, print_graph=True)
    splitae.fit([view1, view2])

    testEmbed, testView1Reconstruction, testView2Reconstruction = splitae.
↪transform([testView1, testView2])

    tsneEmbeddings = tsne.fit_transform(testEmbed)
    plot2DEmbeddings(tsneEmbeddings, testLabels)
```

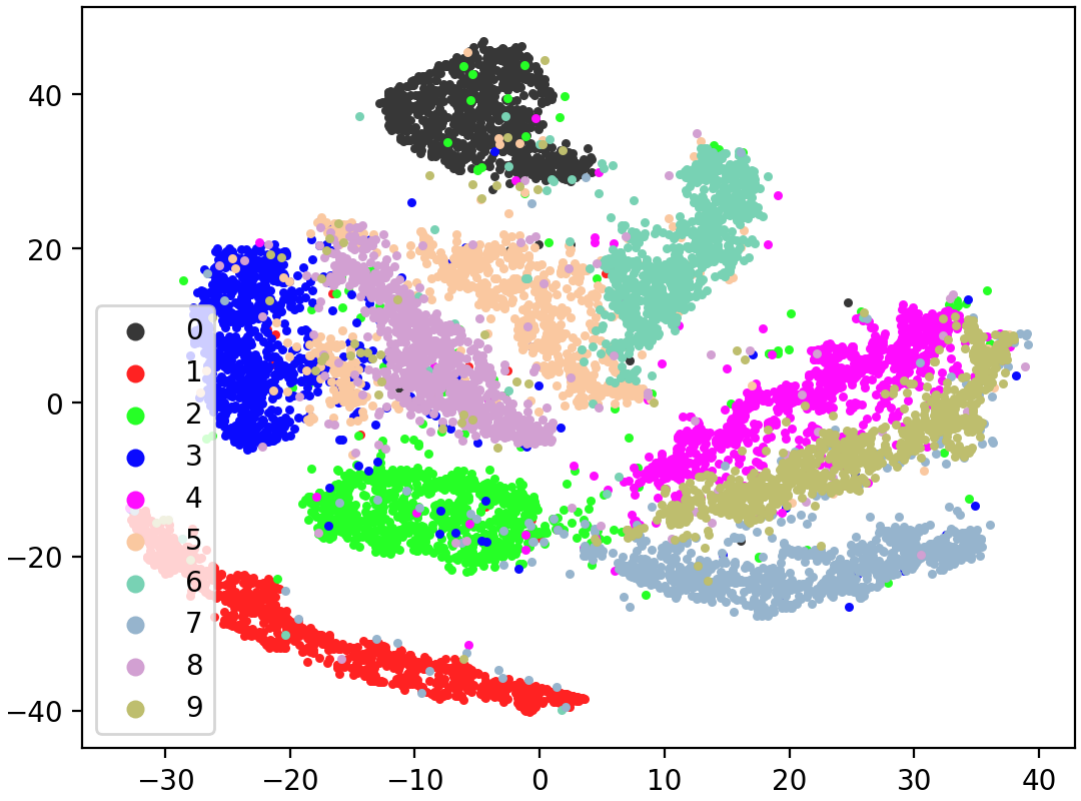
```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```



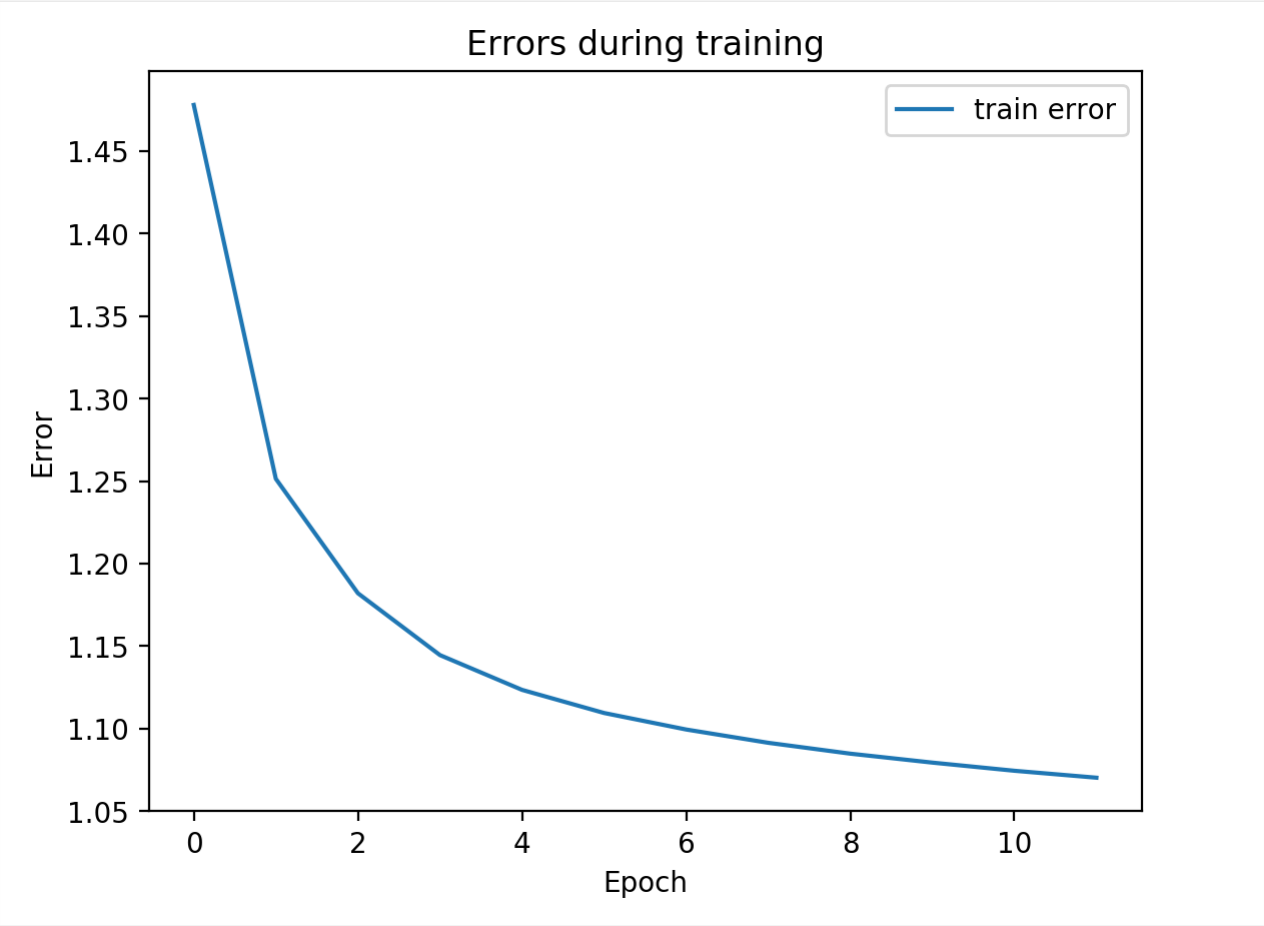


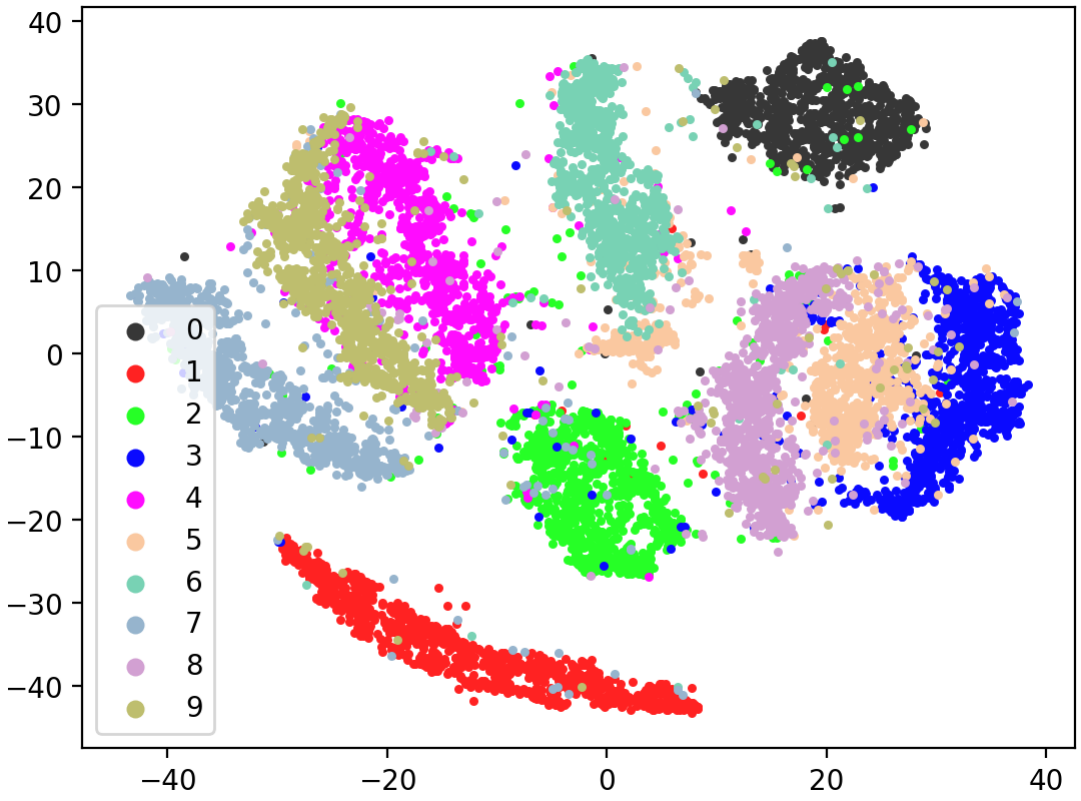

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
Error
0
2
4
6
8
10
Epoch

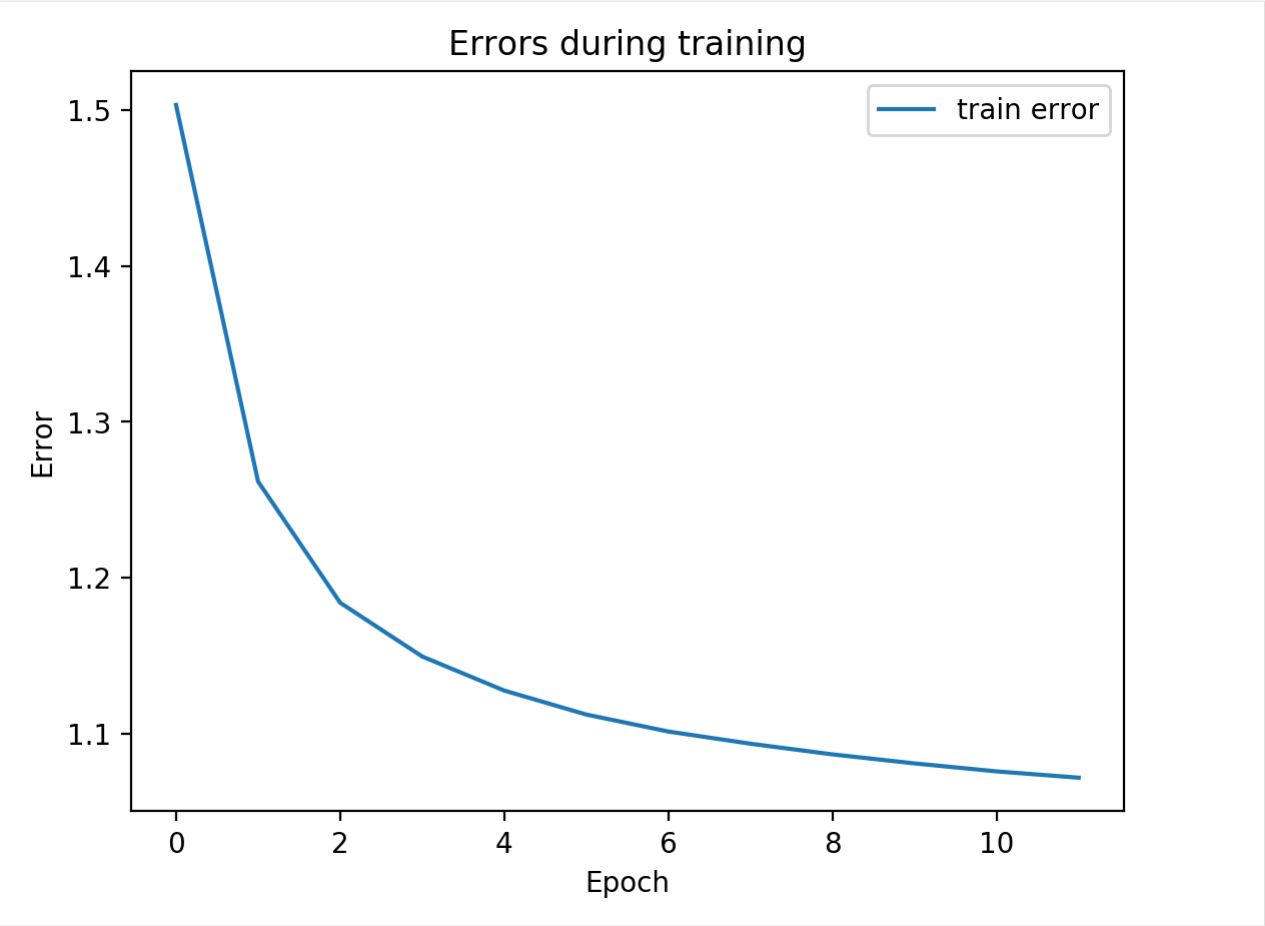


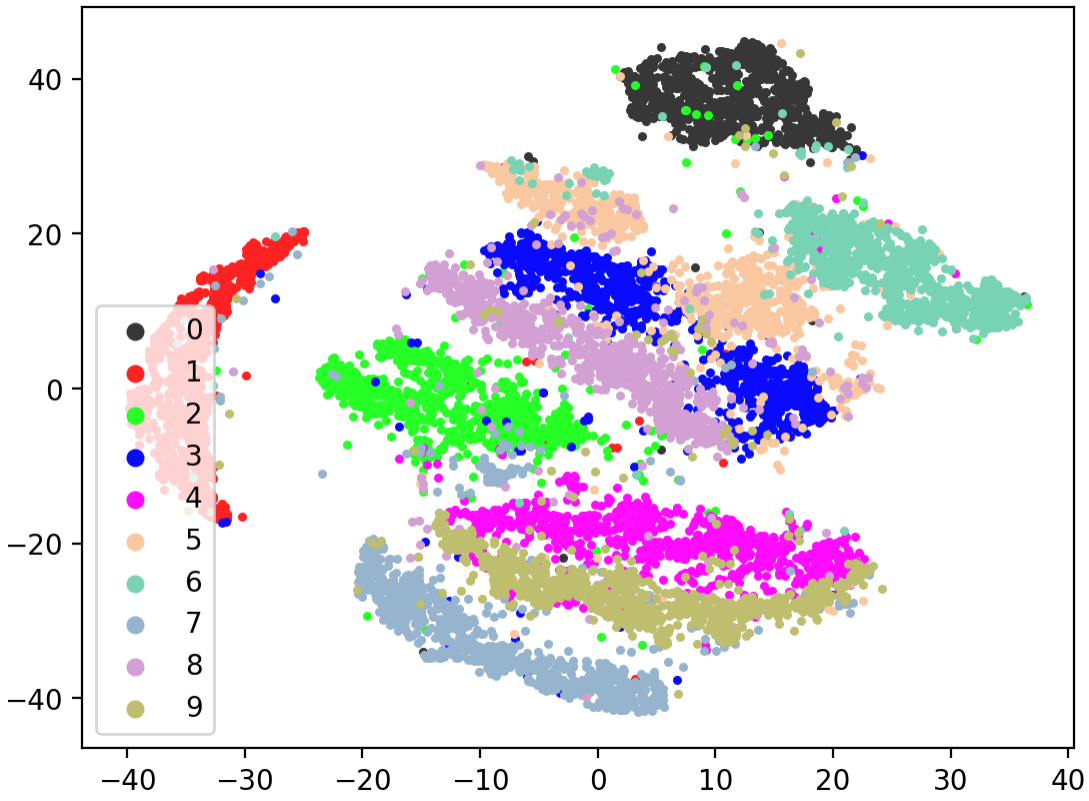

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
1.05
Error
0
2
4
6
8
10
Epoch

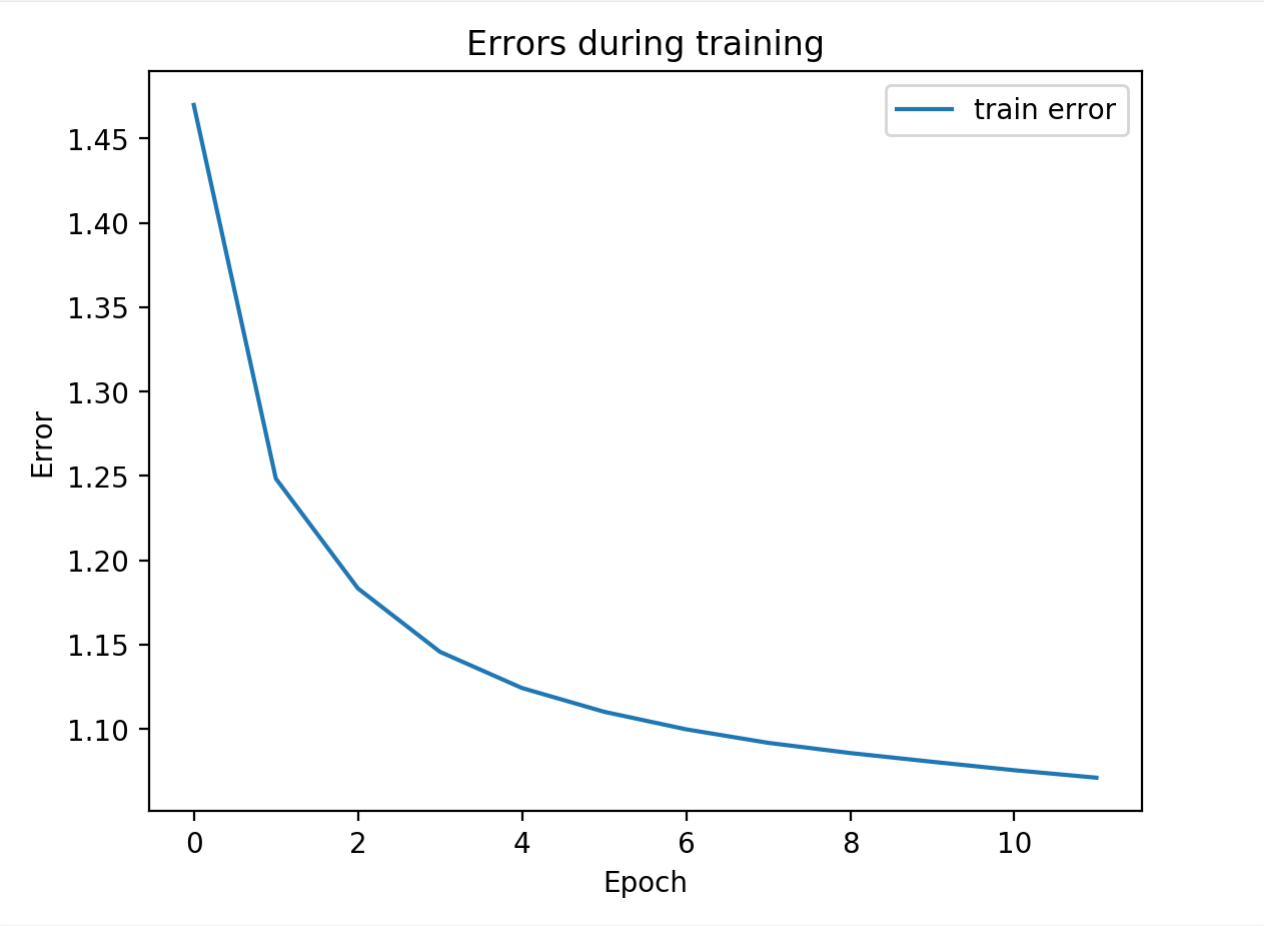


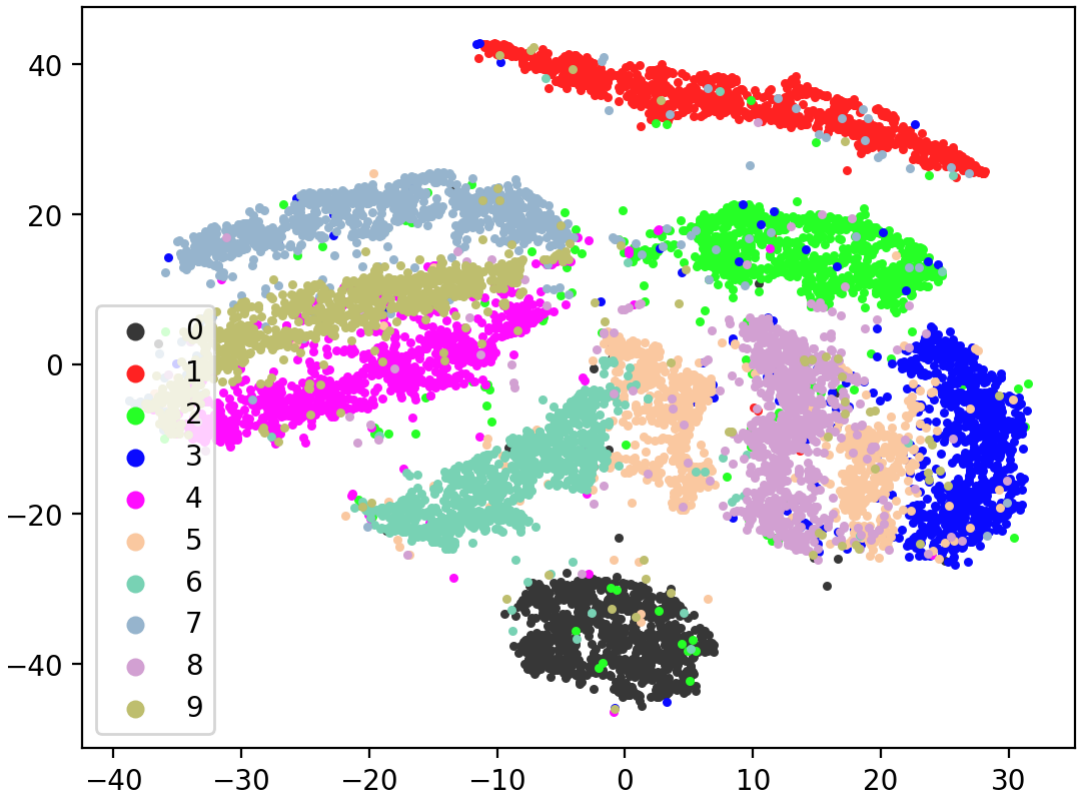

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.5
1.4
1.3
1.2
1.1
Error
0
2
4
6
8
10
Epoch

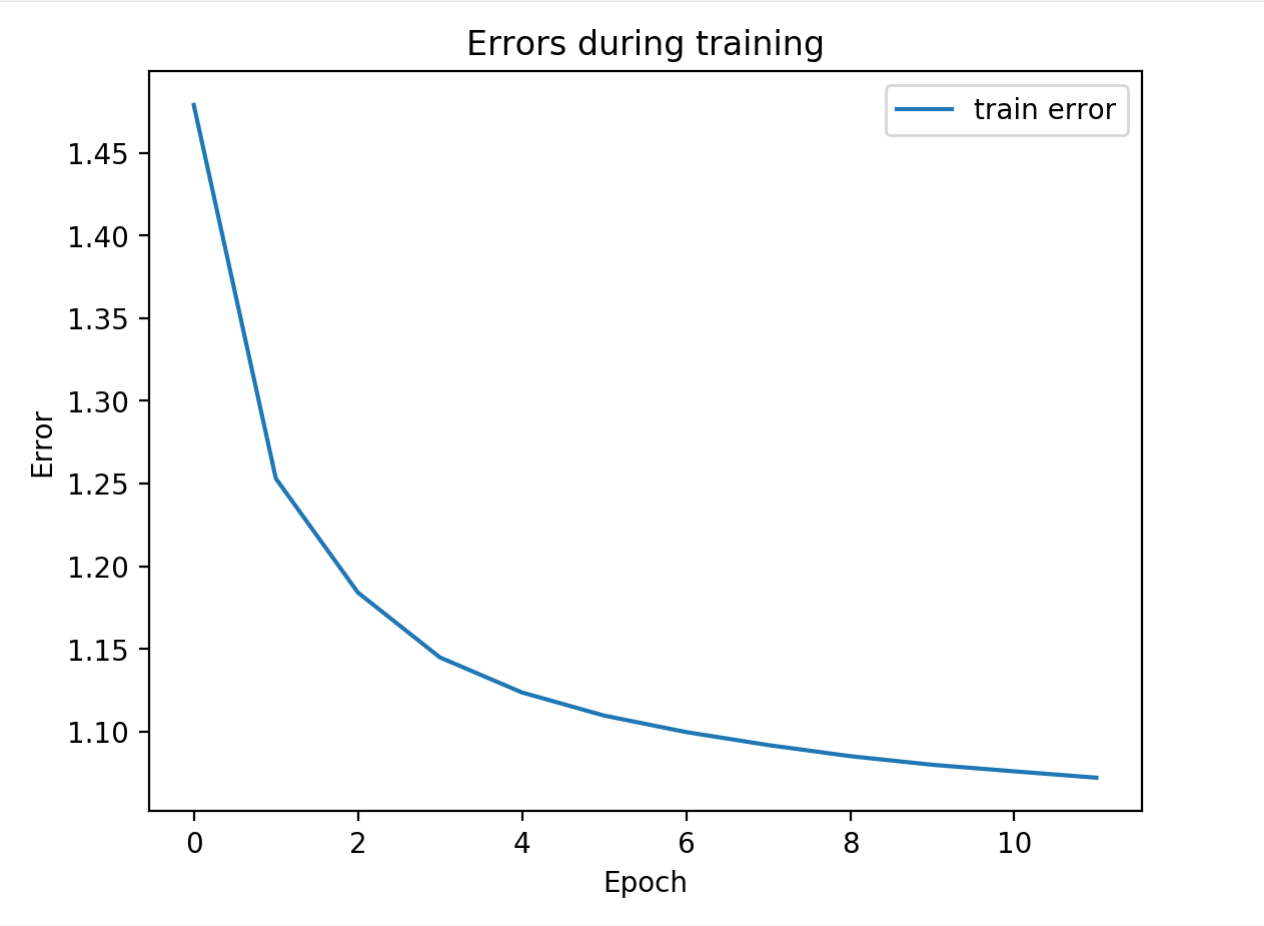


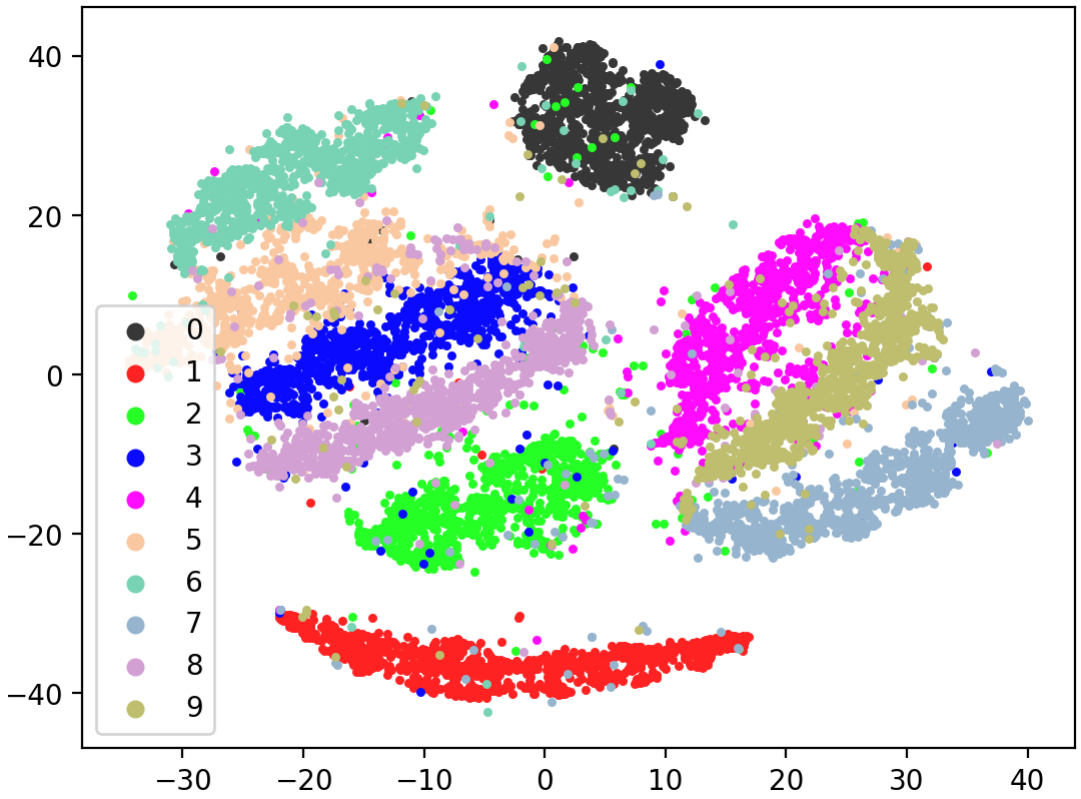

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
Error
0
2
4
6
8
10
Epoch

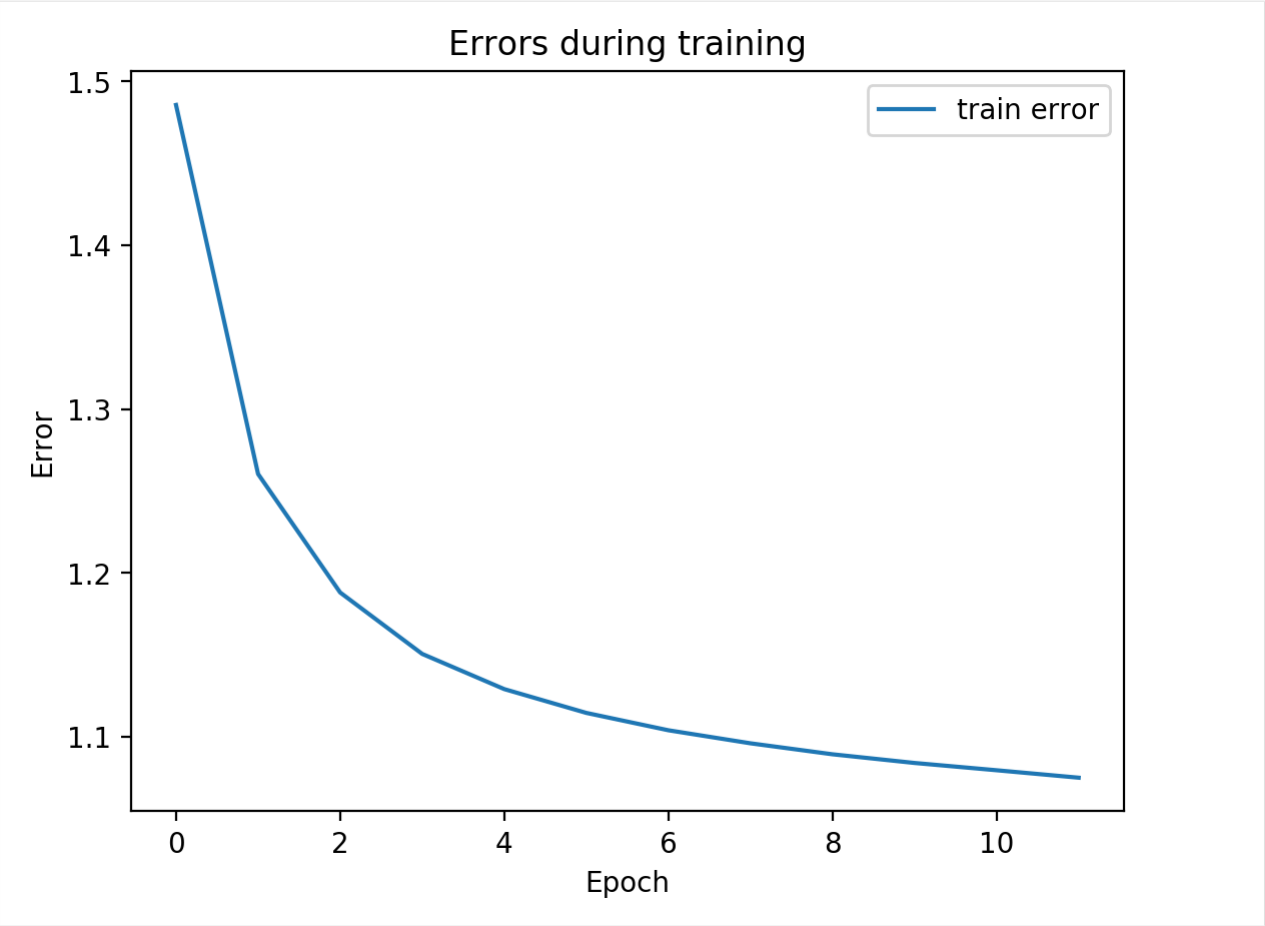


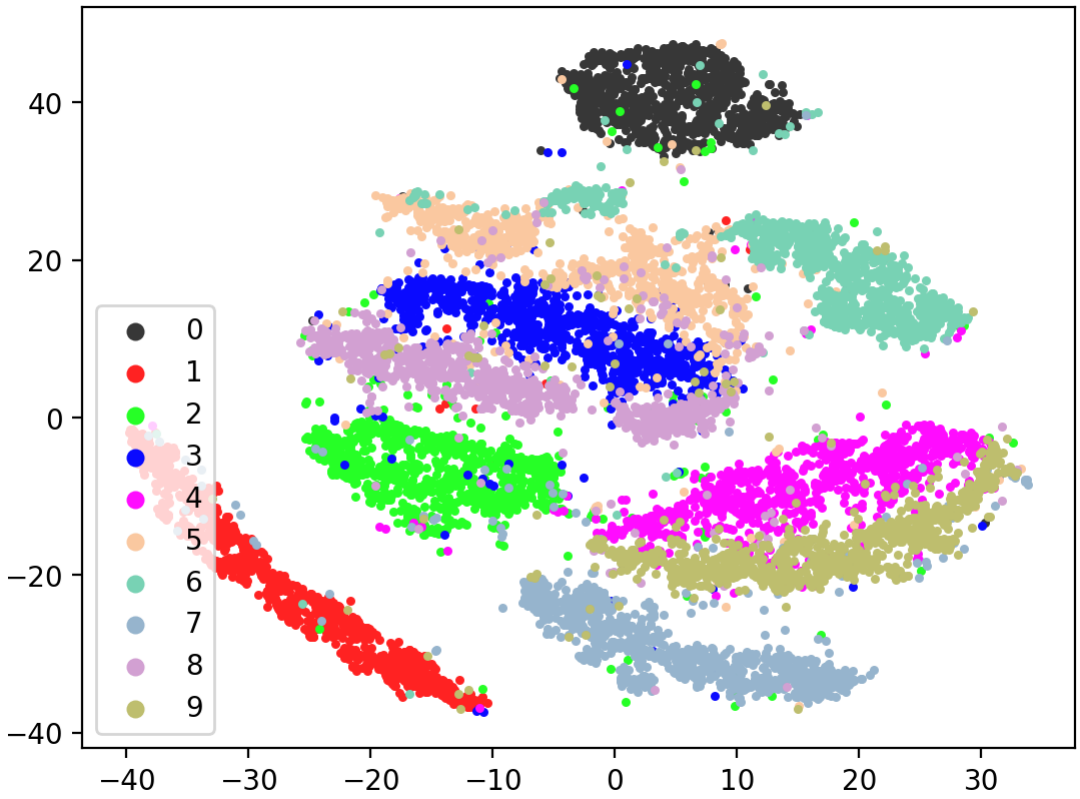

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
Error
0
2
4
6
8
10
Epoch

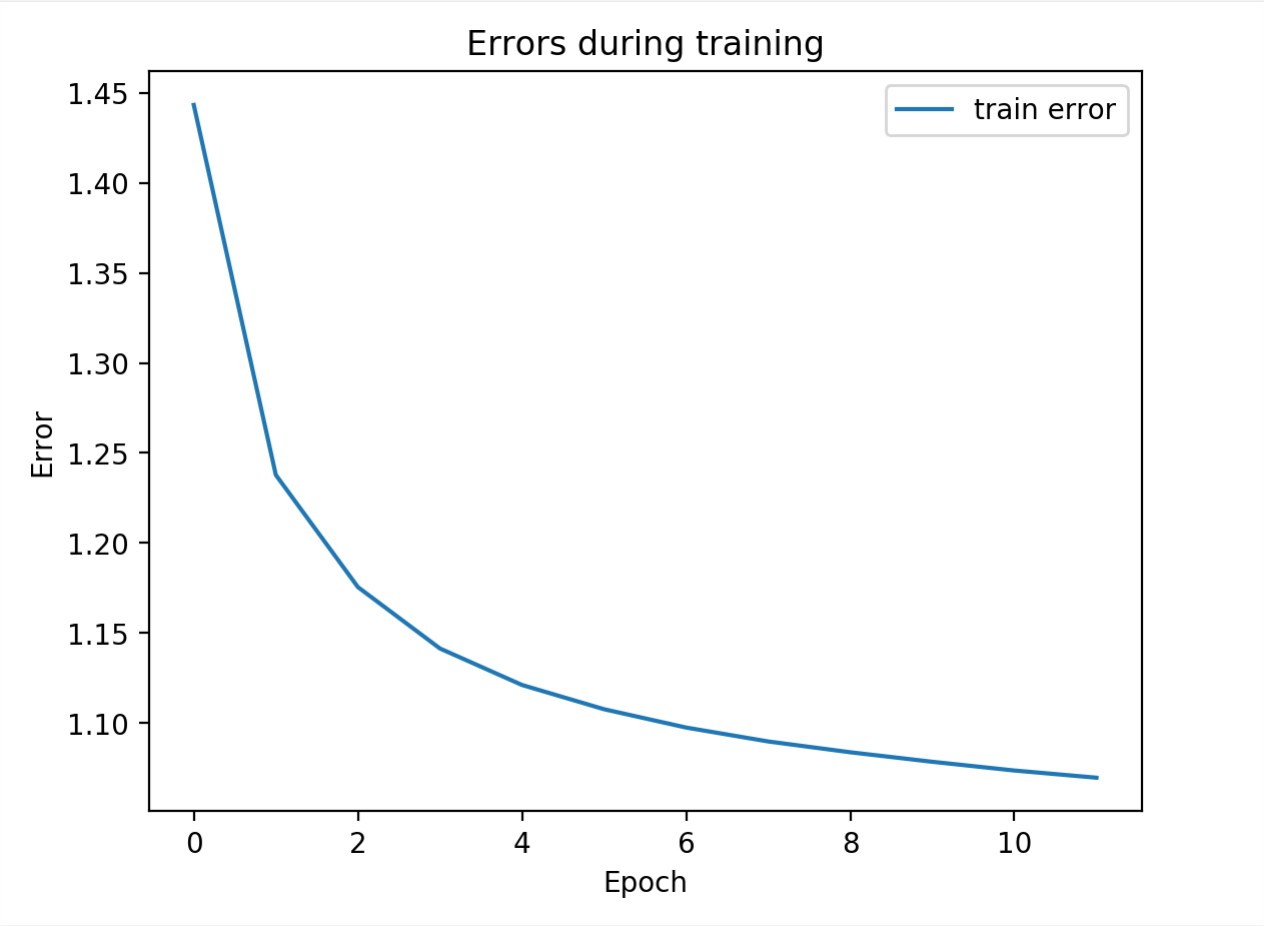


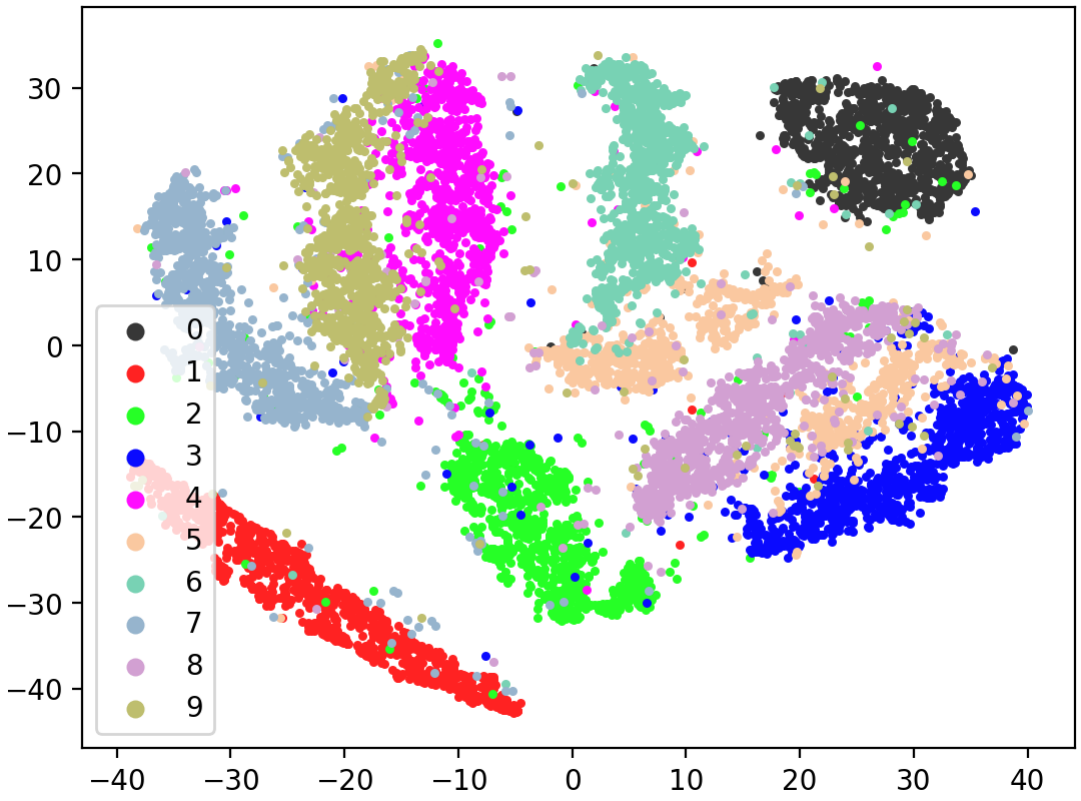

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.5
1.4
1.3
1.2
1.1
Error
0
2
4
6
8
10
Epoch

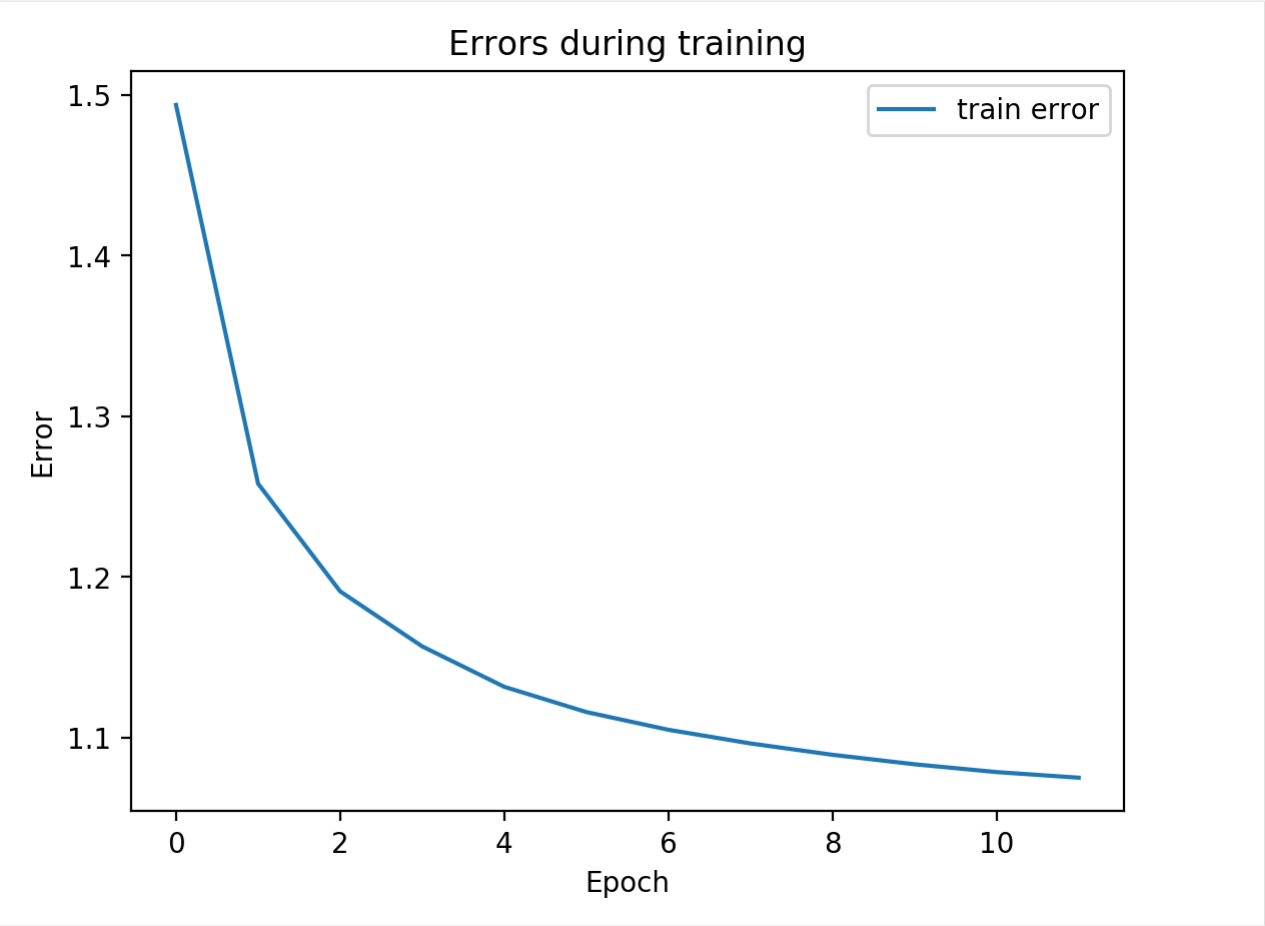


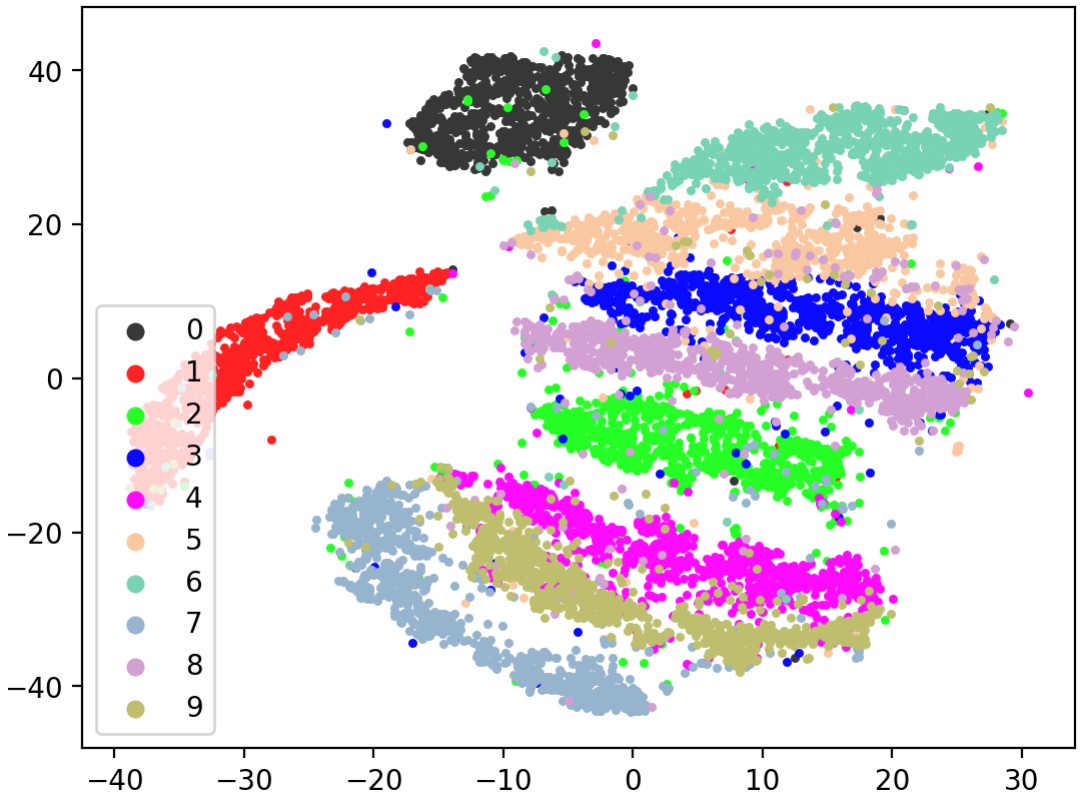

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
Error
0
2
4
6
8
10
Epoch

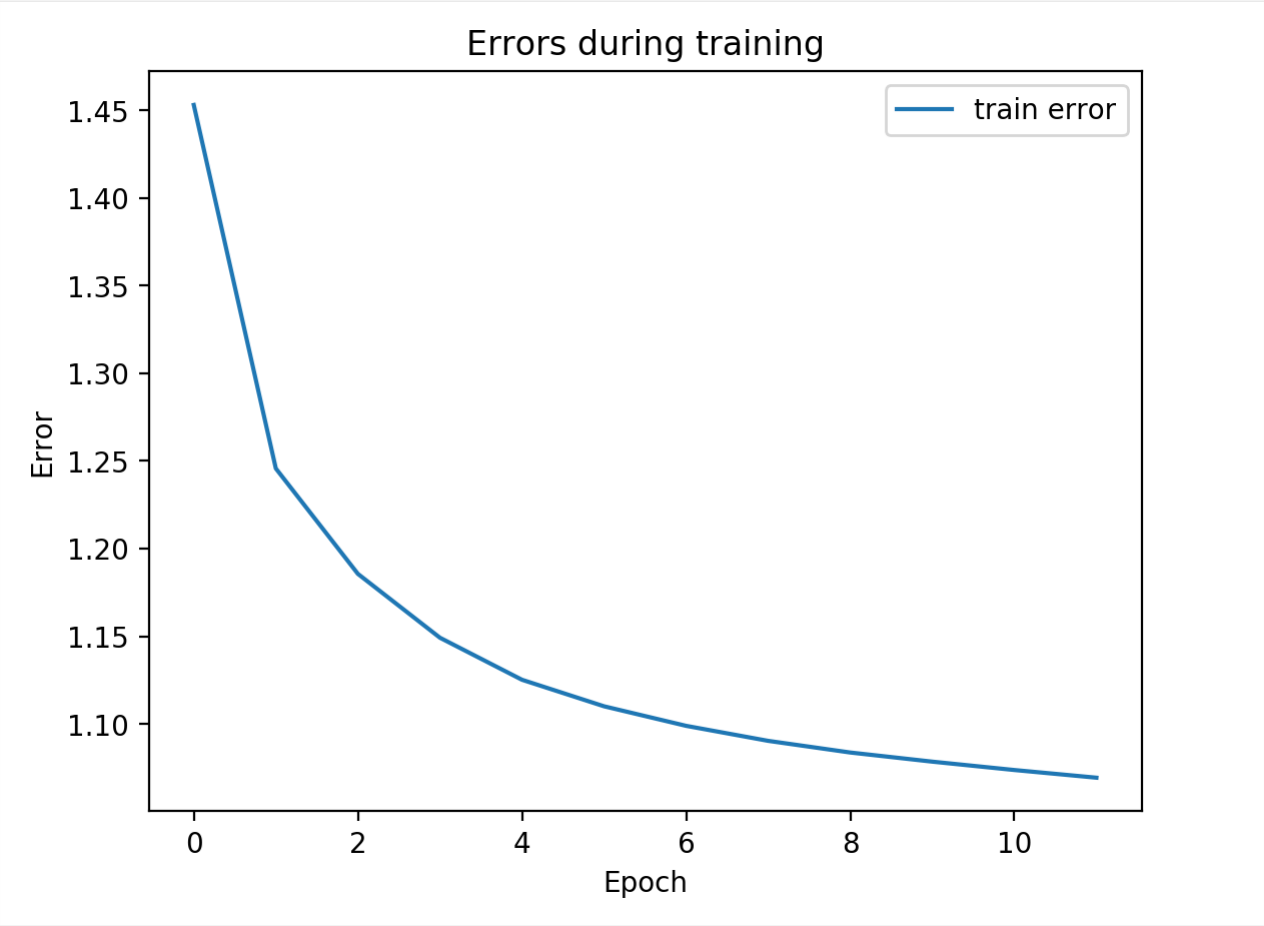


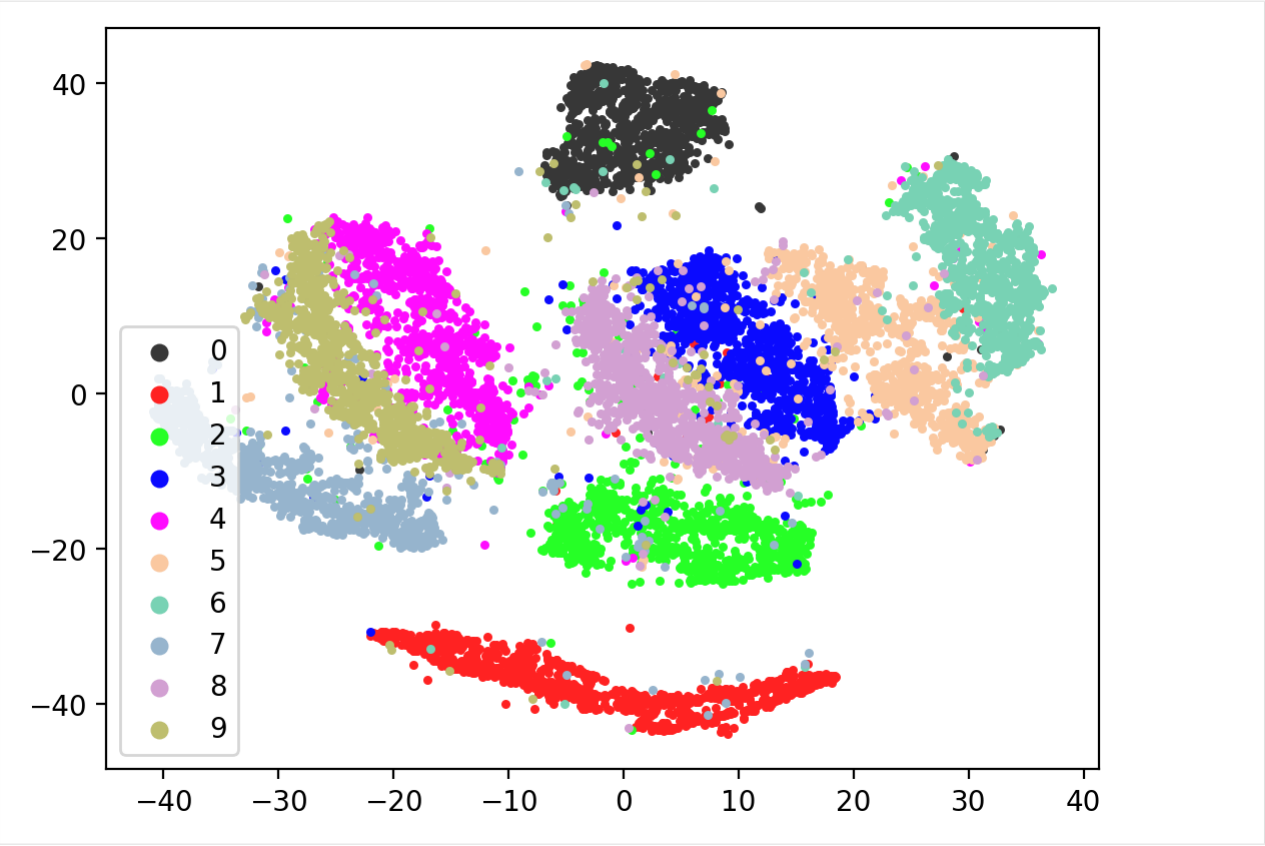

```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.5
1.4
1.3
1.2
1.1
Error
0
2
4
6
8
10
Epoch




```
Parameter counts:
view1Encoder: 1,863,690
view1Decoder: 1,864,464
view2Decoder: 1,864,464
```


Errors during training
train error
1.45
1.40
1.35
1.30
1.25
1.20
1.15
1.10
Error
0
2
4
6
8
10
Epoch



In most of the plots in the above cell we can see the distinct connected bands of the original figure, as well as the distinct black circular blob (corresponding to the digit 0, which appears easiest to learn). In some of the figures, a one or two bands are broken up. With more training of the network, though (stepping the learning rate), the bands converge less stretched (i.e. average distance between vectors of the same class is closer) blobs.

Predicting views using SplitAE

[15]:
```
import numpy as np
import torch
from mvlearn.embed import SplitAE
import matplotlib.pyplot as plt
import sklearn.cross_decomposition
plt.style.use("ggplot")
%config InlineBackend.figure_format = 'svg'
```

[16]:
```
# cca, previously validated against sklearn CCA
def cca(X, Y, regularizationλ=0):

    X = X - X.mean(axis=0)
    Y = Y - Y.mean(axis=0)
    k = min(X.shape[1], Y.shape[1])
    covXX = (X.t() @ X) / X.shape[0] + regularizationλ*torch.eye(X.shape[1], device=X.
↪device)
    covYY = (Y.t() @ Y) / X.shape[0] + regularizationλ*torch.eye(Y.shape[1], device=X.
↪device)
    covXY = (X.t() @ Y) / X.shape[0]
```

(continues on next page)

(continued from previous page)

```
    U_x, S_x, V_x = covXX.svd()
    U_y, S_y, V_y = covYY.svd()
    covXXinvHalf = V_x @ (S_x.sqrt().reciprocal().diag()) @ U_x.t()
    covYYinvHalf = V_y @ (S_y.sqrt().reciprocal().diag()) @ U_y.t()
    T = covXXinvHalf @ covXY @ covYYinvHalf
    U, S, V = T.svd()
    A = covXXinvHalf @ U[:, :k]
    B = covYYinvHalf @ V[:, :k]
    return A.t(), B.t(), S
```

Predicting a held out view with CCA, nonlinear relationship between views

[17]:
```
# The relationship between view1 and view2 is that view2(t) = view1(t) ** 2.
# In words, View1(t) is a nonlinear function of View2(t)
view1 = np.random.randn(10000, 10)
view2 = view1 ** 2
# view2 = view1 @ np.random.randn(10, 10)

# Let's say now say we have 10,000 points with a view1 but only 5000 of those points␣
↪have a view 2. So
# one obvious goal is to somehow reconstruct the missing view2 data for those points.
view1Train = view1[:5000]
view2Train = view2[:5000]
view1Test = view1[5000:]
view2Test = view2[5000:] # these are what we're trying to predict

# Let's try and predict view2Test with CCA
U, V, S = cca(torch.FloatTensor(view1Train), torch.FloatTensor(view2Train))
view1CCs = view1Train @ U.t().numpy()
view2CCs = view2Train @ V.t().numpy()
covariance = np.mean((view1CCs - view1CCs.mean(axis=0)) * (view2CCs - view2CCs.
↪mean(axis=0)), axis=0)
stdprod = np.std(view1CCs, axis=0) * np.std(view2CCs, axis=0)
correlations = covariance / stdprod

# we can see that the canonical correlations are very low. This means that for any␣
↪given sample, the
# vector of view1 canonical variables will not be close to the vector of view2␣
↪canonical variables.
# Ideally the canonical correlations would be 1, so that the for each point, each view
↪'s canonical variable
# has the same vlaue.
plt.plot(correlations)
plt.title("Canonical Correlations")
plt.show()
```

[18]:
```
# This is how we predict our training data given the canonical variables
view1TrainPred = view1CCs @ np.linalg.inv(U.t().numpy())
view2TrainPred = view2CCs @ np.linalg.inv(V.t().numpy())
assert np.all(view1TrainPred - view1Train < 1e-2)
assert np.all(view2TrainPred - view2Train < 1e-2)
```

(continues on next page)

(continued from previous page)

```
# This is how we predict View2 from View1 values. Notice the V.t() matrix being used␣
↪for view1 values.
view1TestCCs = view1Test @ U.t().numpy()
view2TestPred = view1TestCCs @ np.linalg.inv(V.t().numpy())

# Notice that the magnitude of the errors are close to the magnitude of the view2␣
↪elements themselves!
# these are bad predictions.
predictionErrors = np.abs(view2TestPred - view2Test).ravel()
plt.hist(predictionErrors)
plt.title("Prediction Errors")
plt.show()
plt.hist(view2.ravel())
plt.title("View 2 Magnitudes")
plt.show()

print("MSE Loss is ", np.mean((view2TestPred - view2Test)**2))

# If you repeat this experiment with view2 = (some linear combination of the features␣
↪of view1),
# for example view2 = view1 @ np.random.randn(10, 10)
# the prediction errors will be zero. This is where CCA exceeds, when the above is␣
↪true. We will see this
# next time we run CCA.
```

```
MSE Loss is  5.096770717383144
```

Predicting a held out view with SplitAE, nonlinear relationship between views

[19]:
```
# Now lets try the same thing with SplitAE!
splitae = SplitAE(hidden_size=32, num_hidden_layers=1, embed_size=20, training_
↪epochs=50, batch_size=32, learning_rate=0.01, print_info=False, print_graph=True)
splitae.fit([view1Train, view2Train], validationXs=[view1Test, view2Test])

# (I'm using the test data to see validation loss, in a real case the validation set␣
↪is held out data and the test set is unknown / not used until the end)
embeddings, reconstructedView1, predictedView2 = splitae.transform([view1Test])
predictionErrors = np.abs(predictedView2 - view2Test).ravel()
plt.hist(predictionErrors)
plt.title("Prediction Errors")
plt.show()
plt.hist(view2.ravel())
plt.title("View 2 Magnitudes")
plt.show()

print("MSE Loss is ", np.mean((predictedView2 - view2Test)**2))

# The bins near 0 are a bit deceiving on the histograms, but the loss shows it all --␣
↪with splitAE we can
# predict our view2 from view1 with much higher accuracy than CCA.
# The tradeoff here was hyperparameter tuning -- I had to get the embed size right,␣
↪the number of hidden layers right
# (too big, and the loss will converge to something higher), and train for the right␣
↪amount of time.
```

```
Parameter counts:
view1Encoder: 1,012
view1Decoder: 1,002
view2Decoder: 1,002
```

```
MSE Loss is  0.052350855326646545
```

Predicting a held out view with CCA, linear relationship between views, few data points

[20]:
```
# Lets say instead of 5000 input points we only have 50 train points and 50 test points. And that this time,
# we have a generally linear relationship.
view1 = np.random.randn(100, 10)
view2 = view1 @ np.random.randn(10, 10)

view1Train = view1[:50]
view2Train = view2[:50]
view1Test = view1[50:]
view2Test = view2[50:] # these are what we're trying to predict

U, V, S = cca(torch.FloatTensor(view1Train), torch.FloatTensor(view2Train))
view1TestCCs = view1Test @ U.t().numpy()
view2TestPred = view1TestCCs @ np.linalg.inv(V.t().numpy())
print("MSE Loss is ", np.mean((view2TestPred - view2Test)**2))

# CCA achieves a loss of ~0. Can splitAE achieve the same?
```

```
MSE Loss is  2.517854473585315e-11
```

Predicting a held out view with SplitAE, linear relationship between views, few data points

[21]:
```
splitae = SplitAE(hidden_size=32, num_hidden_layers=2, embed_size=20, training_epochs=500, batch_size=10, learning_rate=0.01, print_info=False, print_graph=True)
splitae.fit([view1Train, view2Train], validationXs=[view1Test, view2Test])
embeddings, reconstructedView1, predictedView2 = splitae.transform([view1Test]) # using test data

print("MSE Loss for test data ", np.mean((predictedView2 - view2Test)**2))
embeddings, reconstructedView1, predictedView2 = splitae.transform([view1Train]) # using training data
print("MSE Loss for train data ", np.mean((predictedView2 - view2Train)**2))
print("MSE Loss when predicting mean", np.mean((0 - view2Train)**2))

# Clearly we have overfit, and from the graph we can see that we have done so within the first dozen epochs.
# Our test error is almost as bad a just predicting the mean. Can further tuning the parameters s.t.
# we don't overfit allow us to match CCA performance?
```

```
Parameter counts:
view1Encoder: 2,068
view1Decoder: 2,058
view2Decoder: 2,058
```

```
MSE Loss for test data  5.59877941169036
MSE Loss for train data  0.06715857622620428
MSE Loss when predicting mean 8.489032078377859
```

[22]:
```
splitae = SplitAE(hidden_size=32, num_hidden_layers=0, embed_size=20, training_
↪epochs=500, batch_size=10, learning_rate=0.01, print_info=False, print_graph=True)
splitae.fit([view1Train, view2Train], validationXs=[view1Test, view2Test])
embeddings, reconstructedView1, predictedView2 = splitae.transform([view1Test]) #␣
↪using test data
print("MSE Loss for test data ", np.mean((predictedView2 - view2Test)**2))

# Luckily, by converting our model to a linear one (i.e. numHiddenLayers=0, so no␣
↪activations are performed)
# we have once again predicted the test data correctly.
# But the trade-off here is clear. CCA has performed maybe 10 matrix operations.␣
↪SplitAE has performed at least
# 500*2 = 1000 equivalent matrix operations.
# Using %%timeit,
# - CCA takes ~600us to predict view2Test.
# - SplitAE takes ~4.5s (7,000x slower) to predict view2Test
```

```
Parameter counts:
view1Encoder: 220
view1Decoder: 210
view2Decoder: 210
```

```
MSE Loss for test data  0.00038991122173028984
```

5.2.4 Decomposition

The following tutorials show how to use multi-view decomposition algorithms.

Angle-based Joint and Individual Variation (AJIVE) Explained

AJIVE is a useful algorithm that decomposes multiple views of data into three main categories: - Joint Variation - Individual Variation - Noise

This notebook will prove out the implementation of AJIVE and show some examples of the algorithm's usefulness

[2]:
```
import numpy as np
from mvlearn.decomposition import AJIVE, data_block_heatmaps, ajive_full_estimate_
↪heatmaps
import matplotlib.pyplot as plt
%matplotlib inline
```

Data Creation

Here we create data in the same way detailed in the initial JIVE paper:

```
[1] Lock, Eric F., et al. "Joint and Individual Variation Explained (JIVE) for␣
↪Integrated Analysis of Multiple Data Types."        The Annals of Applied␣
↪Statistics, vol. 7, no. 1, 2013, pp. 523-542., doi:10.1214/12-aoas597.
```

The two views are created with shared joint variation, unique individual variation, and independent noise. A representation of what the implementation of this algorithm does can be seen in the cell below.

[2]:
```
np.random.seed(12)

# First View
V1_joint = np.bmat([[-1 * np.ones((50, 2000))],
                    [np.ones((50, 2000))]])

V1_joint = np.bmat([np.zeros((100, 8000)), V1_joint])

V1_indiv_t = np.bmat([[np.ones((20, 5000))],
                      [-1 * np.ones((20, 5000))],
                      [np.zeros((20, 5000))],
                      [np.ones((20, 5000))],
                      [-1 * np.ones((20, 5000))]])

V1_indiv_b = np.bmat([[np.ones((25, 5000))],
                      [-1 * np.ones((50, 5000))],
                      [np.ones((25, 5000))]])

V1_indiv_tot = np.bmat([V1_indiv_t, V1_indiv_t])

V1_noise = np.random.normal(loc=0, scale=1, size=(100, 10000))


# Second View
V2_joint = np.bmat([[np.ones((50, 50))],
                    [-1*np.ones((50, 50))]])

V2_joint = 5000 * np.bmat([V2_joint, np.zeros((100, 50))])

V2_indiv = 5000 * np.bmat([[-1 * np.ones((25, 100))],
                           [np.ones((25, 100))],
                           [-1 * np.ones((25, 100))],
                           [np.ones((25, 100))]])

V2_noise = 5000 * np.random.normal(loc=0, scale=1, size=(100, 100))

# View Construction

V1 = V1_indiv_tot + V1_joint + V1_noise

V2 = V2_indiv + V2_joint + V2_noise

Views_1 = [V1, V1]
Views_2 = [V1, V2]
```

Scree Plots

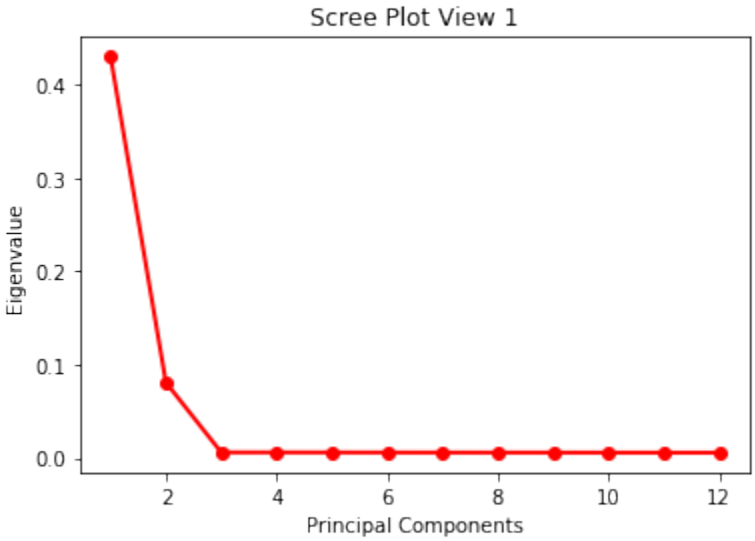
Scree plots allow us to observe variation and determine an appropriate initial signal rank for each view.

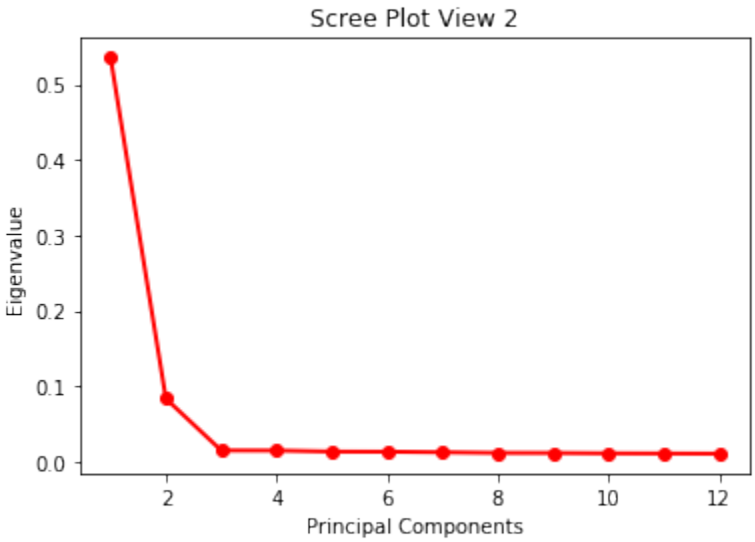
[3]:
```
def scree_plot(n,V,name):
    U, S, V = np.linalg.svd(V)
    eigvals = S**2 / np.sum(S**2)
    eigval_terms = np.arange(n) + 1
```

(continues on next page)

(continued from previous page)

```
    plt.plot(eigval_terms, eigvals[0:n], 'ro-', linewidth=2)
    plt.title('Scree Plot '+ name)
    plt.xlabel('Principal Components')
    plt.ylabel('Eigenvalue')
    plt.figure()

scree_plot(12,V1,'View 1')
scree_plot(12,V2,'View 2')
```






```
<Figure size 432x288 with 0 Axes>
```

Based on the scree plots, we fit AJIVE with both initial signal ranks set to 2.

[4]:
```
ajive1 = AJIVE(init_signal_ranks=[2,2])
ajive1.fit(Xs=[V1,V1], view_names=['x1','x2'])

ajive2 = AJIVE(init_signal_ranks=[2,2])
ajive2.fit(Xs=[V1,V2], view_names=['x','y'])
```

[4]:
```
joint rank: 1, block x indiv rank: 1, block y indiv rank: 1
```

Output Structure

The predict() function returns n dictionaries where n is the number of views fitted. Each dictionary has a joint, individual, and noise matrix taken from the AJIVE decomposition. The keys are 'joint', 'individual', and 'noise' and the values are the respective matrices.

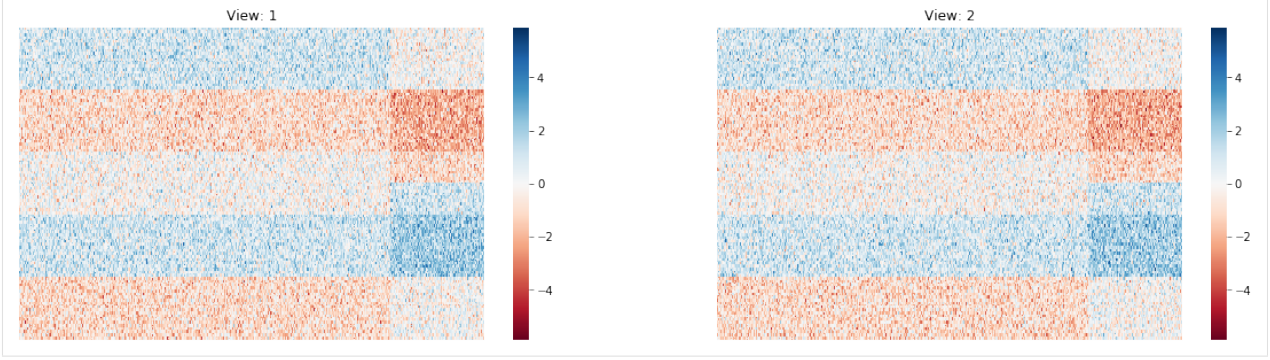
[5]:
```
blocks1 = ajive1.predict()
blocks2 = ajive2.predict()
```

Heatmap Visualizations

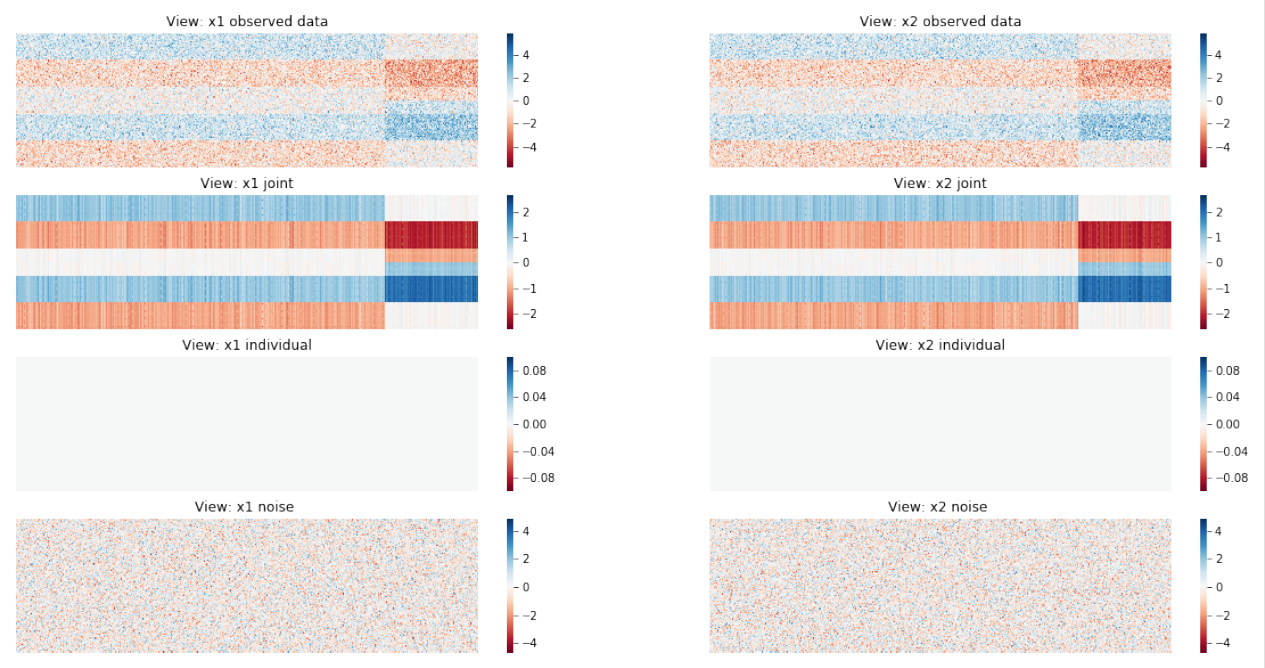
Here we are using heatmaps to visualize the decomposition of our views. As we can see when we use two of the same views there is no Individualized Variation displayed. When we create two different views, the algorithm finds different decompositions where common and individual structural artifacts can be seen in their corresponding heatmaps.

Same Views

[6]:
```
plt.figure(figsize=[20, 5])
data_block_heatmaps(Views_1)
```

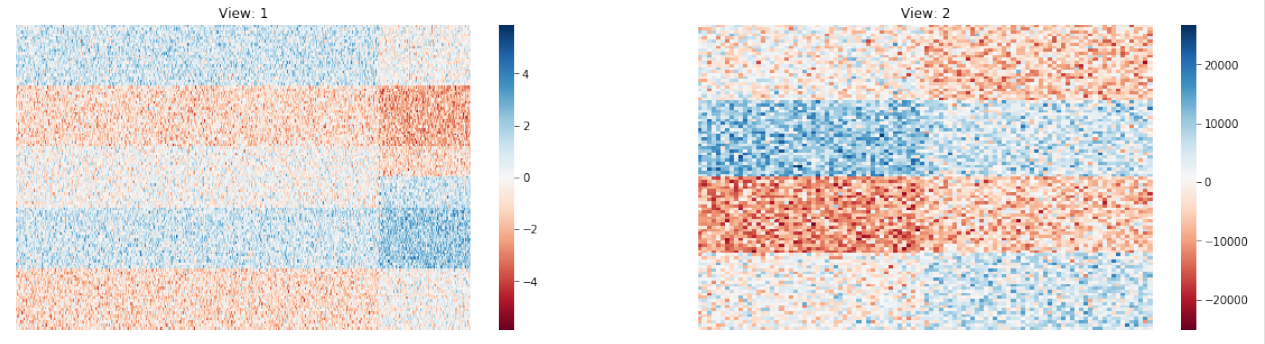


[7]:
```
plt.figure(figsize=[20, 10])
plt.title('Same Views')
ajive_full_estimate_heatmaps(Views_1, blocks1, names=['x1','x2'])
```

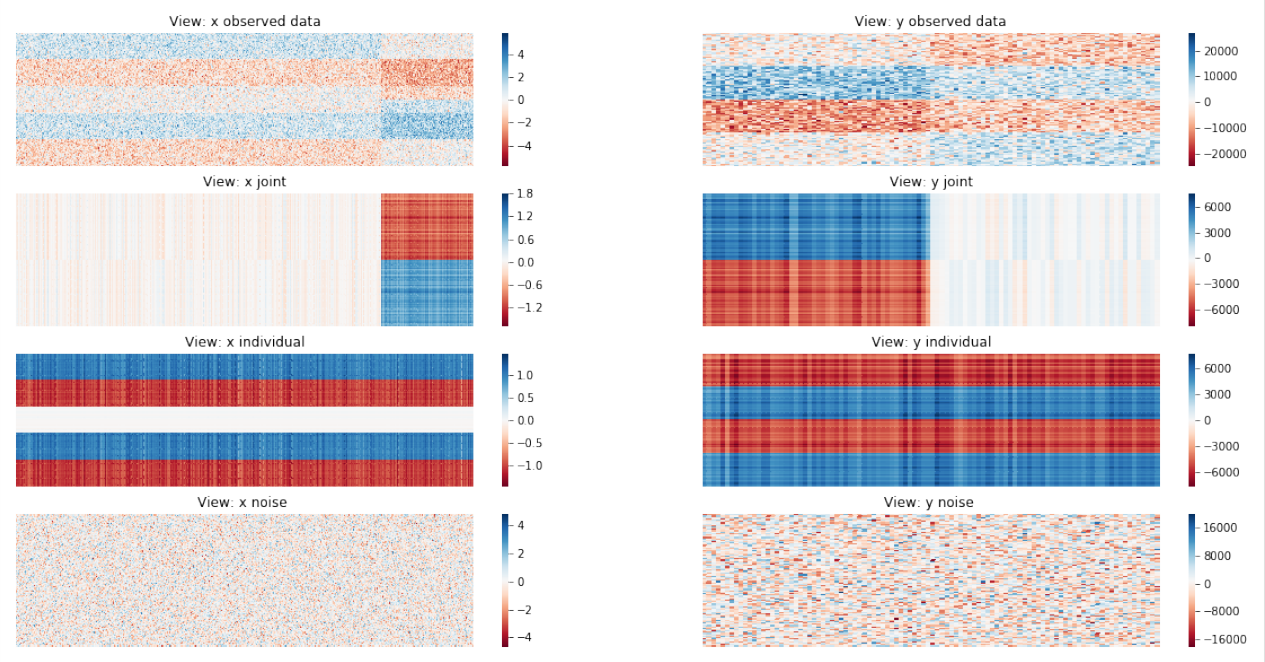


Different Views

[9]:
```
plt.figure(figsize=[20, 5])
data_block_heatmaps(Views_2)
```



[10]:
```
plt.figure(figsize=[20, 10])
ajive_full_estimate_heatmaps(Views_2, blocks2, names=['x','y'])
```



Multiview Independent Component Analysis (ICA) Tutorial

Adopted from the code at https://github.com/hugorichard/multiviewica and their tutorial written by:

Authors: Hugo Richard, Pierre Ablin

License: BSD 3 clause

Three multiview ICA algorithms are compared. GroupICA concatenates the individual views prior to dimensionality reduction and running ICA over the result. PermICA is more sensitive to individual discrepencies, and computes ICA on each view before aligning the reuslts using the hungarian algorithm. Lastly, MultiviewICA performs the best by optimizing the set of mixing matrices relative to the average source signal.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

from mvlearn.decomposition import MultiviewICA, PermICA, GroupICA
```

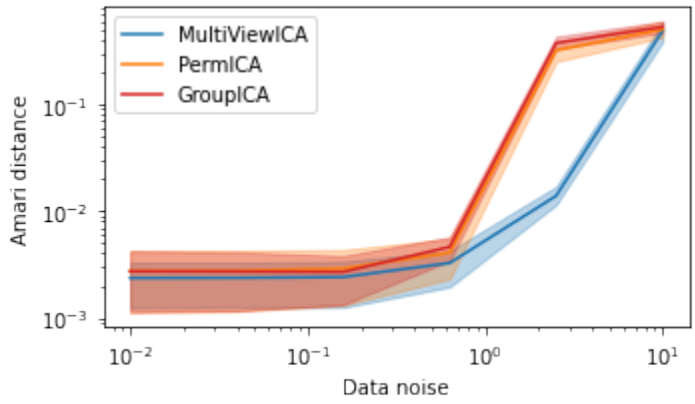
[3]:
```
# sigmas: data noise
# m: number of subjects
# k: number of components
# n: number of samples
sigmas = np.logspace(-2, 1, 6)
n_seeds = 3
m, k, n = 5, 3, 1000

cm = plt.cm.tab20
algos = [
    ("MultiViewICA", cm(0), MultiviewICA),
    ("PermICA", cm(2), PermICA),
    ("GroupICA", cm(6), GroupICA),
]
```

(continues on next page)

(continued from previous page)

```
def amari_d(W, A):
    P = np.dot(A, W)

    def s(r):
        return np.sum(np.sum(r ** 2, axis=1) / np.max(r ** 2, axis=1) - 1)

    return (s(np.abs(P.T)) + s(np.abs(P))) / (2 * P.shape[1])


plots = []
for name, color, algo in algos:
    means = []
    lows = []
    highs = []
    for sigma in sigmas:
        dists = []
        for seed in range(n_seeds):
            rng = np.random.RandomState(seed)
            S_true = rng.laplace(size=(n, k))
            A_list = rng.randn(m, k, k)
            noises = rng.randn(m, n, k)
            Xs = np.array([S_true.dot(A) for A in A_list])
            Xs += [sigma * N.dot(A) for A, N in zip(A_list, noises)]
            ica = algo(tol=1e-4, max_iter=1000, random_state=0).fit(Xs)
            W = ica.unmixings_
            dist = np.mean([amari_d(W[i], A_list[i]) for i in range(m)])
            dists.append(dist)
        dists = np.array(dists)
        mean = np.mean(dists)
        low = np.quantile(dists, 0.1)
        high = np.quantile(dists, 0.9)
        means.append(mean)
        lows.append(low)
        highs.append(high)
    lows = np.array(lows)
    highs = np.array(highs)
    means = np.array(means)
    plots.append((highs, lows, means))
```

[4]:
```
fig = plt.figure(figsize=(5, 3))
for i, (name, color, algo) in enumerate(algos):
    highs, lows, means = plots[i]
    plt.fill_between(
        sigmas, lows, highs, color=color, alpha=0.3,
    )
    plt.loglog(
        sigmas, means, label=name, color=color,
    )
plt.legend()
x_ = plt.xlabel(r"Data noise")
y_ = plt.ylabel(r"Amari distance")
fig.tight_layout()
plt.show()
```



MultiviewICA has the best performance (lowest Amari distance).

5.2.5 Plotting

Methods build on top of Matplotlib and Seaborn have been implemented for convenient plotting of multiview data. See examples of such plots on simulated data.

Using quick_visualize() to quickly understand multi-view data

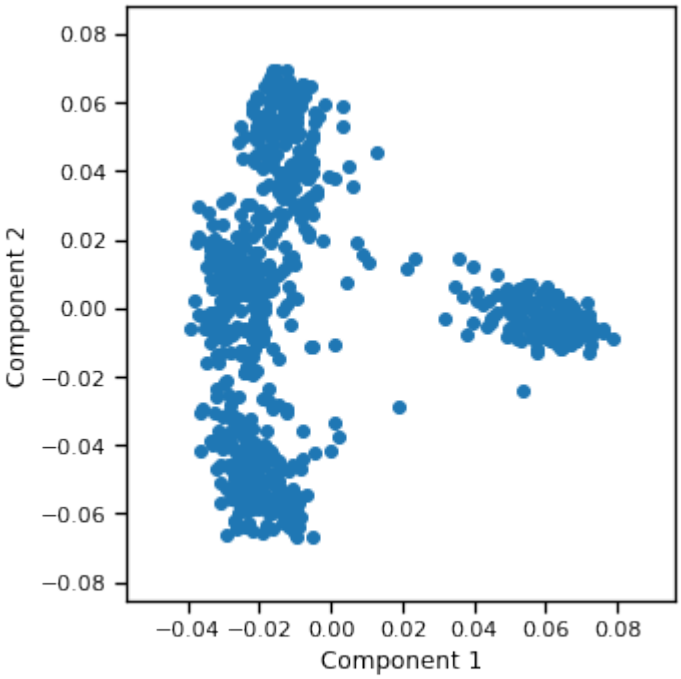
Easily view and understand underlying clusters in multi-view data

As a simple example, say we had high-dimensional multi-view data that we wanted to quickly visualize before we begin our analysis. With quick_visualize, we can easily do this. As an example, we will visualize the UCI Multiple Features dataset.

[1]:
```
# Import the function
from mvlearn.plotting import quick_visualize
from mvlearn.datasets import load_UCImultifeature

import matplotlib.pyplot as plt
%matplotlib inline
```

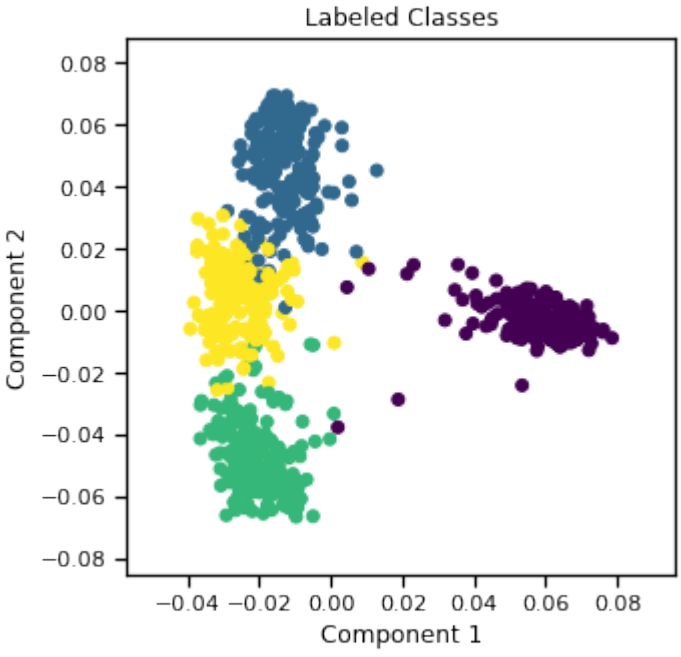
[2]:
```
# Load 4-class data
Xs, y = load_UCImultifeature(select_labeled=[0,1,2,3])
```

[3]:
```
# Quickly visualize the data
quick_visualize(Xs, figsize=(5,5))
```



If we have class labels that we want to visualize too, we can easily add those

[4]:
```
quick_visualize(Xs, labels=y, title='Labeled Classes', figsize=(5,5))
```



Plotting Across 2 Views

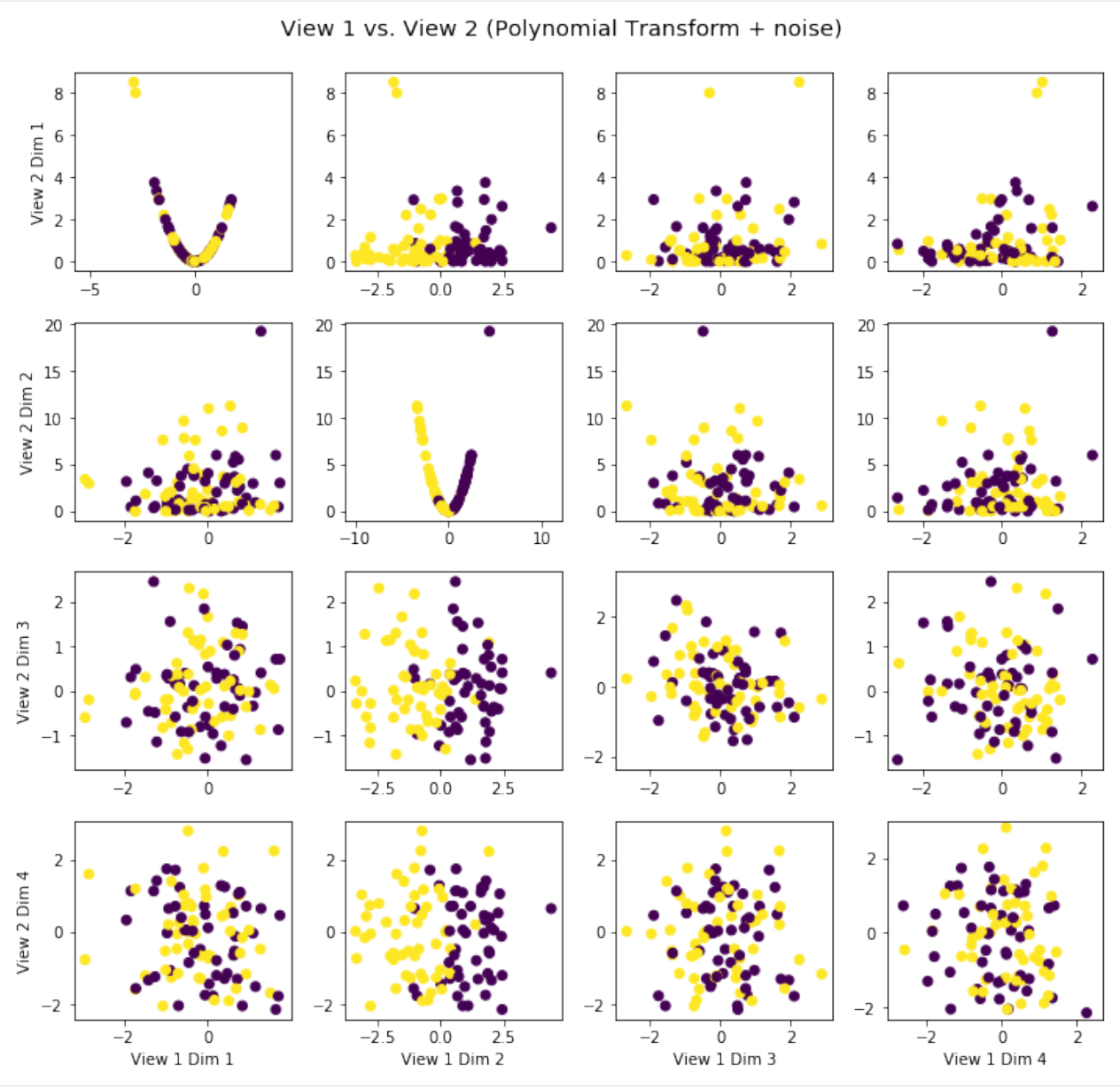
In many cases with multi-view data, especially after use of an embedding algorithm, one is interested in visualizing two views across dimensions. One use is assessing correlation between corresponding dimensions of views. Here, we use this function to display the relationship between two views simulated from transformations of multi-variant gaussians.

[1]:
```
from mvlearn.datasets import GaussianMixture
from mvlearn.plotting import crossviews_plot
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

[2]:
```
n_samples = 100
centers = [[0,1], [0,-1]]
covariances = [np.eye(2), np.eye(2)]
GM = GaussianMixture(n_samples, centers, covariances, shuffle=True)
GM = GM.sample_views(transform='poly', n_noise=2)
```

Below, we see that the first two dimensions are related by a degree 2 polynomial while the latter two dimensions are uncorrelated.

[3]:
```
crossviews_plot(GM.Xs_, labels=GM.y_, title='View 1 vs. View 2 (Polynomial Transform␣
↪+ noise)', equal_axes=True)
```



5.2.6 Test Dataset

In order to conviently run tools in this package on multview data, data can be simulated or be accessed from the publicly available UCI multiple features dataset using a dataloader in this package.

Loading and Viewing the UCI Multiple Features Dataset

[1]:
```
from mvlearn.datasets import load_UCImultifeature
```

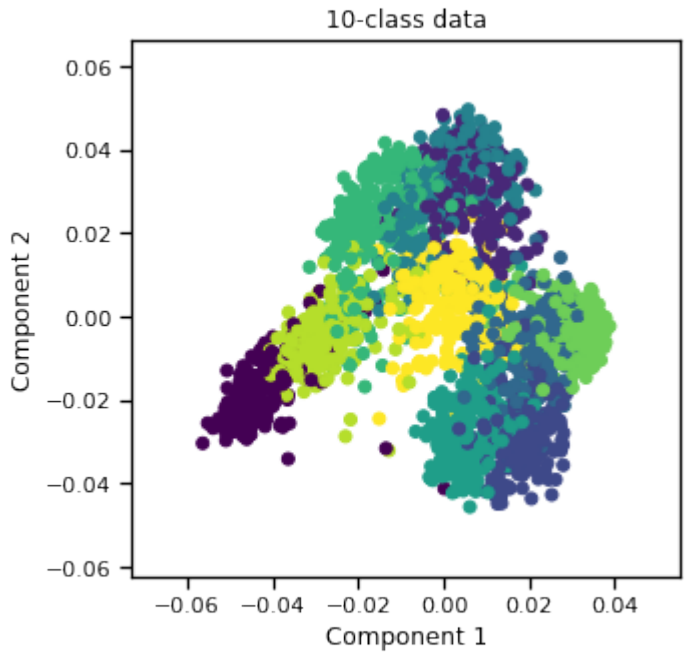
[2]:
```
# load the quick_visualize function for quick visualization in 2D
from mvlearn.plotting import quick_visualize
%matplotlib inline
```

Load the data and labels

We can either load the entire dataset (all 10 digits) or select certain digits. Then, visualize in 2D.

[3]:
```
# Load entire dataset
full_data, full_labels = load_UCImultifeature()

print("Full Dataset\n")
print("Views = " + str(len(full_data)))
print("First view shape = " + str(full_data[0].shape))
print("Labels shape = " + str(full_labels.shape))

quick_visualize(full_data, labels=full_labels, title="10-class data")
```

```
Full Dataset

Views = 6
First view shape = (2000, 76)
Labels shape = (2000,)
```



Load only 2 classes of the data

Also, shuffle the data and set the seed for reproducibility. Then, visualize in 2D.

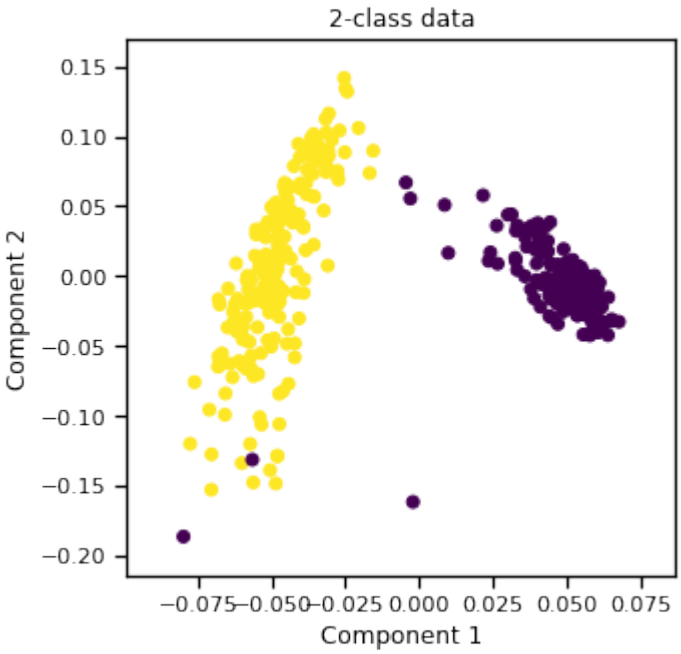
[4]:
```
# Load only the examples labeled 0 or 1, and shuffle them,
# but set the random_state for reproducibility
partial_data, partial_labels = load_UCImultifeature(select_labeled=[0,1],␣
↪shuffle=True, random_state=42)

print("\n\nPartial Dataset (only 0's and 1's)\n")
print("Views = " + str(len(partial_data)))
print("First view shape = " + str(partial_data[0].shape))
print("Labels shape = " + str(partial_labels.shape))
```

(continues on next page)

(continued from previous page)

```
quick_visualize(partial_data, labels=partial_labels, title="2-class data")
```

```
Partial Dataset (only 0's and 1's)

Views = 6
First view shape = (400, 76)
Labels shape = (400,)
```



Multiview Data from Gaussian Mixtures

In this example we show how to simulate multiview data from Gaussian mixtures and plot them using a crossviews plot.

[1]:
```
from mvlearn.datasets import GaussianMixture
from mvlearn.plotting import crossviews_plot
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

%load_ext autoreload
%autoreload 2
```

Latent variables are sampled from two multivariate Gaussians with equal prior probability. Then a polynomial transformation is applied and noise is added independently to both the transformed and untransformed latents.

[2]:
```
n_samples = 100
centers = [[0,1], [0,-1]]
covariances = [np.eye(2), np.eye(2)]
GM = GaussianMixture(n_samples, centers, covariances, random_state=42,
```

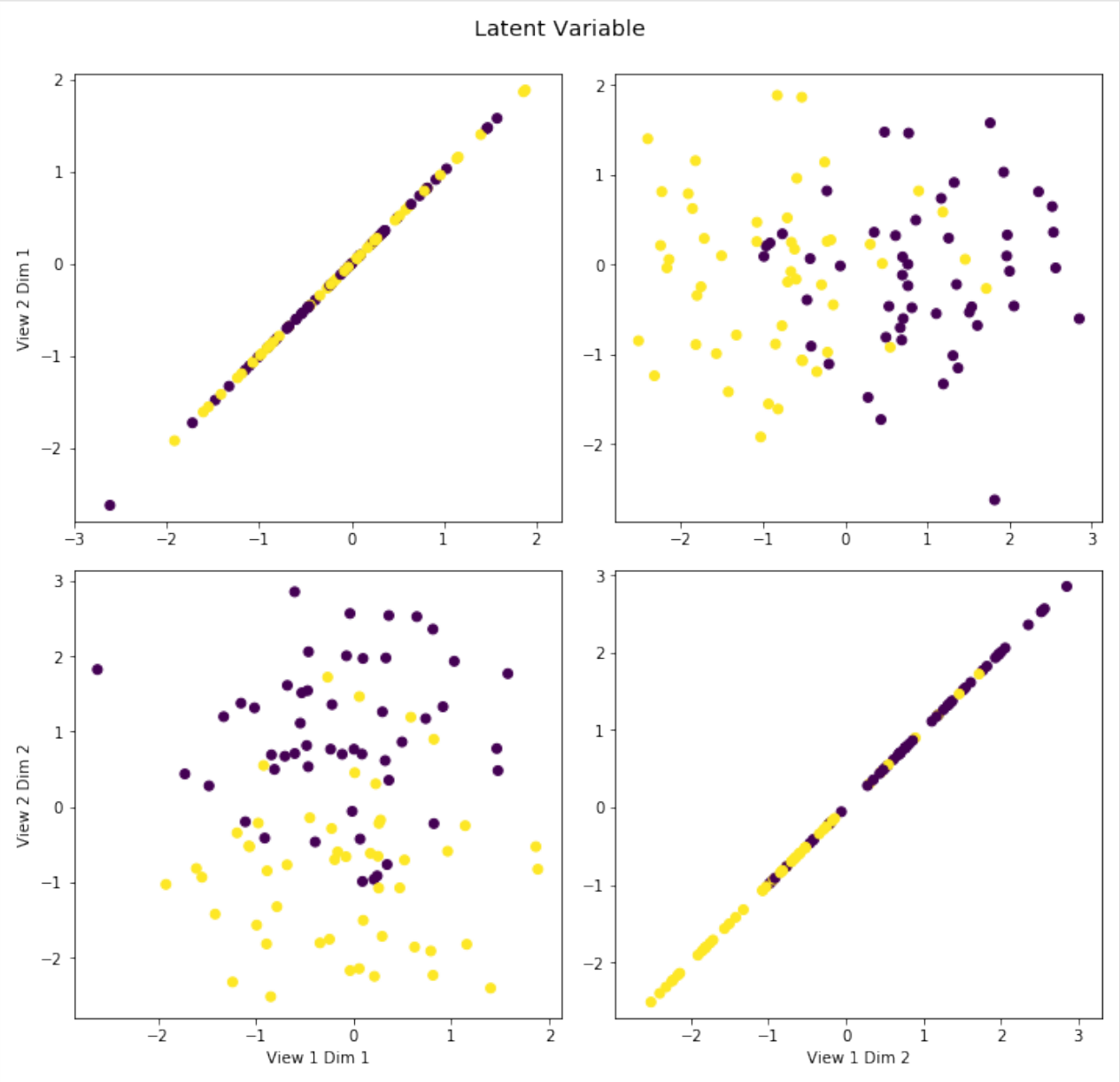
(continues on next page)

(continued from previous page)

```
                        shuffle=True, shuffle_random_state=42)
GM = GM.sample_views(transform='poly', n_noise=2)

latent,y = GM.get_Xy(latents=True)
Xs,_ = GM.get_Xy(latents=False)
```

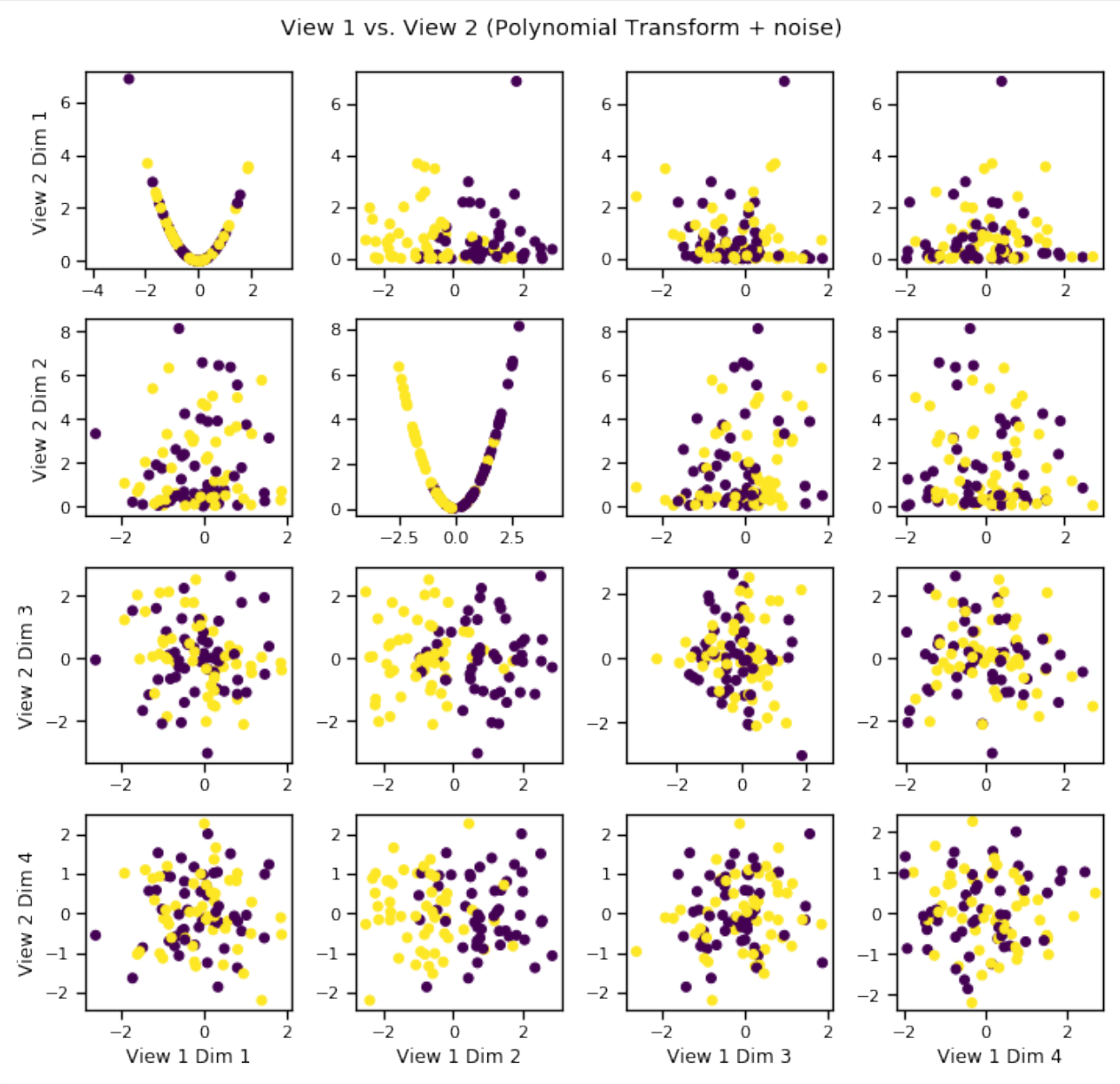
The latent data is plotted against itself to reveal the underlying distribtution.

[3]:
```
crossviews_plot([latent, latent], labels=y, title='Latent Variable', equal_axes=True)
```



The noisy latent variable (view 1) is plotted against the transformed latent variable (view 2), an example of a dataset with two views.

[4]:
```
crossviews_plot(Xs, labels=y, title='View 1 vs. View 2 (Polynomial Transform + noise)
↪', equal_axes=True)
```



5.3 Reference

The package is split up into submodules.

5.3.1 Clustering

Multiview Spectral Clustering

Co-Regularized Multiview Spectral Clustering

Multiview K Means

Multiview Spherical K Means

5.3.2 Semi-Supervised

Cotraining Classifier

Cotraining Regressor

5.3.3 View Embedding

Generalized Canonical Correlation Analysis

Kernel Canonical Correlation Analysis

Deep Canonical Correlation Analysis

Omnibus Embedding

Multiview Multidimensional Scaling

Split Autoencoder

DCCA Utilities

Dimension Selection

5.3.4 Decomposition

Multiview ICA

Permutation ICA

Group ICA

Angle-Based Joint and Individual Variation Explained (AJIVE)

AJIVE

AJIVE Plotting Functions

5.3.5 View Construction

Random Gaussian Projection

Read more about sklearn's implementation here.

Random Subspace Method

5.3.6 Multiview Datasets

UCI multiple feature dataset (located here)

Data Simulator

5.3.7 Plotting

Quick Visualize

Crossviews Plot

5.3.8 Utility Functions

IO

5.4 Contributing to mvlearn

(adopted from scikit-learn)

5.4.1 Submitting a bug report or a feature request

We use GitHub issues to track all bugs and feature requests; feel free to open an issue if you have found a bug or wish to see a feature implemented.

In case you experience issues using this package, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

It is recommended to check that your issue complies with the following rules before submitting:

- Verify that your issue is not being currently addressed by other issues or pull requests.
- If you are submitting a bug report, we strongly encourage you to follow the guidelines in *How to make a good bug report*.
- Always make sure your code follows the general *Guidelines* and adheres to the *API of mvlearn Objects*.

How to make a good bug report

When you submit an issue to Github, please do your best to follow these guidelines! This will make it a lot easier to provide you with good feedback:

- The ideal bug report contains a **short reproducible code snippet**, this way anyone can try to reproduce the bug easily (see this for more details). If your snippet is longer than around 50 lines, please link to a gist or a github repo.
- If not feasible to include a reproducible snippet, please be specific about what **estimators and/or functions are involved and the shape of the data**.
- If an exception is raised, please **provide the full traceback**.
- Please include your **operating system type and version number**, as well as your **Python and mvlearn versions**. This information can be found by running the following code snippet in Python.

```
import platform; print(platform.platform());
import sys; print("Python", sys.version);
import mvlearn; print("mvlearn", mvlearn.version)
```

- Please ensure all **code snippets and error messages are formatted in appropriate code blocks**. See Creating and highlighting code blocks for more details.

5.4.2 Contributing Code

The preferred workflow for contributing to mvlearn is to fork the main repository on GitHub, clone, and develop on a branch. Steps:

1. Fork the project repository by clicking on the 'Fork' button near the top right of the page. This creates a copy of the code under your GitHub user account. For more details on how to fork a repository see this guide.
2. Clone your fork of the mvlearn repo from your GitHub account to your local disk:

```
$ git clone git@github.com:YourLogin/mvlearn.git
$ cd mvlearn
```

3. Create a `feature` branch to hold your development changes:

```
$ git checkout -b my-feature
```

   Always use a `feature` branch. It's good practice to never work on the `master` branch!

4. Develop the feature on your feature branch. Add changed files using `git add` and then `git commit` files:

```
$ git add modified_files
$ git commit
```

   to record your changes in Git, then push the changes to your GitHub account with:

```
$ git push -u origin my-feature
```

Pull Request Checklist

We recommended that your contribution complies with the following rules before you submit a pull request:

- Follow the *coding-guidelines*.
- Give your pull request a helpful title that summarises what your contribution does. In some cases `Fix <ISSUE TITLE>` is enough. `Fix #<ISSUE NUMBER>` is not enough.
- All public methods should have informative docstrings with sample usage presented as doctests when appropriate.
- At least one paragraph of narrative documentation with links to references in the literature (with PDF links when possible) and the example.
- All functions and classes must have unit tests. These should include, at the very least, type checking and ensuring correct computation/outputs.
- Ensure all tests are passing locally using `pytest`. Install the necessary packages by:

```
$ pip install pytest pytest-cov
```

   then run

```
$ pytest
```

or you can run pytest on a single test file by

```
$ pytest path/to/test.py
```

- Run an autoformatter to conform to PEP 8 style guidelines. We use `black` and would like for you to format all files using `black`. You can run the following lines to format your files.

```
$ pip install black
$ black path/to/module.py
```

5.4.3 Guidelines

Coding Guidelines

Uniformly formatted code makes it easier to share code ownership. mvlearn package closely follows the official Python guidelines detailed in PEP8 that detail how code should be formatted and indented. Please read it and follow it.

Docstring Guidelines

Properly formatted docstrings is required for documentation generation by Sphinx. The pygraphstats package closely follows the numpydoc guidelines. Please read and follow the numpydoc guidelines. Refer to the example.py provided by numpydoc.

5.4.4 API of mvlearn Objects

Estimators

The main mvlearn object is the estimator and its documentation draws mainly from the formatting of sklearn's estimator object. An estimator is an object that fits a set of training data and generates some new view of the data. Each module in mvlearn contains a main base class (found in `module_name.base`) which all estimators in that module should implement. Each of these base classes implements sklearn.base.BaseEstimator. If you are contributing a new estimator, be sure that it properly implements the base class of the module it is contained within.

When contributing, borrow from sklearn requirements as much as possible and utilize their checks to automatically check the suitability of inputted data, or use the checks available in `mvlearn.utils` such as `check_Xs`.

Instantiation

An estimator object's `__init__` method may accept constants that determine the behavior of the object's methods. These constants should not be the data nor should they be data-dependent as those are left to the `fit` method. All instantiation arguments are keyworded and have default values. Thus, the object keeps these values across different method calls. Every keyword argument accepted by `__init__` should correspond to an instance attribute and there should be no input validation logic on instantiation, as that is left to `fit`. A correct implementation of `__init__` looks like

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

Fitting

All estimators should implement the `fit(Xs, y=None)` method to make some estimation, which is called with:

```
estimator.fit(Xs, y)
```

or

```
estimator.fit(Xs)
```

The former case corresponds to the supervised case and the latter to the unsupervised case. In unsupervised cases, y takes on a default value of `None` and is ignored. Xs corresponds to a list of data matrices and y to a list of sample labels. The samples across views in Xs and y are matched. Note that data matrices in Xs must have the same number of samples (rows) but the number of features (columns) may differ.

Parameters	Format
Xs	**list of array-likes:** • Xs shape: (n_views,) • Xs[i] shape: (n_samples, n_features_i)
y	array, shape (n_samples,)
kwargs	optional data-dependent parameters.

The `fit` method should return the object (`self`) so that simple one line processes can be written.

All attributes calculated in the `fit` method should be saved with a trailing underscore to distinguish them from the constants passes to `__init__`. They are overwritten every time `fit` is called.

Additional Functionality

Transformers and Predictors

A `transformer` object modifies the data it is given. An estimator may also be a transformer that learns the transformation parameters. The transformer object implements the `transform` method, i.e.

```
new_data = transformer.transform(Xs)
```

or if the fit method must be called first,

```
new_data = transformer.fit_transform(Xs, y)
```

It may be more efficient in some cases to compute the latter example rather than call `fit` and `transform` separately.

Similarly, a `predictor` object makes predictions based on the data it is given. An estimator may also be a predictor that learns the prediction parameters. The predictor object implements the `predict` method, i.e.

```
predictions = predictor.predict(Xs)
```

or if the fit method must be called first,

```
predictions = predictor.fit_predict(Xs, y)
```

It may be more efficient in some cases to compute the latter example rather than call `fit` and `predict` separately.

5.5 Changelog

5.5.1 Version 0.3.0

Updates in this release:

- `cotraining` module changed to `semi_supervised`.
- `factorization` module changed to `decomposition`.
- A new class within the `semi_supervised` module, `CTRegressor`, and regression tool for 2-view semi-supervised learning, following the cotraining framework.
- Three multiview ICA methods added: MultiviewICA, GroupICA, PermICA with `python-picard` dependency.
- Added parallelizability to GCCA using joblib and added `partial_fit` function to handle streaming or large data.
- Adds a function (get_stats()) to perform statistical tests within the `embed.KCCA` class so that canonical correlations and canonical variates can be robustly. assessed for significance. See the documentation in Reference for more details.
- Adds ability to select which views to return from the UCI multiple features dataset loader, `datasets.UCI_multifeature`.
- API enhancements including base classes for each module and algorithm type, allowing for greater flexibility to extend `mvlearn`.
- Internals of `SplitAE` changed to snake case to fit with the rest of the package.
- Fixes a bug which prevented the `visualize.crossviews_plot` from plotting when each view only has a single feature.
- Changes to the `mvlearn.datasets.gaussian_mixture.GaussianMixture` parameters to better mimic sklearn's datasets.
- Fixes a bug with printing error messages in a few classes.

5.5.2 Patch 0.2.1

Fixed missing `__init__.py` file in the `ajive_utils` submodule.

5.5.3 Version 0.2.0

Updates in this release:

- `MVMDS` can now also accept distance matrices as input, rather than only views of data with samples and features
- A new clustering algorithm, `CoRegMultiviewSpectralClustering` - co-regularized multi-view spectral clustering functionality
- Some attribute names slightly changed for more intuitive use in `DCCA`, `KCCA`, `MVMDS`, `CTClassifier`
- Option to use an Incomplete Cholesky Decomposition method for `KCCA` to reduce up computation times
- A new module, `factorization`, containing the `AJIVE` algorithm - angle-based joint and individual variance explained
- Fixed issue where signal dimensions of noise were dependent in the GaussianMixtures class

- Added a dependecy to `joblib` to enable parallel clustering implementation
- Removed the requirements for `torchvision` and `pillow`, since they are only used in tutorials

5.5.4 Version 0.1.0

We're happy to announce the first major stable version of `mvlearn`. This version includes multiple new algorithms, more utility functions, as well as significant enhancements to the documentation. Here are some highlights of the big updates.

- Deep CCA, (`DCCA`) in the `embed` module
- Updated `KCCA` with multiple kernels
- Synthetic multi-view dataset generator class, `GaussianMixture`, in the `datasets` module
- A new module, `plotting`, which includes functions for visualizing multi-view data, such as `crossviews_plot` and `quick_visualize`
- More detailed tutorial notebooks for all algorithms

Additionally, mvlearn now makes the `torch` and `tqdm` dependencies optional, so users who don't need the DCCA or SplitAE functionality do not have to import such a large package. **Note** this is only the case for installing with pip. Installing from `conda` includes these dependencies automatically. To install the full version of mvlearn with `torch` and `tqdm` from pip, you must include the optional torch in brackets:

```
pip3 install mvlearn[torch]
```

or

```
pip3 install --upgrade mvlearn[torch]
```

To install **without** `torch`, do:

```
pip3 install mvlearn
```

or

```
pip3 install --upgrade mvlearn
```

5.6 License

mvlearn is distributed with Apache 2.0 license.

```
Apache License
                        Version 2.0, January 2004
                        http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

    "License" shall mean the terms and conditions for use, reproduction,
    and distribution as defined by Sections 1 through 9 of this document.

    "Licensor" shall mean the copyright owner or entity authorized by
```

(continues on next page)

(continued from previous page)

```
    the copyright owner that is granting the License.

    "Legal Entity" shall mean the union of the acting entity and all
    other entities that control, are controlled by, or are under common
    control with that entity. For the purposes of this definition,
    "control" means (i) the power, direct or indirect, to cause the
    direction or management of such entity, whether by contract or
    otherwise, or (ii) ownership of fifty percent (50%) or more of the
    outstanding shares, or (iii) beneficial ownership of such entity.

    "You" (or "Your") shall mean an individual or Legal Entity
    exercising permissions granted by this License.

    "Source" form shall mean the preferred form for making modifications,
    including but not limited to software source code, documentation
    source, and configuration files.

    "Object" form shall mean any form resulting from mechanical
    transformation or translation of a Source form, including but
    not limited to compiled object code, generated documentation,
    and conversions to other media types.

    "Work" shall mean the work of authorship, whether in Source or
    Object form, made available under the License, as indicated by a
    copyright notice that is included in or attached to the work
    (an example is provided in the Appendix below).

    "Derivative Works" shall mean any work, whether in Source or Object
    form, that is based on (or derived from) the Work and for which the
    editorial revisions, annotations, elaborations, or other modifications
    represent, as a whole, an original work of authorship. For the purposes
    of this License, Derivative Works shall not include works that remain
    separable from, or merely link (or bind by name) to the interfaces of,
    the Work and Derivative Works thereof.

    "Contribution" shall mean any work of authorship, including
    the original version of the Work and any modifications or additions
    to that Work or Derivative Works thereof, that is intentionally
    submitted to Licensor for inclusion in the Work by the copyright owner
    or by an individual or Legal Entity authorized to submit on behalf of
    the copyright owner. For the purposes of this definition, "submitted"
    means any form of electronic, verbal, or written communication sent
    to the Licensor or its representatives, including but not limited to
    communication on electronic mailing lists, source code control systems,
    and issue tracking systems that are managed by, or on behalf of, the
    Licensor for the purpose of discussing and improving the Work, but
    excluding communication that is conspicuously marked or otherwise
    designated in writing by the copyright owner as "Not a Contribution."

    "Contributor" shall mean Licensor and any individual or Legal Entity
    on behalf of whom a Contribution has been received by Licensor and
    subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    copyright license to reproduce, prepare Derivative Works of,
```

(continues on next page)

(continued from previous page)

```
    publicly display, publicly perform, sublicense, and distribute the
    Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    (except as stated in this section) patent license to make, have made,
    use, offer to sell, sell, import, and otherwise transfer the Work,
    where such license applies only to those patent claims licensable
    by such Contributor that are necessarily infringed by their
    Contribution(s) alone or by combination of their Contribution(s)
    with the Work to which such Contribution(s) was submitted. If You
    institute patent litigation against any entity (including a
    cross-claim or counterclaim in a lawsuit) alleging that the Work
    or a Contribution incorporated within the Work constitutes direct
    or contributory patent infringement, then any patent licenses
    granted to You under this License for that Work shall terminate
    as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
    Work or Derivative Works thereof in any medium, with or without
    modifications, and in Source or Object form, provided that You
    meet the following conditions:

    (a) You must give any other recipients of the Work or
        Derivative Works a copy of this License; and

    (b) You must cause any modified files to carry prominent notices
        stating that You changed the files; and

    (c) You must retain, in the Source form of any Derivative Works
        that You distribute, all copyright, patent, trademark, and
        attribution notices from the Source form of the Work,
        excluding those notices that do not pertain to any part of
        the Derivative Works; and

    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
```

(continues on next page)

(continued from previous page)

```
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
    any Contribution intentionally submitted for inclusion in the Work
    by You to the Licensor shall be under the terms and conditions of
    this License, without any additional terms or conditions.
    Notwithstanding the above, nothing herein shall supersede or modify
    the terms of any separate license agreement you may have executed
    with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
    names, trademarks, service marks, or product names of the Licensor,
    except as required for reasonable and customary use in describing the
    origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
    agreed to in writing, Licensor provides the Work (and each
    Contributor provides its Contributions) on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied, including, without limitation, any warranties or conditions
    of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
    PARTICULAR PURPOSE. You are solely responsible for determining the
    appropriateness of using or redistributing the Work and assume any
    risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
    whether in tort (including negligence), contract, or otherwise,
    unless required by applicable law (such as deliberate and grossly
    negligent acts) or agreed to in writing, shall any Contributor be
    liable to You for damages, including any direct, indirect, special,
    incidental, or consequential damages of any character arising as a
    result of this License or out of the use or inability to use the
    Work (including but not limited to damages for loss of goodwill,
    work stoppage, computer failure or malfunction, or any and all
    other commercial damages or losses), even if such Contributor
    has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
    the Work or Derivative Works thereof, You may choose to offer,
    and charge a fee for, acceptance of support, warranty, indemnity,
    or other liability obligations and/or rights consistent with this
    License. However, in accepting such obligations, You may act only
    on Your own behalf and on Your sole responsibility, not on behalf
    of any other Contributor, and only if You agree to indemnify,
    defend, and hold each Contributor harmless for any liability
    incurred by, or claims asserted against, such Contributor by reason
    of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

    To apply the Apache License to your work, attach the following
    boilerplate notice, with the fields enclosed by brackets "[]"
    replaced with your own identifying information. (Don't include
    the brackets!)  The text should be enclosed in the appropriate
```

(continues on next page)

(continued from previous page)

```
    comment syntax for the file format. We also recommend that a
    file or class name and description of purpose be included on the
    same "printed page" as the copyright notice for easier
    identification within third-party archives.

Copyright 2020 Richard Guo, Ronan Perry, Gavin Mischler, Theo Lee,
    Alexander Chang, Arman Koul, Cameron Franz

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

CHAPTER 6

Indices and tables

- genindex
- search